Fundamentals of Artificial Intelligence and Knowledge Representation

Module 4

Matteo Donati

October 4, 2021

Contents

Chapter 1

Introduction

A field of artificial intelligence is devoted to represent and reasoning over knowledge bases. Many different paradigms are available for representing knowledge and reasoning upon it. The main approach studied in this module is a rule-based approach. In particular, reasoning upon rules is a form of reasoning which is common between humans.

Chapter 2

Prolog - A (not so) Quick Recall

2.1 Terminology

- A Prolog **program** is a set of definite clauses of the following form:

Fact:	`A.`
Rule:	`A :- B1, B2, ..., Bn`
Goal:	`:- B1, B2, ..., Bn`

  where:

  - `A` is the head of the clause.
  - `:- B1, B2, ..., Bn` is the body of the clause.
  - The symbol `,` is the conjunction symbol.
  - The symbol `:-` is the logical implication symbol where `A` is the consequent and `:- B1, B2, ..., Bn` is the antecedent.

  Moreover, variables within a clause are universally quantified.

- An **atomic formula** has the following form:

  ```
  p(t1, t2, ..., tn)
  ```

  where `p` is a predicate symbol (alphanumeric string starting with a low-capital letter) and `t1, t2, ..., tn` are terms.

- A **term** is recursively defined as:

  - **Constants** (i.e. integer/floating point numbers, alphanumeric strings starting with a low-capital letter) are terms.

– **Variables** (i.e. alphanumeric strings starting with a capital letter or starting with the symbol `...`) are terms.

– `f(t1, t2, ..., tk)` is a term if `f` is a **function symbol** with k arguments where every argument is a term. Moreover, constants can be viewed as functions with zero zero arguments (i.e. with arity zero).

2.2 Execution of a Prolog Program

Prolog uses SLD (left to right, depth-first search)

A computation is the attempt to prove. through resolution, the a formula logically follows from the program (i.e. it is a theorem). Moreover, it aims to determine a substitution for the variables in the goal, for which the goal logically follows from the program. In particular, given a program `P` and the following goal/query:

```
:- p(t1, t2, ..., tm).
```

if X_1, X_2, ..., X_n are the variables appearing in `t1, t2, ..., tm`, the objective is to determine a substitution:

$$\sigma = \{X_1/\texttt{s1}, X_2/\texttt{s2}, \ldots, X_n/\texttt{sn}\}$$

where `si` are terms, such that `P` $\models$ `[p(t1, t2, ..., tm)]`σ.

2.3 Procedural Interpretation of Prolog

- A **procedure** is a set of clauses of a program `P` whose heads have the same predicate symbol and whose predicate symbols have the same arity.
- The arguments which appear in the head of a given procedure are the **formal parameters**. The following query:

  ```
  :- p(t1, t2, ..., tn).
  ```

  can be viewd as the **call** to precedure `p`. The arguments of `p` are the **actual parameters**.
- Unification can be seen as the mechanism for parameters passing.
- There is no distinction between input parameters and output parameters.
- Two clauses with the same head are two alternative definitions of the body of a procedure.
- All the variables are subject to "single assignment" (i.e. singleton) e.g. X is Z+3, X is X+1 is wrong

2.4 Arithmetic

- Both integers and floating point numbers are atoms.
- Math operators are function symbols predefined in the language.
- Every expression is a term.
- Binary operators are supported with a prefix notation, as well as the infix notation (e.g. `+(2, 3)` and `(2 + 3)` are equivalent).
- The expression evaluation is done by using the `is` predicate. For example,

  ```
  T is Expr
  ```

  The expression `Expr` is evaluated and the result is unified with `T`. Moreover, a term representing an expression is evaluated only if it is the second argument of a predicate `is`.
- The relational operators are the following:

  `>`, `<`, `>=`, `=<` , `==`, `=/=`

  Such operators can be used as goals within clauses and have a infix notation.
- Given the math operators and the `is` operator it is possible to define functions. In particular, given a function f with arity n, the corresponding Prolog function can be implemented through a $(n+1)$-arity predicate:

  $f : x_1, x_2, \ldots, x_n \to y$ is written as `f(X1, X2, ..., Xn, Y)`

2.5 Iteration and Recursion in Prolog

In Prolog there is no iteration but one can get an iterative behaviour through recursion. In particular:

- A function `f` is tail-recursive if it is the most external call in the recursive definition (i.e. is the result of the recursive call is not subject of any other call).
- Tail-recursion is equivalent to iteration: it can be evaluated in constant space.
- There are many cases where a non-tail recursion can be re-written as a tail recursion.

2.6 Controlling a Prolog Program

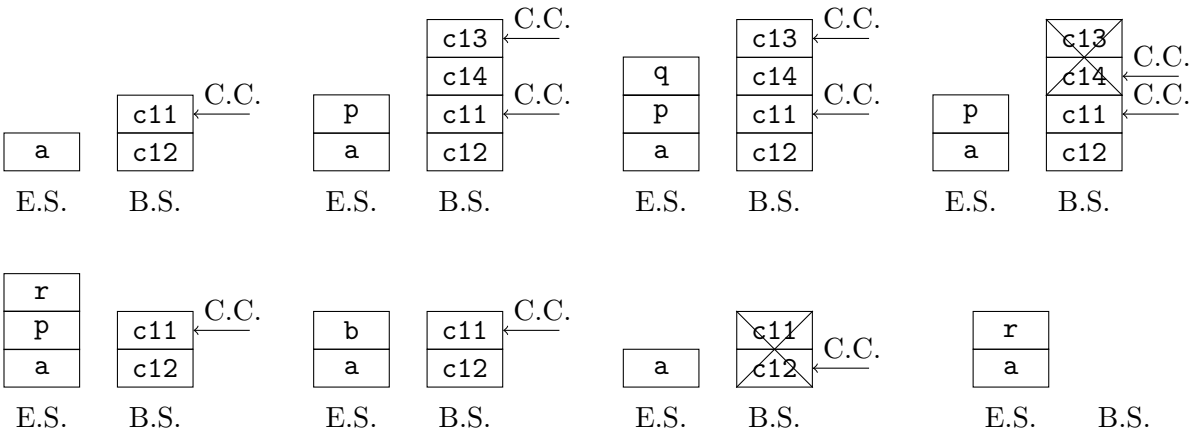
There exist a number of pre-defined predicates that allows to interfere and control the execution process of a goal. The **CUT** predicate is among them. This predicate has no logic meaning and no declarative semantics but it heavily affects the execution process. In particular:

- The execution process is built upon, at least, two stacks:
  - The **execution stack**, which contains the activation records of the procedures/predicates.
  - The **backtracking stack**, which contains the set of open choice points.

  For example, given the following program:

```
(c11)  a :- p, b.
(c12)  a :- r.
(c13)  p :- q.
(c14)  p :- r.
(c15)  r.
```

  and the query `:- a.`, the configurations of the two stacks are the following:



  where "E.S." is the execution stack, "B.S." is the backtracking stack and "C.C." stands for "current choice".

- The evaluation of the CUT is to make some choices as definitive and non-backtrackable (i.e. some blocks are removed from the backtracking stack). For example, considering the following clause:

```
p :- q1, q2, ..., qi, !, qi+1, qi+2, ..., qn.
```

*
e.g. p(X) :- q(X), r(X).
q(1).
q(2).
r(2).
?- p(X). X = 2

p(X) :- q(X), !, r(X).
;
false

– The evaluation of `!` succeeds and it is ignored during backtracking.
– All the choices made in the evaluation of goals `q1, q2, ..., qi` and goal `p` are made definitive (i.e. the choice points are removed from the backtracking stack).
– Alternative choices related to goals placed after the cut are not touched nor modified.
– If the evaluation of `qi+1, qi+2, ..., qn` fails, then `p` fails. Even if there were other alternatives for `p` these would have been removed by the cut. *

2.7 The Negation

Prolog allows only definite clauses and not negative literals. Moreover, it does not work upon the Close World Assumption (i.e. if a ground atom A is not logical consequence of a program `P`, ~~the~~ then one can infer $\sim A$), but adopts a less powerful rule, the Negation as Failure:

- Negation as Failure derives only the negation of atoms whose proof terminates with failure in a finite time.
- Given a program `P`, one can name $FF(\texttt{P})$ the set of atoms for which the proof fails in a finite time.
- The Negation as Failure rule is then defined as follows:

$$NF(\texttt{P}) = \{\sim A | A \in FF(\texttt{P})\} \tag{2.1}$$

- If an atom A belongs to $FF(\texttt{P})$, then A is not logical consequence of `P` but not all the atoms that are not logical consequence of `P` ~~beong~~ belong to $FF(\texttt{P})$. For example, given the following program:

```
city(rome) :- city(rome).
city(bologna).
```

`city(rome)` is not logical consequence of `P` but it does not belong to $FF(\texttt{P})$.

2.7.1 SLDNF

To solve goals containing also negative atoms, SLDNF has been proposed. It extends SLD resolution with Negation as Failure and it is used in Prolog to implement the Negation as Failure rule. In particular:

- Let `:- L1, ..., Lm` be the ~~foal~~ goal, where `L1, ..., Lm` are literals (i.e. atoms or negation of atoms).
- A SLDNF step proceeds as follows:

  1. Do not select any negative literal `Li` if it is not a ground literal.

2. If the selected literal `Li` is positive, then apply a normal SLD step.
3. If `Li` is of the form $\sim A$, with A being a ground literal, and A fails in a finite time (i.e. it has a finite failure SLD tree), then `Li` succeeds and the new resolvent is:

```
:- L1, ..., Li-1, Li+1, ..., Lm
```
(2.2)

- To prove $\sim A$, where A is an atom, the Prolog interpreter tries to prove A:
  - If the proof for A succeed, then the proof of $\sim A$ fails.
  - If the proof for A fails in a finite time, then $\sim A$ is proved successfully.

Chapter 3

Prolog - Meta-Predicates

work on the program itself (program as data)

In Prolog, predicates (i.e. programs) and terms (i.e. data) have the same syntactical structure. As a consequence, predicates and terms can be exchanged and exploited in different roles. There are several pre-defined predicates that allow to deal with these structures.

3.1 The `call` Predicate

The predicate `call` takes as input a term which can be interpreted as a predicate. For example, given the following program:

```
p(a).
q(X) :- p(X).
```

and the query `:- call(q(Y)).` the result would be `yes Y = a.`. When executed, `call` asks the Prolog interpreter to prove `q(Y)`. Since terms have the same syntactical structure of predicates, it is possible to create a term in a proper way and then execute it (e.g. `call(T)`, where `T` is considered a predicate).

3.2 The `fail` Predicate

The `fail` predicate takes no arguments (i.e. has arity equal to zero) and its evaluation always fails. As a consequence, it forces the interpreter to explore other alternatives. This predicate is mainly used to:

- Obtain some form of iteration over data.
- Implement the negation as failure.
- Implement a logical implication.

3.3 The `setof` and `bagof` Predicates

In Prolog, the usual query `:- p(X).` is interpreted with `X` existentially quantified. As a result, it is returned a possible substitution for variables of `p` such that the query is satisfied. However, sometimes it might be interesting to answer second-order queries like "which is the set S of element `X` such that `p(X)` is *true*?". Some Prolog interpreters support this type of queries by providing pre-defined predicates. In particular:

- `setof(X, P, S).`, where `S` is the set of instances `X` which satisfy the goal `P`.
- `bagof(X, P, L).`, where `L` is the list of instances `X` which satisfy the goal `P`.

The difference between these predicates is that `bagof` returns a list which can contain repetitions, while `setof` does not contain any repetitions. For example, considering the following knowledge base:

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).
```

the query `:- setof(X, p(X), S).` will return: `yes S = [0, 1, 2] X = X`, while the query `:- bagof(X, p(X), L).` will return: `yes L = [1, 2, 0, 1] X = X`.

3.4 The `findall` Predicate

The predicate `findall(X, P, S)` is *true* if `S` is the list of instance `X`, without repetitions, for which perdicate `P` is *true*. If there is no `X` satisfying `P`, then `findall` returns an empty list. For example, considering the following knowledge base:

```
father(giovanni, mario).
father(giovanni, giuseppe).
father(mario, paola).
father(mario, aldo).
father(giuseppe, maria).
```

the query `findall(X, father(X, Y), S).` would return: `yes S = [giovanni, mario, giuseppe] X = X Y = Y`.

3.5 The `clause` Predicate

The predicate `clause(Head, Body).` is *true* if `(Head, Body)` is unified with a clause stored within the database program. If more clauses with the same head are available, its evaluation opens choice points. For example:

```
p(1).
q(X, a) :- p(X), r(a).
q(2, Y) :- d(Y).
```

```
?- clause(p(1), BODY).
   yes   BODY = true
?- clause(p(X), true).
   yes   X = 1
?- clause(q(X, Y), BODY).
   yes   X = _1   Y = a    BODY = p(_1), r(a);
         X = 2    Y = _2   BODY = d(_2);
   no
?- clause(HEAD, true).
   Error - invalid key to data-base
```

3.6 Other Predicates

Below are some other predicates:

- `var(Term)`, which is *true* if `Term` is currently a variable.
- `nonvar(Term)`, which is *true* if `Term` is currently not a free variable.
- `number(Term)`, which is *true* if `Term` is currently a number.
- `ground(Term)`, which is *true* if `Term` holds no free variables.

Chapter 4

Meta-program: program which takes in input other programs

Prolog - Meta-Interpreters

Meta-interpreters are Prolog programs which works with other programs. These are used for the rapid prototyping of interpreters of symbolic languages. In Prolog, a meta-interpreter for a language L is defined as an interpreter for L, but written in Prolog.

4.1 Vanilla Meta-Interpreter

The vanilla meta-interpreter defines a predicate `solve` which takes as input a goal and returns *true* if and only if the goal is provable using the current knowledge base:

this rule can be applied with any conjunction of literals, since comma operator is binary: $A \wedge B \wedge c \equiv (A, (B, c)) \equiv (A, B)$

one could invert these two in order to select the rightmost rule

```
solve(true) :- !.
solve((A, B)) :- !, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

For example, if one wants to define a Prolog interpreter `solve(Goal, Step)` that:

- It is *true* is `Goal` can be proven.
- In case `Goal` is proved, `Step` is the number of resolution steps used to prove such goal. In case of conjunctions, the number of steps is defined as the sum of the steps needed for each atomic conjunct.

one should define the following meta-interpreter:

```
solve(true, 0) :- !.
solve((A, B), S) :- !, solve(A, SA), solve(B, SB), S is SA + SB.
solve(A, S) :- clause(A, B), solve(B, SB), S is 1 + SB.
```

Given the program:

```
a :- b, c.
b :- d.
c.
d.
```

The query `?- solve(a, Step)` would output `yes Step = 4.` *

(depth of the search tree

* other examples in slides

4.2 Dynamically Modifying the Program

When a Prolog program is loaded, its representation in terms of data structures (terms) is loaded into a table in memory. Such table is often referred as the program database. In order to modify the entries of such table, the following predicates can be used: └ it is a real database

- `assert(T)`, this predicates adds the clause `T` to the program database. In particular:
  - `T` must be instantiated to a term denoting a clause (i.e. a fact or a rule).
  - `T` is added in a non-specified position. With predicate `asserta(T)` the specific clause is added at the beginning of the database, while with `assertz(T)` the specific clause is added at the end of the database.
  - During backtracking, `assert` is ignored.
  - For efficiency reasons, functors of predicates that will be added must be declared as dynamic (e.g. `:- dynamic(foo/1).`).
- `retract(T)`, this predicate removes from the program database the first clause which unifies with `T`. In particular:
  - `T` must be instantiated to a term denoting a clause (i.e. a fact or a rule).
  - Some Prolog implementations provide the predicate `abolish`/`retract_all`, which remove all the occurrences of the specified term with the specified arity.

Before this chapter:
Prolog - Definite clause grammars, CLP, CHR

Chapter 5

Probabilistic Logic Programming

Rule-based representation of knowledge is a powerful method. However, one might want to take into consideration uncertainty and probabilistic reasoning. To issue this problem, probabilistic logic programming was introduced. In particular: (open-world assumption is about uncertainty)

- A **probabilistic logic program** (PLP) defines a probability distribution over normal logic programs, called **instances** or **possible worlds**. This distribution is then extended to a joint distribution over worlds and interpretations. The probability of a query is then obtained from this distribution.
- Many languages have been proposed. Among this there is **Logic Programs with Annotated Disjunctions** (LPADs). In LPAD, the head of a clause is extended with disjunctions, and each disjunct is annotated with a probability. For example, one could sample that:
  - If "people have flu", they also "sneeze in 70% of the cases".
  - If "people have hay fever", they also "sneeze in 80% of the cases".
  - Bob has a flu.
  - Bob suffers of hay fever.

  which is translated as follows:

  (do nothing (introduced to obtain a probability distribution))

  ```
  sneezing(X): 0.7; null: 0.3 :- flu(X).
  sneezing(X): 0.8; null: 0.2 :- hay_fever(X).
  flu(Bob).
  hay_fever(Bob).
  ```

  (logic OR)

  where each rule has a probability distribution over its head.
- The possible worlds will then be obtained by selecting one atom from the head of every grounding of each clause. This is done by applying the steps below:
  1. Ground the program.

2. For each atom in each head, choose to include it or not.

For example, the grounded above-mentioned program would be the following:

```
sneezing(Bob): 0.7; null: 0.3 :- flu(Bob).
sneezing(Bob): 0.8; null: 0.2 :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

The possible worlds would be the following:

w_1:
```
sneezing(Bob) :- flu(Bob).
sneezing(Bob) :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

w_2:
```
null :- flu(Bob).
sneezing(Bob) :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

w_3:
```
sneezing(Bob) :- flu(Bob).
null :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

w_4:
```
null :- flu(Bob).
null :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

- Given a clause C and a substitution θ such that $C\theta$ is ground:
  - An **atomic choice** (C, θ, i) is the selection of the i-th atom of the head of C for grounding $C\theta$. (e.g. choosing between 0.7 on null: 0.3)
  - A **composite choice** κ is a consistent set of atomic choices.
  - The **probability of a composite choice** κ is given by:

$$P(\kappa) = \prod_{(C,\theta,i)\in\kappa} P(C, i) \tag{5.1}$$

  - The **selection** σ is a total composite choice (i.e. one atomic choice for every grounding of each clause).
  - A selection σ identifies a logic program w_σ called **world**.
  - The **probability of a world** is given by:

$$P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i)\in\sigma} P(C, i) \tag{5.2}$$

    For example, considering the previous example:

$P(w_1) = 0.7 \cdot 0.8$

```
sneezing(Bob) :- flu(Bob).
sneezing(Bob) :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

$P(w_2) = 0.3 \cdot 0.8$

```
null :- flu(Bob).
sneezing(Bob) :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

$P(w_3) = 0.7 \cdot 0.2$

```
sneezing(Bob) :- flu(Bob).
null :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

$P(w_4) = 0.3 \cdot 0.2$

```
null :- flu(Bob).
null :- hay_fever(Bob).
flu(Bob).
hay_fever(Bob).
```

- Given a ground query Q and a world w:

$$P(Q|w) = \begin{cases} 1 & \text{if } Q \text{ is } \textit{true} \text{ in } w \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

Product rule: $P(A,B) = P(A|B)P(B)$

$$P(Q) = \sum_w P(Q,w) = \sum_w P(Q|w)P(w) = \sum_{w \models Q} P(w) \tag{5.4}$$

For example, considering the previous example, `sneezing(Bob)` is *true* in w_1, w_2 and w_3:

$$P(\texttt{sneezing(Bob)}) = 0.7 \cdot 0.8 + 0.3 \cdot 0.8 + 0.7 \cdot 0.2 = 0.94$$

× Chapter 6

Decision Model and Notation

Usually, taking a decision is a complex and sophisticated process which needs to be shared among managers, customers, developers, needs to be properly documented and possibly executed without ambiguities or further human interventions. A decision can be modelled as an iterative process:

1. Identify the decision.
2. Describe the decision.
3. Specify the decision requirements.
4. Decompose and refine (go back to step 1.).

The Decision Model and Notation (DMN) is a standard published by the Object Management Group (OMG). In particular, it is a standard approach for describing and modelling repeatable decisions within organizations.

Chapter 7

Rule-Based Systems and Forward Reasoning

Rules are a very common way for eliciting knowledge and how decision should be in given and known contexts. Moreover, rules have a solid formal background in logic. In particular:

- The simplest form of a rule is the logical implication:

$$p_1, \ldots, p_i \rightarrow q_1, \ldots, q_j \tag{7.1}$$

  where $p_1, \ldots, p_i$ are the **premises/antecedents/body** of the rule, and $q_1, \ldots, q_j$ are the **consequences/consequent/head** of the rule.

- Reasoning with rules has been systematized in classic Greek philosophy. The default reasoning mechanism is called **Modus Ponens**:

$$\frac{p_1, \ldots, p_i \qquad p_1, \ldots p_i \rightarrow q_1, \ldots, q_j}{q_1, \ldots, q_j} \tag{7.2}$$

  However, this is not the only possible inference mechanism.

7.1 The Production Rules Approach

In Prolog: p:-q ≡ p←q. this is an example of backward chaining

All the previous chapters were about backward chaining and static knowledge. It could happen that, during a certain reasoning step, new information become available. In the production rules approach, new facts can be dynamically added to the knowledge base. If such new facts matches with the left-hand side of any rule, the reasoning mechanism is triggered. In particular, when a new fact is added to the knowledge base, or initially, the following steps are performed:

1. Search for the rules whose left-hand side matches the fact and decide which ones will be triggered.
2. Triggered rules are put into the Agenda and possible conflicts are solve.

list of rules ready to be triggered

3. Right-hand side are executed and effects are performed. Thus, the knowledge base is modified.
4. Go to step 1.

Moreover, production rules systems work with a specific data structure which memorizes the current set of facts (initially it is just a copy of the knowledge base) and the set of rules.

7.2 The RETE Algorithm

This algorithm focuses of the first step of the production rules approach, the matching step. In particular, deciding if the left-hand side os a rule is satisfied/is matched, consists on verifying if all the patters do have a corresponding element (a fact) in the working memory.

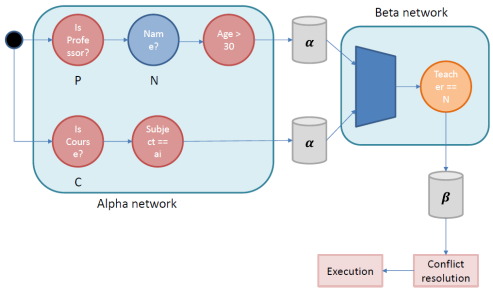
- RETE focuses on determining the so called **conflict set**, i.e. the set of all possible instantiations of production rules such that:

$$\langle \text{Rule (right-hand side), List of elements matched by its left-hand side} \rangle \tag{7.3}$$

- Iteration over facts is avoided by storing, for each pattern, which are the facts that match it. When a fact is added, deleted or modified, the list of matches is updated accordingly.
- Iteration over rules is avoided by compiling a left-hand side into a network of nodes. In particular, there exist two types of patters:
  - Patterns testing intra-elements[1] features. These patterns are compiled into **alpha-networks** and their outcomes are stored into alpha-memories.
  - Patterns testing ~~intra~~inter-elements[2] features. These patterns are compiled into **beta-networks** and their outcomes are stored into beta-memories.

  For example:

  > When a Professor P with name N and age > 30, when a course C with subject "AI" and teacher N, then do something.



  In the end, the beta.memory contains the conflict set, i.e. the list of ⟨Rule, LHS⟩ with left-hand side completely matched and ready to be executed.

The second step of the production rules approach, the one step regarding the resolution of conflicts, is handled by applying some sort of priority. In this step, the RETE algorithm chooses which rule to execute by sorting the selected rules. In the last step, finally, the rules are executed.

7.3 The DROOLS Framework

The DROOLS framework is a rules engine which follows a forward reasoning approach based on the RETE algorithm. Rules have the following structure:

```
rule  "Any string as id of the rule"
      // Possible rule attributes
when
      // LHS: premises or antecendents
then
      // RHS: conclusions or consequents
end
```

In particular:

- The left-hand side is a conjunction (disjunction of) of patterns, where:
  - A pattern is the atomic element for describing a conjunct.
  - Describes a fact.
  - Describes also the conditions about the specific fact.

  For example, "when a person is inserted in the system, send him a greeting email":

  ```
  rule  "Greeting email"
  when
        Person()
  then
        sendEmail("greetings");
  end
  ```

  The specific rule is triggered as soon as a fact `Person()` is inserted in the working memory or at the start of the rule engine, if a fact `Person()` was already in the working memory. Moreover, it is possible to define variables in order to keep in mind certain values/fields/objects. This is achieved as follows:

  ```
  rule  "Greeting email"
  when
        $p: Person($n: name, $n == "Federico", age >= 18)
  then
        sendEmail($p.getEmailAddr(), "greetings to " + $n);
  end
  ```

It is also possible to filter on the basis of three quantifiers:

  - `exists P(...)`: there exists at least a fact `P(...)` in the working memory.
  - `not P(...)`: the working memory does not contain any fact `P(...)`.
  - `forall P(...)`: trigger the rule if all the instance of `P(...)` match.

- The right-hand side, i.e. consequences, can be of two types:

  - Logic consequences, namely the affect the working memory through the insert, retract, modify operations. For example, considering only the insertion case:

e.g. modify ($Log) { setLastLogin (newDate()) }

```
declare  Logged
         person : Person
end

rule     "log in"
when
         $p: Person()
         $e: LogInEvent(person == $p)
then
         Logged ooo = new Logged();
         ooo.setPerson($p);
         insert(ooo);
end

rule     "log out"
when
         $ooo: Logged($p: person)
         $e: LogOutEvent(person == $p)
then
         System.out.println("Goodbye!")
end
```

retract(ooo) / delete(ooo)

    If the new fact matches with the left-hand side of any rule, then the rule will possibly trigger.
  - Non-logic consequences, namely any external side effect which do not affect the working memory.

Lastly, in order to avoid conflicts:

- Rules can have the property `salience`: the higher the value, the higher the priority of their execution:

```
rule  "log in"
      salience 100
```

- Rules can have the property `agenda-group`: a group of rules is selected to be executed from an external method:

```
rule  "log in"
      agenda-group "log"
```

- Rules can have the property `activation-group`: among all the activated rules of the same group only one of them is executed, while all the others are discarded:

```
rule  "log in"
      activation-group "log"
```

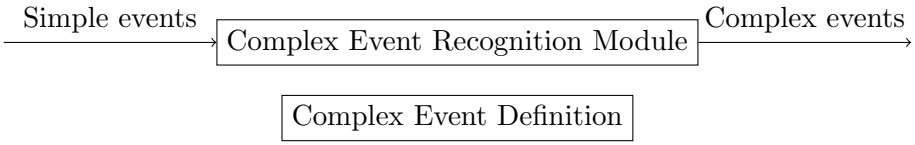
Before this chapter:
DMN - Decision Model and Notation } not asked in the exam

Chapter 8

Complex Event Processing and Event Calculus

Complex Event Processing (CEP) is a paradigm for dealing with a huge amount of information (i.e. Big Data). In particular:

- Collected information is described in terms of **events**. An event is composed of a description of something, in some language, and temporal information about when something happened. These events can be instantaneous or can have a duration.
- This paradigm is about dealing with simple events, reason upon them and derive complex events. As a matter of fact, events can be divided into simple events, which are events whose information payload alone does not provide any meaningful answer, and complex events, which are the events generated by the system itself that provide useful information.



8.1 CEP and DROOLS

DROOLS provide **DROOLS Fusion**, which is a support for doing complex event processing. In particular:

- Events are particular facts.
- Events have a timestamp.

- On the base of the defined rules, DROOLS Fusion automatically decides when it is possible to discard events.
- **Sliding windows** are a way to scope the events of interest by defining a window that is constantly moving. There exist two types of sliding windows:
  - Time-based windows. For example:

```
rule  "Sound the alarm in case temperature rises above threshold"
when
      TemperatureThreshold($max: max)
      Number(doubleValue > $max) from accumulate(
       SensorReading($temp: temperature) over window:time(10m),
       average($temp)
      )
then
      // sound the alarm
end
```

  - Length-based windows. For example:

```
rule  "Sound the alarm in case temperature rises above threshold"
when
      TemperatureThreshold($max: max)
      Number(doubleValue > $max) from accumulate(
       SensorReading($temp: temperature) over window:length(10),
       average($temp)
      )
then
      // sound the alarm
end
```

- Due to the huge amount of events that should be treated, events will be discarded from the working memory as soon as possible.
- A number of Allen temporal operators are supported: `after`, `before`, `coincides`, `during`, `finishes`, `finishedby`, `includes`, `meets`, `metby`, `overlaps`, `overlappedby`, `starts`, `startedby`.

8.2 Event Calculus

CEP is a nice way of reasoning about events, but more complex systems require to reason about the state of a system. In order to do so, it is possible to consider the **Event Calculus Framework**, which is based on points of time and reifies both fluents and events. Examples of the usage of such framework are the following:

- `Holds(F, T)`, which means that the fluent `F` holds at time `T`.

e.g. initially(light_off).
initiates(push_button, light_on, T) <= HoldsAt(light_off, T)

Event Calculus Ontology

- `Happens(E, T)`, which means that the event `E` happened at time `T`.
- `Initiates(E, F, T)`, which means that the event `E` causes the fluent `F` to hold at time `T`.
- `Terminates(E, F, T)`, which means that the event `E` causes the fluent `F` to cease to hold at time `F`.
- `Clipped(T_1, F, T)`, which means that the fluent `F` has been made false between `T_1` and `T`.
- `Initially(F)`, which means that fluent `F` holds at time zero.

Domain-independent axioms

- $\texttt{HoldsAt(F, T)} \Leftarrow \texttt{Happens(E, T\_1)} \wedge \texttt{Initiates(E, F, T\_1)} \wedge \texttt{T\_1 < T} \wedge \neg\texttt{Clipped(T\_1, F, T)}$
- $\texttt{HoldsAt(F, T)} \Leftarrow \texttt{Initially(F)} \wedge \neg\texttt{Clipped(0, F, T)}$
- $\texttt{Clipped(T\_1, F, T\_2)} \Leftarrow \texttt{Happens(E, T)} \wedge \texttt{(T\_1 < T, T\_2)} \wedge \texttt{Terminates(E, F, T)}$

Given a set of events, Event Calculus allows to answer queries very easily. In particular, Event Calculus allows deductive reasoning only and it is easy to implement using Prolog. However, if a new happened event is observed, a new query is needed and so the results are re-computed from scratch.

8.2.1 Reactive Event Calculus

Reactive event calculus overcome the deductive nature of the original formulation of Event Calculus. In particular, new happened events can be added dynamically, i.e. the result is updated and not re-computed from scratch.

Chapter 9

Semantic Web and Knowledge Graphs

9.1 The Web 1.0

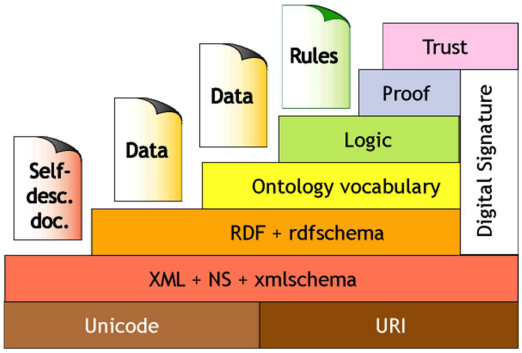
- The information is represented by means of natural language, images, multimedia and graphic rendering/aspect.
- Human users easily exploit all these means for deducting facts from partial information and/or creating mental associations between facts and, e.g., images.
- The content is published on the Web with the principal aim of being human readable. In particular, standard HTML is focused on how to represent the content but there is no notion of what is represented (namely, the semantics).
- The Web is global, i.e. any page can link to anything, anyone can publish anything about any topic.
- Information on the Web are distributed, inconsistent and incomplete.

9.2 Semantic Web

The Semantic Web is an extension of the World Wide Web through standards set by the World Wide Web Consortium (W3C). The goal of the Semantic Web is to make Internet data machine-readable. This is achieved be extending the current Web with knowledge about the content (i.e. semantic information). In particular, the semantic web would like to preserve the following properties:

- Globality.
- Information distribution.
- Information inconsistency.
- Information incompleteness.

* based on triples:
  ⟨subject, predicate, object⟩,
  ⟨resource, attribute, value⟩



The semantic web stack (i.e. the formats and technologies which enables it) is composed as follows: In particular, the semantic web:

- Exploits already available name systems: URIs (Uniform Resource Identifiers). By definition, URI guarantees unicity of names, i.e. to each URI corresponds one and only one concept, but more URI can refer to the same concept.
- It is based on XML (eXtensible Markup Language), which was first created for supporting data exchange between heterogeneous systems. This language is extendible, so that anyone can represent any type of data, is hierarchically structured by means of tags and can contain a description of the grammar used in such document.
- It is based on RDF/RDFS (Resource Description Framework), which is an XML-based language for representing knowledge. In particular, this language refer to objects and their relationships. *
- It uses Ontology Web Language, which is another standard by W3C which extends RDF/RDFS.

knowledge representation

9.3 Knowledge Graphs

Knowledge graphs have been introduced by Google in 2012. The semantic web stack seemed too complex and so fast for Google's needs. In particular, the basic idea behind knowledge graphs is to store the information in terms of nodes and arcs. Knowledge graphs are very large semantic nets that integrate various and heterogeneous information sources to represent knowledge about certain domains of discourse. By implementing this concept, Google transformed from being a search engine to a query-answering engine. Moreover:

- These graphs contains billions of factual knowledge.
- There exists no conceptual schema, i.e. data from various sources, with different semantics can be mixed freely.
- There is no reasoning.
- Graph algorithms are used to fast traverse the graph, looking for the solution.