Logic Programming – Part: 1 General Concepts]

We not describing computation steps we describe problem well enough for the system to infer solution

INTRODUCTION:

- Use *logic* to express knowledge, describe a problem
- Use *inference* to compute, manipulate knowledge, obtain a solution to a problem

ADVANTAGES:

- Knowledge-based programming
- It is a *declarative* style of programming: the program says what should be computed, rather than how it is computed
- Precise and simple semantics
- The same formalism can be used to specify a problem, write a program, prove properties of program
- The same program can be used in many different ways – for example inferring health guide.

The first points are shared with functional languages, but the last point is specific to logic programming languages.

DISADVANTAGES:

- The ability to support **efficient arithmetic and I/O** operations such as file handing is provided at the expense of the **declarative semantics**
- Most logic languages are restricted to a fragment of classical first-order logic. There are some languages based on more powerful logics, but they are not widely used.

PROLOG

Several logic programming languages have been developed. The most popular is *Prolog*.

- Developed in the early 1970s by Colmerauer and Roussel
- Used in natural language processing and artificial intelligence
- Syntax — the clausal fragment of classical first-order logic
- Semantics — SLD-resolution with automatic backtracking
- Impure — includes non-logical primitives

We suggest you use SWI Prolog, an open-source Prolog implementation developed at the University of Amsterdam.
`http://www.swi-prolog.org`

On startup a message will appear, followed by the goal prompt:
`?-`

To load a program from a local file, type:
`?- ['myprogram.pl']`

The software, source code, and reference manual are available online.

DOMAIN OF COMPUTATION: **TERMS**

The set of *terms* is defined using:

- *variables*, represented by X, Y, $Z, \ldots$
- and function symbols, with fixed arities represented by f, g, $h, \ldots$ or a, b, $c, \ldots$ for constants of arity zero.

A *term* is either a variable, or has the form $f(t_1, \ldots, t_n)$, where f is a function symbol of arity n and $t_1, \ldots, t_n$ are terms.

EXAMPLES OF TERMS

- VARIABLE IN CAPITAL LETTERS
- Lowercase is a constant
- Terms are basically all operations that can be applied in the program as mentioned above it can be either a variable or a rule.

If a is a constant, f a binary function, and g a unary function, and X, Y are variables, then the following are possible terms:

- X
- a
- $g(a)$
- $f(X, g(a))$
- Y
- $f(f(X, g(a)), Y)$
- $g(f(f(X, g(a)), Y))$

DOMAIN OF COMPUTATION: **LITERALS**

- Let p, q, $r \ldots$ represent *predicate symbols*, each with a fixed arity.
- If p is a predicate of arity n and $t_1, \ldots, t_n$ are terms, then $p(t_1, \ldots, t_n)$ is an *atomic formula*, A, B, $\ldots$
- A *literal* is an atomic formula A, or a negated atomic formula, $\neg A$.

- Its either P applied to some term or -P applied.

EXAMPLES OF LITERALS

If *rainy* and *snowy* are unary predicates, *temperature* is a binary predicate, *celsius* is a unary function symbol, *tuesday* and *zero* are constants, and X is a variable, then the following are possible literals:

- $temperature(tuesday, celsius(zero))$
- $\neg rainy(tuesday)$
- $snowy(X)$

EXAMPLE

If a is a constant, f a binary function, and g a unary function, and X, Y are variables, which of these are valid terms; if not, why not?

- (X, a) – is invalid because it does not have a function symbol root has to start of function symbol or a variable.
- g(X) – is valid function symbol taking one term.
- g(f) – is invalid you can't have function symbol on its own, it has to have right number of arguments just like constructors in java (unless they are overloaded)
- f(X, f(X, g(f(Y, a)))) – is valid every function symbol has right number of arguments

- notp(X, a) – is valid
- q(g(Y), a) – is invalid because q takes two terms and it is unary by definition.

DOMAIN OF COMPUTATION: **CLAUSES**

HORN CLAUSE: is a disjunction of literals of which at most one may be positive. ONLY ONE POSITIVE

- A Horn clause with one positive literal, $A \vee \neg B_1 \vee \ldots \vee \neg B_n$, is called a *definite clause* and can be read as "A if B_1 and $\ldots$ and B_n".

HORN CLAUSE WITH ONLY ONE POSITIVE LITERAL IS CALLED *fact* WHEN IT HAS ONLY ONE VARIABLE LIKE rainy(Tuesday) OR *rule WITH MORE THAN ONE VARIABLES*

HORN CLAUSE THAT CONTAINS ONLY NEGATIVE LITERALS IS CALLED A *goal* OR *query*

- *Programs are sets of definite clauses.*

TERM – OKRESLENIE ***PREDICATE – TWIERDZENIE***

EXAMPLE

IF *rainy* and *sunny* are unary predicates, *temperature*, is a binary predicate, *Celsius* is a unary function symbol, *Tuesday* and *zero* are constants and *X* is a variable, then the following possible clauses are:

- *rainy(tuesday) – Fact- Horn clause with only one positive literal which is allowed*
- *temperature(tuesday, celsius(zero)) – Rule saying that is raining on day x and temperature was 0.*
- *snowy(X) v ¬raining(X) v ¬temperature(X, celsius(zero)) – Query asking was it not raining and snowing and temperature on that wasn't zero?*
- *¬rainy(X) v ¬snowy(X) – Query asking was it not snowing and not raining ?*
- *p(g(Y), a) – It is a Fact.*
- ¬p(X, a) v ¬q(Y) – It is a Goal
- q(f (Y , a)) ∨ ¬q(X) ∨ ¬p(Y , g(a)) – It is a Rule
- ¬q(f (a, g(Y))) – it is a Goal
- q(a)) ∨ p(b, g(X)) – It is not horn clause as it has more than one positive literal.

SUBSTITUTION

Values are terms, associated to variables by means of automatically generated substitutions.

- A *substitution* is a partial mapping from variables to terms, with a finite domain.
- A substitution σ is written as a mapping $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$.
- $dom(\sigma)$ denotes the domain of the substitution $\{X_1, \ldots, X_n\}$.
- A substitution σ is applied to a term t or a literal L by simultaneously replacing each variable occurring in $dom(\sigma)$ by the corresponding term. The resulting term is denoted $t\sigma$ or $L\sigma$.

SUBSITTUTIONS ARE DONE ALL IN ONE NOT IN SEQUENCE

EXAMPLE

- t = *f(f(X, g(a)), Y)* THIS ARE THE TERMS THAT WE WILL BE SUBSTITUTING
- σ = {*X* → g(Y), Y → a} THIS IS DOMAIN OF ALL VARIABLES WHAT TO SUBSTITUTE
- to = *f(f(g(Y), g(a)), a)* THIS IS MAPPED TERM WITH DOMAIN

When we substitute values, we have to look into domain σ and if I can find a match then we can substitute those In and form new expression tσ.

- $L = \neg p(X, X, f(g(a), Y))$
- $\sigma = \{X \mapsto f(a, b), Y \mapsto Z, Z \mapsto b\}$
- $L\sigma = \neg p(f(a, b), f(a, b), f(g(a), Z))$

In the example on the right side, Y was replaced with value Z, however in sigma there is still a substitution like in this case Z. IT DOES NOT GET REPLACED FROM Z -> B. And this is because the substitution does not happen in a sequence it is done all in one go. Therefore, nothing is evaluated to b

tσ - term substitution σ - sigma lσ - literal substitution

PROLOG SYNTAX: CLAUSES

parent is PREDICATE NAME in this example– parent(Jane, Leszek) – extra not relevant

PROLOG PROGRAM: program is a set of **PREDICATES:** defined as a list of **FACTS** and **RULES**. The order the clauses are defined is critical to the evaluation of the program.

- Variables begin with an upper-case letter or underscore
- Function and predicate symbols begin with a lower-case letter

A rule $A \vee \neg B_1 \vee \ldots \vee \neg B_n$ is written:
`a :- b1, ..., bn.`

A fact A is written:
`a.`

A goal $\neg B_1 \vee \ldots \vee \neg B_n$ is written:
`:- b1, ..., bn.`

FOR SOMETHING TO BE TYPABLE OF WELL TYPED YOU NEED TO BE ABLE TO DRAW DERIVATION TREE

Example

```
based(prolog, logic).
based(haskell, maths).
likes(claire, maths).
likes(max, logic).
likes(X, P) :- based(P, Y), likes(X, Y).
```

This program consists of four facts and a final rule. Sample goals might include:

- `:- likes(claire, haskell).`
- `:- likes(max, P).`
- `:- likes(Z, prolog).`

- This for example can be read as “Claire likes maths” or “Haskell is based on maths”

Prolog syntax: built-in predicates

The programmer can freely choose the names of variables, function symbols and predicate symbols. However, there are some built-in predicates with specific meanings.

- Boolean operators: =, >, <
- Arithmetic operators: + −, ∗, /
- Arithemtic evaluation: `is`
- Tuples: `,`
- I/O: `write`, `nl`

Example

- `+(3, 2)` or `3 + 2`
- `>(X, 2)` or `X > 2`
- `is(X, *(+(3, 2), 2))` or `X is (3 + 2) * 2`
- `,(cat, dog)` or `(cat, dog)`
- `write("error")`

Example

```
hanoi(X) :- move(X, left, right, middle).
move(1, X, Y, _) :- write([X, "->", Y]), nl.
move(N, X, Y, Z) :- M is N - 1,
    move(M, X, Z, Y),
    move(1, X, Y, Z),
    move(M, Z, Y, X).
```

Example

```
rainy(tuesday).
temperature(tuesday, celsius(-4)).
snowy(X) :- rainy(X),
    temperature(X, celsius(Y)), Y <= 0.
```

Example

```
fact(0, 1).
fact(X, N) :- X > 0, Y is X - 1,
    fact(Y, M), N is X * M
```

EXERCISES

Recall the binary predicate `temperature(X, Y)` that expresses the temperature Y on day X, and let `follows(X, Y)` be a binary predicate indicating that day X follows day Y.

- Write a predicate `hot(X)` that decides whether the temperature on a day is over 30 degrees celsius.
- Using `hot`, write a second predicate `heatwave(M, N)` that expresses that it has been been above 30 degrees celsius on consecutive days from M to N.
- Test your answer on the facts:

```
follows(wed, tue).
follows(thu, wed).
follows(fri, thu).
temperature(tue, celsius(32)).
temperature(wed, celsius(31)).
temperature(thu, celsius(34)).
temperature(fri, celsius(29)).
```

- ```
  hot(Day) :-
      temperature(Day, celsius(Temp)), Temp > 30.
  ```
- ```
  heatwave(Day, Day) :- hot(Day).
  heatwave(First, Last) :- hot(First),
      follows(Next, First), heatwave(Next, Last).
  ```

PROLOG: **LISTS**

Like in functional programming, linked lists are a very important data structure and have special syntax.

- The constant `[]` denotes the empty list.
- The built-in predicate | is the "cons" operator that joins an element X to the front of a list L, `[X | L]`
- `[X | [Y | [Z | []]]]` is abbreviated to `[X, Y, Z]`.
- H is called the *head* of the list `[H | T]`
- T is called the *tail* of the list `[H | T]`
- Built-in predicates for common operations: `member/2`, `length/2`, `sort/2` ...

The notation p/*n* indicates that the predicate symbol p has arity *n*.

length takes two argument /2 arity p/n

The predicate `append(S, T, U)` expresses that the result of appending the list T onto the end of list S is the list U.

```
append([], L, L).
append([X | L], Y, [X | Z]) :- append(L, Y, Z).
```

The following goals represent questions to be solved using the definitions given in the program:

```
:- append([0], [1, 2], U)
:- append(X, [1, 2], U)
:- append([1, 2], X, [0])
```

Third case is a failure because by adding [1, 2] we can't end up with only one element in the listke

Write clauses to define predicates `member/2`, `length/2` and `reverse/2`, such that:

- `member(X, L)` decides whether `X` is contained in list `L`.
- `length(L, N)` expresses the length of list `L` in variable `N`.
- `reverse(L, R)` decides whether list `R` is the reverse of list `L`. Hint: write an auxilliary predicate `reverseAux(L, R, A)` which uses a third list argument `A` as an accumulator.

-
```
member(X, [X | _]).
member(X, [Y | T]) :- X \= Y, member(X, T).
```
-
```
length([], 0).
length([_| T], N) :- length(T, M), N is 1 + M.
```
-
```
reverse(L, R) :- reverseAux(L, R, []).
reverseAux([], A, A).
reverseAux([X | L], R, A) :-
    reverseAux(L, R, [X | A]).
```