Functional and **Logic programming**

Tutorial 2: Unification, Lists, Proof trees for recursive predicates.

Unification algorithm

Where do we meet the algorithm?

1) When matching a query against a rule:
   connected(picadilly_circus, bank_street).
?- connected(X,Y).

2) Using the *equals* predicate.
   picadilly_circus = X.

Input: Two terms *A* and *B*.

Output: A minimal set of substitution to make the terms *A* and *B* equal.

Unification algorithm: Examples 1/2

Input: *X* = *picadilly*. Output: **{X = picadilly}**.

Input: *plus(X,Y)* = *plus(Z,4)*
Output: **{X \ 3, Y \ Z}**.

Input: *cons(first,cons(second,nil))* = *cons(A,cons(B,nil))*
Output: **{A \ first, B \ second}**

Unification algorithm: Examples 2/2

Input: ***cons(first,cons(second,nil)) = cons(A,B)***

Output: **{A \ first, B \ cons(second,nil)}**

Input: ***cons(first,nil) = cons(A,cons(B,nil))***
Output: **false**.

Input: ***cons(X,cons(Y,nil)) = cons(Y,cons(element,nil)***
Output: **{X \ element, Y \ element}**

Unification algorithm

1) A variable and a constant do unify.

?- X = piccadily_circus.
true.

2) A variable and a variable do unify.

?- X = Y.
true.

3) Two functions unify if

a) their symbol are equal
b) their arguments unify

Why is it important?

- Working with **lists**.
- Prolog vs. LISP:
  - '.' instead of **cons()**
  - [] instead of **nil**.
- Iterating over a list:
  - **[a,b,c]** = **[X|Y]** then
  - X corresponds to **car**
  - Y corresponds to **cdr**
  - **'.'(a,'.'(b,'.'(c,[]))) = '.'(X,Y)**

Underground journey 1/2

- Let's use the code from the last tutorial to declare predicate **journey/2** which finds a route between two stations and, if successful, writes them down.
- Example:
- **?- journey(bond_street, leicester_square).**
  leicester_square tottenham_court_road oxford_circus

Underground journey 2/2

- Now define predicate **journey/3** which returns a list of visited stations on a journey

- **?- journey(bond_street, leicester_square, LS).**
  LS = [oxford_circus, tottenham_court_road, leicester_square]

Is 'a' member of a list?

- Define your own implementation of the member/2 predicate.
- Example:
- **?- my_member(a, [b, a, c]). yes.**
- **?- my_member(a, []). no.**
- **?- my_member([1,a], [b, a, c, [1,a]]). yes.**

Reverse a list

- Define predicate **addtoend/2** which adds element to the end of given list.
- Example:
  ?- addtoend(0, [1,2,3], L). L = [1,2,3,0].
- Define predicate **my_reverse/2** that reverse elements of given list (you can use the addtoend/2 predicate)
- Example:
  ?- my_reverse([1,2,3], L). L = [3,2,1].

Append two lists

- Define your own implementation of the append/3 predicate
- Example:
  ?- my_append([a,b,c], [d,e], L).
  L = [a,b,c,d,e].
- What would **my_append(L1, L2, [a, b, c])** do?

If time permits

- Define **common/3**, which finds common elements of a two lists.
- Study the astonishing beauty of **sublist/3** predicate: http://www.cs.bris.ac.uk/~flach/SL/labs/labs3.pl