# Lecture Note -- Jan 31st, 2019

1. HW1 released by weekend, due on Feb 19(Tuesday).
2. Review what we discussed last time?

   a. **List** a very basic data structure. (learned two operations on list, islist, member, we are going to learn more).

   b. Thinking in a declarative way.(recursion, formal definition of your predicate signature).

3. Intuition & motivation.

   Why declarative programming/prolog emerged? Or am I wasting my time learning declarative programming?

   Brief answer: No, more and more languages adapted ideas from declarative paradigms. Like Java introduced Stream in the latest version of Java.

   Comprehensive answer: Recall PvsNP from the algorithm class, what is the first NPC Problem? (SAT). And we know the reducibility theory, once we have a solver for a harder NPC problem( named it T), we can reduce any problem (S) to T, then use the solver solve T, at last, converted the T's answer back to S.

   And the SAT problem is given a logic formula, say (p ^ q ^ ~r), and the SAT solver provided the algorithm **control**, we only need to **model** the problem **logic**. **You will see/feel our program is very easy comparing with its imperative counterpart.** Calculation happens when trying to make a statement to be true.

   Let's learn how to solve the problem by **modeling**. We all know that solving a problem contains two parts, **data structure**, and the **algorithm**. Furthermore the **algorithm** contains two parts, **logic** and **control**. Since the solver in declarative programming takes care of the **control** part, before we start writing the **logic**, we need to deal with the **data structure** first.

4. List is everything you need. Really?

   Brief answer: Yes. The languages provide a great support in list, Python, Lisp, Prolog, Lua. More closely related to human thinking(e.g., ToDo list), so it's the keystone in AI.

   Comprehensive answer: Recall the Turning Machine(TM) definition from computation theory class. A single tape turning machine(S) is equivalent to any alternative version of TMs. How to proof equivalent? They can simulate each other. Example one, multi-tape turning machine(M), we can simulate S using M

as well as simulate M using S. Example two, modern CPU, we can simulate CPU using S as well as simulate S using CPU.

How the modern computer stores the data in the memory? The memory is a linear thing can be treated as a huge **List**. You may find out by simulating M/CPU using S is a little bit costly, but it's more generalized, and that's the reason why declarative language is architecture independent. (Concept behind the scenes).

Now, let simulate everything using List! Keep in mind; we can put **anything** as a list element. And think how they will lay in the linear memory. Once we have the representation of the data structure, we can write a program for it.

5. Data Structures in Prolog

   5.1. List -> String, easy we put characters in the list. ['a', 'b', 'c', 'd'].

   5.2. List -> Array, easy we put numbers in the list. [1, 2, 3, 4, 5].

   5.3. List -> Matrix, median we put lists in the list. [ [1, 2], [3, 4] ], [ ['a', 'b'], ['c', 'd'] ].

   5.4. List -> Tree, hard we use a recursive definition. How to represent tree node.

In imperative language

```
typedef struct Node node_st
struct Node {
    void *data;
    node_st *left;
    node_st *right;
};
```

In Prolog

```
t(P, L, R)

Remind: formal definition
P is the data, can be anything. void *data;
L is the left child, the same as t(_, _, _). node_st *left;
R is the right child, the same as t(_, _, _). node_st *right;
```

A quick hint, actually in Prolog we only have a term(a more fundamental element), every list [1, 2, 3] is syntax sugar, ultimately, it will convert to a term with two parameters (like how we represent linked list in C). So we can recursively chain them up.

```
// imperative way to define a list.
typedef struct Node node_st
struct Node {
    void *data;
    node_st *next;
};

% Define a list in Prolog
l(Value, Next)

Value is the data, can be anything. void *data;
Next is the same as l(_, _). node_st *next;
```

So the list [1, 2, 3] will expand to

```
l(1, l(2, l(3, l())))
```

in the Prolog runtime.

Eventually, our tree representation will be

t(a, t(b, nil, nil), t(c, nil, nil))

**Exercise**: write a predicate `istree(T)`, verify whether the T is a valid tree? (Recall the `islist` predicate we wrote last class.)

```
islist([]).
islist([_|T]) :- islist(T).
```

Similarly, think about how the program will traverse on the data structure we described above, the list can be expanded as the follwing term format.

```
islist(l()).
islist(l(_, T)) :- islist(T).
```

then we will create `istree` using the same idea.

**Solution**:

```
istree(nil).
istree(t(P, L, R) :- istree(L), istree(R)).
```

**Question**: is this tree representation enough to represent every tree in our program? What happen if we have a tree with more than 2 children. Do we need to change the tree definition from `t(P, L, R)` to `t(P, L, M, R)` ?

**Answer**: No, we can use left-child right-sibling representation to convert any tree to a binary tree.

5.5 List -> Graph, median we can represent using the formal definition of the graph. `G = {V, E}.`

```
g(v(a,b,c), edges(e(a,b), e(b,c), e(c,a)))
```

or we can represent using all the edges in the graph. Recall the family tree problem.

```
edge(a,b).
edge(b,c).
edge(c,d).
edge(c,e).
edge(c,f).
edge(d,f).
```

**Exercise**: Given the edges representation of the graph as above, write a predicate `connected(X, Y)` , verify whether the two points, X and Y are connected? (Recall the `parent` predicate we wrote last class in the family tree using `father` / `mother` ).

**Solution**:

```
connected(X, Y) :- edge(X, Y).
connected(X, Y) :- edge(X, Z), connected(Z, Y).
```

Note: recall the mistakes we made during the exercise, and why the are wrong.

6.Solving more problems in Prolog.

**Exercise**: Write a predicate `prefix(Xstring, Ystring)` to verify whether Xstring is the prefix of Ystring.

**Solution**:

```
prefix([], Ystring).
prefix([H|Xstring], [H|Ystring]) :- prefix(Xstring, Ystring).
```

**A small magic**:

There is **no distinguish** between **parameter** and **return value** in Prolog program, they are just relations.

So, the Prolog program is **reversible** a Prolog program can work as a checker, when we write the query. `prefix([a], [a, b, c]).` or `prefix([b], [a, b, c]).`

Also, a Prolog program can work as a generator, when we write the query. `prefix(X, [a, b, c]).` or `prefix([a, b], X).`

More exercise for you to practise.

**Exercise**:Write a predicate `suffix(Xstring, Ystring)` to verify whether Xstring is the suffix of Ystring.

**Exercise**:Write a predicate `substring(Xstring, Ystring)` to verify whether Xstring is the substring of Ystring.

**Exercise**:Write a predicate `sorted(X)` to verify whether a list of number X is sorted.

Topics not covered today, and will be covered next time.

Unify `prefix` , `suffix` , `substring` using `append` predicate.

tail-recursive and left-recursive, and also we will discuss how to do the sorting using Prolog.