The algorithm described below uses as input that pair of tree-shaped automata, and uses the following notation:

- A* - prefix-tree automaton
- s - input symbol to be recognized
- A - suffix-tree automaton
- j - number to be attached to states

**Algorithm:** Building of the composed automaton

**Input:** prefix- and suffix-tree automata for a single sample

**Output:** an approximation to the acceptor of the language

Recursive Procedure Propagate Transition (s, σ) → y from tree-shaped automaton T1 to T2:

1. for each transition (s, σ) → ? or (σ, s) → y in T1
   fill state ? with corresponding x or y, getting (x, σ) → y
2. Recursively,
   Propagate transition (s, σ) → y from T2 to T1

**Main Program:**
1. Initialize j = 1 and attach j to the initial state of both A and A*
2. For each transition (s, σ) → y in A* (visited in depth-first order)
   (a) if y is defined - do nothing
   (b) if y is undefined - attach a new state number y = j+1;
   - call Propagate Transition (s, σ) → y from A* and A
3. For each number attached to the states of A* create a corresponding state in A
4. For each transition in A, build a corresponding transition in A* according to the numbers attached to their origin and destination states.

**First example:** The first example infers the syntax of language L1 = a b* c d*

The prefix- and suffix-tree automata will be built from positive sample

S+ = { a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z }

**Figures 2 and 3 illustrate the prefix-tree and the suffix-tree automata built from the positive sample.**

**Second example:** This example is intended to infer regular language L2 = ab (c d)* g* from positive sample

S+ = { a, b, ab, abc, abd, abe, abfg, abgg, abggfg, abggfgg }  

**An example of such a sample is**

S = { a, b, ab, abc, abd, abe, abfg, abgg, abggfg, abggfgg }  

Applying the same algorithm presented before, the resulting automaton is:

**Figures 4 and 5 illustrate the composed automaton**

> Fig. 4 illustrates the composed automaton obtained by the application of the algorithm. This automaton is able to recognize the superset of strings of the language L1.

> Fig. 5 - positive automaton

In this case, the positive automaton recognizes a superset of language L2. A negative sample of representative non-sentences helps changing the automaton to reject the corresponding syntax.

Non that token g precedes symbols c, d, e, or f in all strings of the chosen negative sample.

Submit the string simultaneously to both automata - positive and negative - and accept if it and only if it is accepted by the positive automaton and rejected by the negative automaton.

If speed matters, merge both automata into a single faster equivalent one.

This linear automaton may be easily constructed by applying classical methods [Lew61].

**Conclusions**

It is possible and practical to use adaptive formalisms in order to make inferences, syntactical analysis and other tasks inherent to natural language processing.

The powerful features of adaptive automata - ease to express learning activities and potential high efficiency of the resulting automaton - may be used and explored with this practice, allowing the construction of efficient working devices for natural language processing.

**Figures 5 and 6 illustrate the negative automaton.