An introduction to forest-regular languages

Mika Raento

Basic Research Unit, Helsinki Institute for Information Technology
Department of Computer Science, University of Helsinki
Mika.Raento@cs.Helsinki.FI

1 Motivation

Structured documents have been used since the 80's (SGML), but interest in them has really exploded with the introduction of XML in mid-90's. To enable the use of structured documents we need several techniques:

- Validation
- Querying
- Transformations

These have been originally supported in SGML and XML with somewhat ad-hoc means, with e.g. DTDs. *Forest-regular languages* give a solid theoretical framework for defining these techniques.

- Forest-regular languages are probably the simplest formalisation of forest-languages that is suitable for document processing
- The same way string-regular languages have a natural correspondance with finite-automata, forest-regular languages have a correposdance with forest-automata (we'll see two examples)
- Forest-regular grammars have about the same difficulty of implementation as DTDs, but are more powerful
- The theoretic framework allows 'easily constructed' correctness proofs, complexity analysis, communication, equivalence solutions etc.
- Forest-regular languages are closed under set union, intersection and difference (as e.g. DTDs or context-free languages are not)
- As with string-regular languages, the same formalism and implementation (automata) can be used for recognizing the language (validation), finding patterns (querying) and transformations (with some extra work)

Forest-regular grammars, languages and the correspondant automata are *not* really complex or difficult, but as with most formalisms getting used to the notation and the formalism. This introduction is meant to give you a good intuition to go with the formal notation.

Concretely, with a good understanding of forest-regular languages you can do the following:

- Implement an XML validator and/or query engine, if you need one for a platform where none are available (e.g. Symbian)
- Have an intuitive grasp of the power and applications of Relax NG
- Be able to follow currently active research regarding structured documents

2 Basics

2.1 Trees and Forests

Based on [1, section 4.2].

With the alphabet Σ (ordered) trees t and forests f are defined:

$$t := a\langle f\rangle, a \in \Sigma \quad f := t_1...t_n, n \geq 0$$

Here the brackets $\langle\rangle$ signify nesting in the tree and are not meant as the only possible syntactic representation.

Much of the pre-regular-forests work was done with *ranked* trees, where each symbol a of the alphabet has an associated *arity*, the number of children. There is a mapping of general trees to ranked trees, which is used to transfer results concerning ranked trees to non-ranked trees, so they are equal in a real sense. Ranked trees model really well e.g. nested functions (or logical statements), but are cumbersome for documents, which are naturally unranked.

2.2 Note on terminology

In this introduction we talk about forest languages and forest-regular languages. Although the term 'tree-regular' could be applied to XML and SGML documents (since they only have a single root), we'd need two levels of definitions for our language, since the children of a tree always form a forest. An alternative name that has been used in some literature is *hedge*-languages, since the forests are ordered. We feel that there is no danger of mistake in using the word forest in this text.

(Suomenkieliset termit: puhumme hieman epätäsmällisesti keskenään samanarvoisesti puu- ja metsäkielistä, koska vakintunutta termistöä ei suomeksi ole. 'Hedge'-sanaa vastaavaa sanaa ei oikein suomeksi ole (siis suoraa kasvavista puista/pensaista muodostuvaa aitaa). 'Pensasaita-kielet' on jo termiksi turhan hankala.

2.3 Forest-regular languages

A forest-regular grammar G is a four-tuple

$$G = (X, \Sigma, r_0, R)$$

where X is the set of variables (or non-terminals), Σ the alphabet of terminals, R a (finite) set of rules (productions) of the form $x \to a\langle r\rangle$ where r is a regular expression over X (and $x \in X, a \in \Sigma$), and r_0 is a regular expression over X, the start-expression.

Now the language L_G generated by G can be defined (e.g.) inductively, via the generator $[\![G]\!]$:

- The forest $f = t_1...t_n \in [\![G]\!]r$ iff there is a word $x_1...x_n$ that belongs to the language generated by the regular expression r such that for each $1 \leq i \leq n$, $t_i \in [\![G]\!]x_i$.
- The tree $t = a\langle t_1...t_n\rangle \in [\![G]\!]x$ iff there is a rule $x \to a\langle r\rangle \in R$ and $t_1...t_n \in [\![G]\!]r$.
- The forest $f = t_1...t_n \in L_G$ iff $t_1...t_n \in [\![G]\!]r_0$

(Note that in XML all documents must have a single root. This means that in considering XML, we restrict the start-expression so that it only allows one tree: $|w| = 1 \; \forall \; w$ generated by r_0.)

Properties of forest-regular languages:

- Closed under union (easy to implement)
- Closed under intersection and difference (more expensive, probably worst-case exponential-time)
- Can be used to both recognize and generate words (contrast with rule-based grammars)
- Efficiently and relatively easily implementable

Example forest regular grammar:

- $X =$ { BOOK, TITLE, PART, CHAPTER, P, LINK, FN, PFN }
- $\Sigma =$ { book, title, part, chapter, p, link, footnote }
- $R =$

```
BOOK    ->  book (TITLE, PART+)|(TITLE, CHAPTER+)
TITLE   ->  title
PART    ->  part CHAPTER+
CHAPTER ->  chapter P+
P       ->  p (LINK|FN)*
LINK    ->  link
FN      ->  footnote PFN+
PFN     ->  p LINK*
```

- $r_0 =$ BOOK

Figure 1 shows example trees corresponding to this grammar.

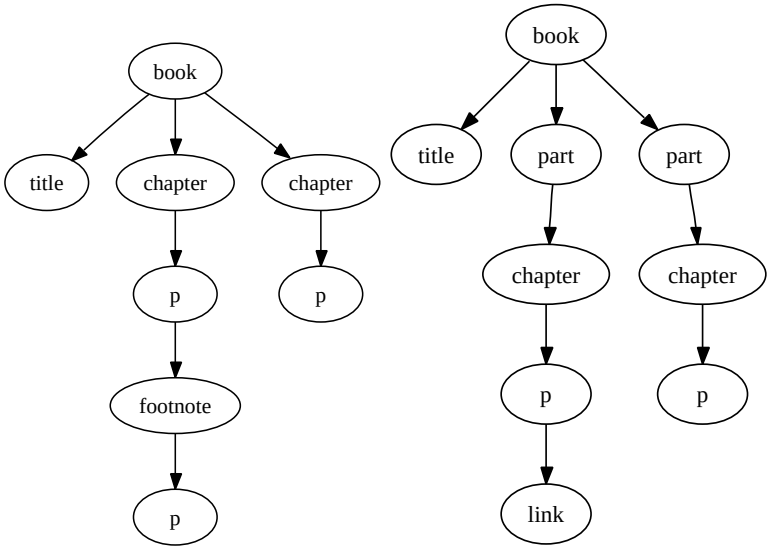

Fig. 1. Example trees corresponding to the grammar in section 2.3

Note that forest-regular grammars can be used to solve the problems that SGML uses inclusions and exclusions for (the P in a FOOTNOTE) with a much cleaner approach, as well as removing the unambiguity constraint of both XML and SGML. Even SGML's &-groups can be done, but the implementation cannot be efficiently done with automata.

2.4 The Berry-Sethi construction

We'll need to be able construct the automaton corresponding to the string-regular expression on the right-hand sides of the production rules of our grammars. A good way to do this is the *Berry-Sethi* construction [1, sec. 4.1.1], which produces a ϵ-free non-deterministic finite automaton (NFA) from a regular expression.

3 Forest automata

In the same way string-regular languages can be recognized by finite automata, forest-regular languages can be recognized by a class of tree-automata. The earliest and simplest formulation is *bottom-up forest automata* [2], a slightly later, more convenient formulation is *pushdown forest automata* [1]. (Note that tree-languages always need some-kind of pushdown — otherwise they would be string-regular. The pushdown automata just makes this more explicit and uses it for more efficient processing.) We shall only describe pushdown automata here. Again we use [1] for notations and definitions.

3.1 Pushdown forest automata

To implement the processing of a bottom-up automata, a pushdown is always needed. A bottom-up automaton cannot take advantage of knowing the structure above a forest, so it has to check the forest agains all right-hand sides of the rules instead of being able to restrict itself to only those rules that can occur in a certain place in the tree.

A (non-deterministic) *left-to-right pushdown automata (LPA)* is a six-tuple $A = (P, Q, I, F, Down, Up, Side)$:

P is the set of *tree states* (corresponding to the names of non-terminals in a grammar, or the rules for them)

Q is the set of *forest-states* (corresponding to the states of the automaton recognizing the right-hand side of the productions)

I , the set of *initial states* (corresponding to the start-states of the automaton recognizing r_0)

Up transition relation $\subseteq Q \times \Sigma \times P$ (in the deterministic case, this would be a function $Up(q, a) \rightarrow P, q \in Q, a \in \Sigma$)

$Side$ transition relation $\subseteq Q \times P \times Q$ (in the deterministic case, this would be a function $Side(q, p) \rightarrow Q$)

$Down$ relation $\subseteq Q \times \Sigma \times Q$, which tells the possible automaton states to begin a new child forest with (in the deterministic case this would be a function $Down(q, a) \rightarrow Q$)

The basic idea is that the *Side* relation recognizes the regular expressions on the right-hand sides of the rules, *Up* labels nodes with rules (or non-terminals), based on the result of the result of traversing the children with *Side* and *Down* selects suitable regular expressions to check.

To construct a pushdown automaton $A = (P, Q, I, F, Down, Up, Side)$ from a forest-regular grammar $G = (X, \Sigma, r_0, R)$, we use the following method:

- Let $R = (x_1 \to a_1 \langle r_1 \rangle) ... x_n \to a_n \langle r_n \rangle)$ be the set of rules
- Let $(Q_j, q_{0,j}, F_j, \delta_j) = BerrySethi(r_j)$ (the non-deterministic finite automata matching the regular expression r_j. A reminder of the automata notation: Q_j are the states, $q_{0,j}$ initial state, F_j final states and δ_j the transition relation)
- $P = X$ (the set of rules)
- $Q = Q_0 ... Q_n$ (the union of all finite automata states of the regular expressions in the rules)
- $I = q_{0,0}$
- $F = F_0$
- $Up = \{(q, a, x) \mid q \in F_j \text{ and } x \to a \langle r_j \rangle\}$ (the node $a \langle f \rangle$ is labeled with x if f matches r_j)
- $Side = \delta_0 ... \delta_n$ (move to the side according to the finite automata matching rs)
- $Down = \{(q_1, a, q_2) \mid q_2 = q_{0,j} \text{ for some } j : x_j \to a \langle r_j \rangle \text{ and } (q_1, x_j, q) \in Side \text{ for some } q\}$ (pick the initial state for the forest under a corresponding to the possible labelings x of a considering that the forest state on the level of a is q_1)

For the grammar in section 2.3 we get from the Berry-Sethi constructions the non-deterministic finite automata (NFA) in figure 2. Now these build the *Side* relation. The top-most ('TOP') automaton representes r_0. Additionally:

- $P =$ { BOOK, TITLE, PART, CHAPTER, P, LINK, FN, PFN }
- Q includes all the states of the automata in figure 2
- $I =$ TOP20
- $F =$ TOP19
- *Down* and *Up* are as described above, e.g. $Down(\text{P13}, \text{footnote}) = \{\text{FN16}\}$ and $Up(\text{BOOK2}, \text{book}) = \{\text{BOOK}\}$.

3.2 Running the LPA

We'll describe the processing model here in a slightly less formal manner: showing how the automaton A is used in a recursive travelsal of a tree.

```
sub process_whole_tree(A, t)
  q = A.I
  p = process_node(A, q, t)
  q = A.Side(q, p)
  if ( q intersects A.F ) return matches
  return does_not_match
end sub

sub process_node(A, q, t)
  q = A.Down(q, t.a)
```

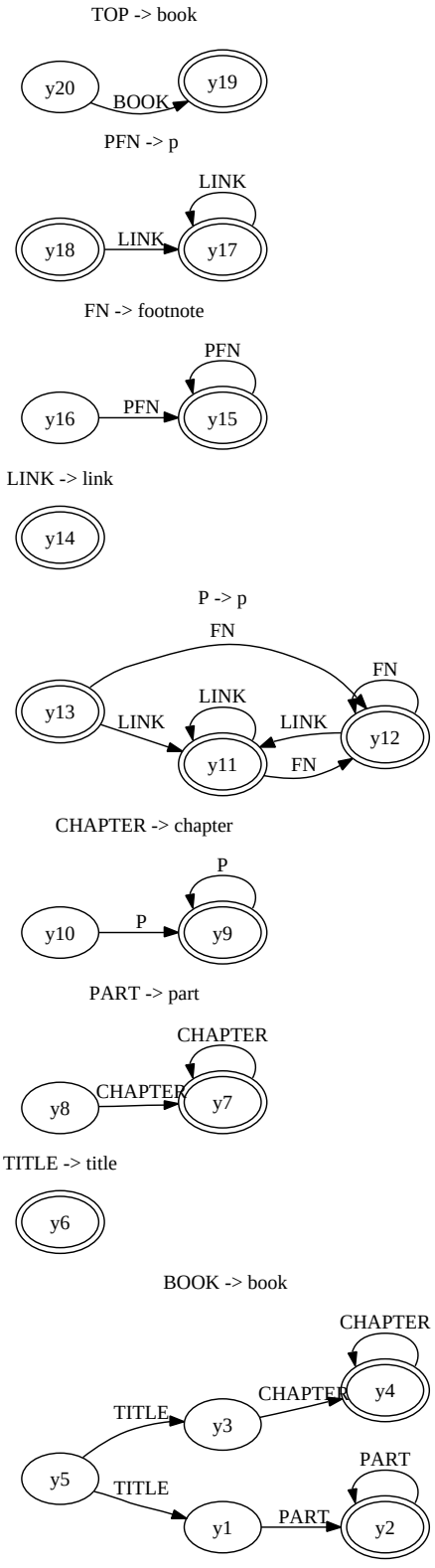

Fig. 2. NFAs from grammar in 2.3

```
  foreach child in t.f
    q = A.Side( q, process_node(A, q, child) )
  next
  p = A.Up(q, t.a)
  return p
end sub
```

Equivalently, we may think of two procedures `begin_element` and `end_element` that are called when seeing the beginning and end of a node in the tree in a streaming (SAX-like) fashion:

```
global p, q, pushdown, A
q = A.I

sub begin_element(a)
  pushdown.push(q)
  q = A.Down(q, a)
end sub

sub end_element(a)
  p = A.Up(q, a)
  q = pushdown.pop()
  q = A.Side(q, p)
end sub
```

And now we can check the global variable q after running the whole tree through and see whether it is in a final state or not.

A more formal description of the transition model of the whole automaton is given in [1, sec. 6.2].

Figure 3 shows a tree which we can label with the forest and tree states of the automaton given in section 3.1.

4 Deterministic automata

The same way finite automata can be made deterministic by a subset construction (the states in the DFA corresponds to (sub)sets of states in the LFA), so can tree-automata be made deterministic in exactly the same way. Of course in the naive setting this means an exponential number of states, as P, Q are replaced by $2^P, 2^Q$ (and correspondingly a larger number of possible transition). The theoretical solution is to remove those states that cannot be reached, and the practical solution is to lazily construct the deterministic automaton based on the non-deterministic one as input is being processed (which incidentally also guarantees that non-reachable states are never constructed).

We shall not provide details of the determinization here. For the purposes of understanding the basics, we can simulate the non-deterministic automaton.

5 Queries

The idea of tree-automata based queries is that we specify the nodes we want to select on the basis of non-terminals in the regular expressions: the query is a subset of these, or equivalently the query a subset of $S \subseteq Q$.

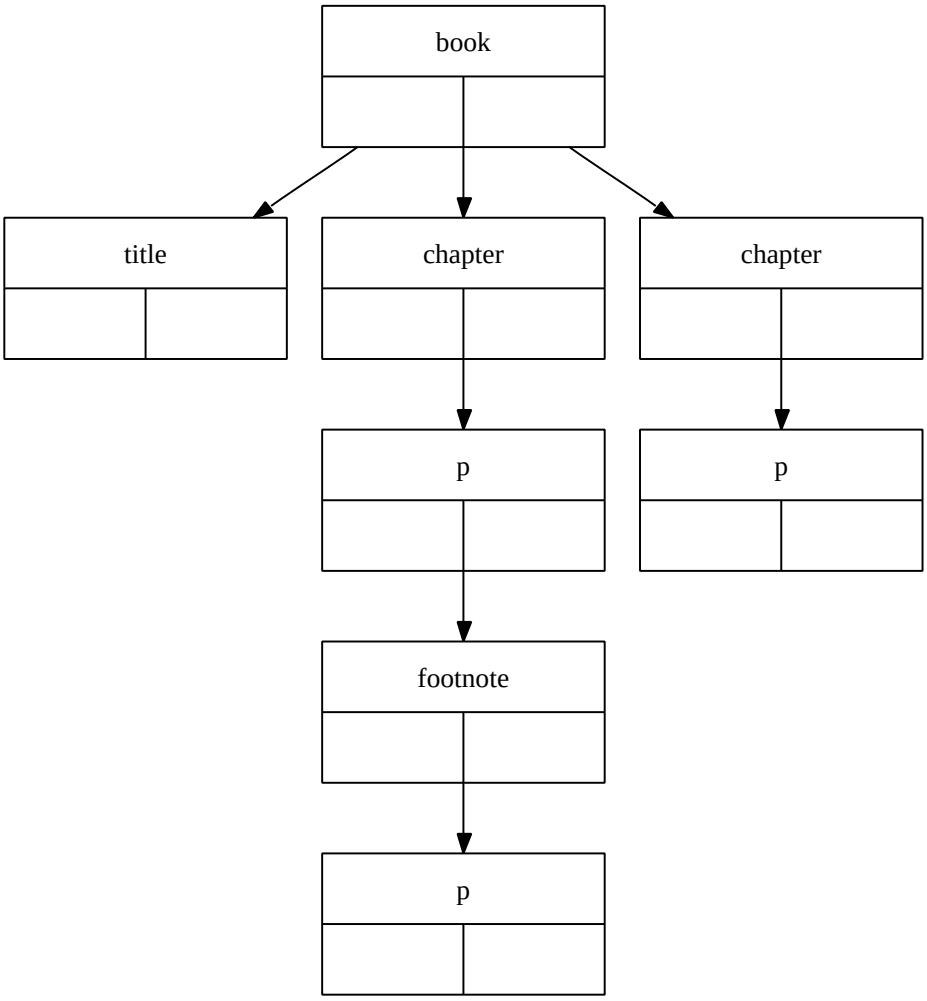

Fig. 3. Fill the slots with a simulated run of the LPA in 3.1

Example query, grammar rules where the queried non-terminals are marked. This query should give as a result the first chapter of a book:

```
BOOK         -> book FRONTMATTER,MAINMATTER,BACKMATTER
FRONTMATTER  -> frontmatter
MAINMATTER   -> mainmatter #CHAPTER,CHAPTER*
BACKMATTER   -> backmatter
CHAPTER      -> chapter XREF*
XREF         -> xref
```

Now if during the run of the left-to-right automaton we arrive at state q a node $a\langle f\rangle$ with $q \in S$, this does not necessarily mean that the node actually matches the query, as only its structure and the nodes above and to the left of it have been seen, and the rest might not match the grammar, or might prove that it did not match the set corresponding to q.

To actually match a query, we must go through the tree twice: once from left-to-right and once from right-to-left. We label the tree nodes with the sets of tree-states (non-terminal names) and possible forest-states during the left-to-right run, run right-to-left and remove those labels that prove impossible. Now if a node still has a label $q \in S$ after the right-to-left run, it actually matches our query.

For details, see [3].

References

1. Neumann, A.: Parsing and Querying XML Documents in SML. PhD thesis (2000)
2. Murata, M.: Forest-regular languages and tree-regular languages. Working paper (1995) http://www.geocities.com/ResearchTriangle/Lab/6259/prelim1.pdf.
3. Berlea, A., Seidl, H.: Binary queries for document trees. Nordic Journal of Computing (NJC) 11:41–71 (2004)