# Web Data Models

## Typing: DTD, Schema

**Silviu Maniu**

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir

université PARIS-SACLAY

# XML Type Definition Language

- XML type definition language: a way to specify a certain subset of XML document — a type
- specification should be simple: a validator should be built automatically and efficiently

# DTD: Syntax

- `<!ELEMENT elem_name elem_regexp>` — an element named `elem_name` contains elements described by the regular expression `elem_regexp`

- `<!ATTLIST elem_name att_name att_type att_values>` — the element `elem_name` has an attribute named `att_name` of type `att_type` and having possible values described by `att_values`

# DTD: Syntax

- regular expressions are formed of `*`,`+`,`?`, sequence `[,]`, `EMPTY`, `ANY`, `#PCDATA` (text)
- attribute types are `ID` (primary key), `IDREF` (foreign key), `CDATA` (text), `v1|v2|,…,vn` (fixed value list)
- attribute values are `v` (default value), `#REQUIRED` (mandatory attribute), `#IMPLIED` (optional attribute), `#FIXED v` (constant value `v`)

# DTD

```
<!ELEMENT books (book*)>
<!ELEMENT book (publisher,edition, authors)>
<!ATTLIST book title CDATA #REQUIRED>
<!ELEMENT publisher #PCDATA>
<!ELEMENT edition #PCDATA>
<!ELEMENT authors (author+)>
<!ELEMENT author (first,last)>
<!ELEMENT first #PCDATA>
<!ELEMENT last #PCDATA>
```

# Mixed Content

- Mixed contend described by a repeatable OR group (between |):

```
(#PCDATA | element-name | … )
```

- `#PCDATA` must be first followed by 0 or more elements — can be repeated multiple times

# DTD: Regular Expressions

- most interesting part of DTD — matching regular expressions on the contents

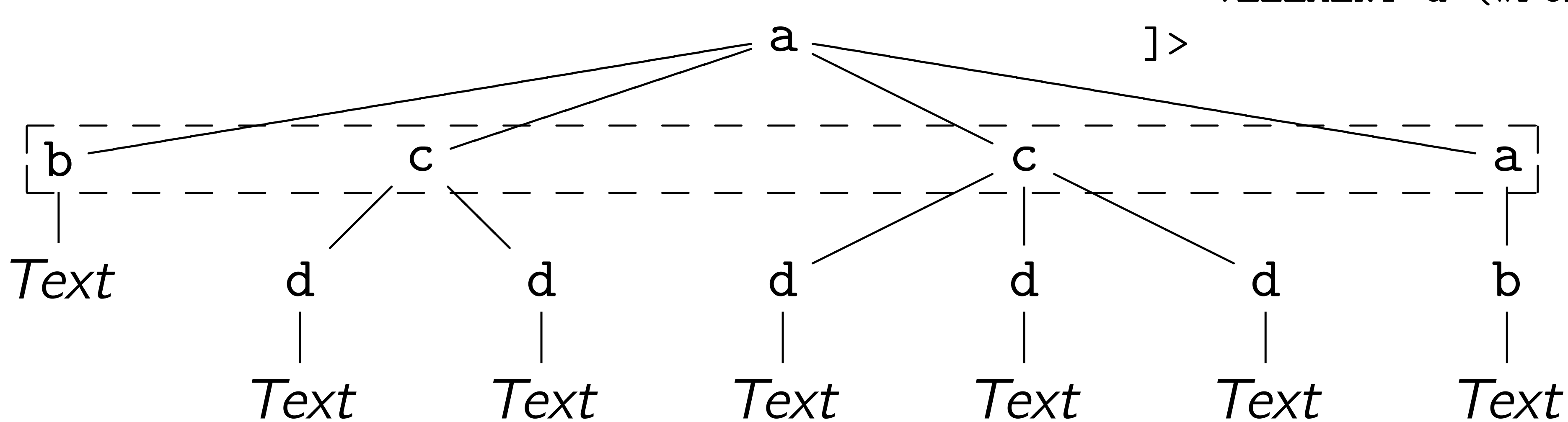
```
<!ELEMENT person

 (name, title?, address*, (fax|tel)*,
 email*) >
```

# DTD: Regular Expressions

- The sequence of children labels has to match its regular expression content model:

```
<!DOCTYPE a [
  <!ELEMENT a (b, c*, a?)>
  <!ELEMENT b (#PCDATA)  >
  <!ELEMENT c (d, d+)    >
  <!ELEMENT d (#PCDATA)  >
]>
```



# Questions to Answer

1. What is a regular expression? How can we match a string against it?

2. What is a finite-state automaton?

3. What is a deterministic regular expression?

4. What is an 1-unambiguous regular expression?

# Regular Expressions

| | meaning |
|---|---|
| **a** | tag/element *a* occurs |
| **e1, e2** | expression *e1* is followed by expression *e2* |
| **e*** | 0 or more occurrences of *e* |
| **e?** | optional — 0 or 1 occurrences of *e* |
| **e+** | 1 or more occurrences of *e* |
| **e1 \| e2** | *e1* or *e2* |
| **(e)** | grouping |

# Regular Expressions

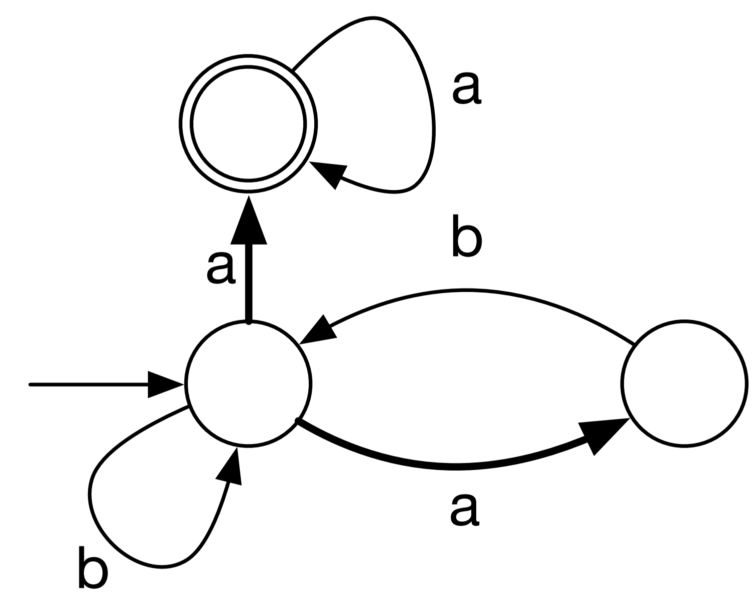
- very useful for defining programming language syntax
- in various Unix tools (`grep`), text editors (`vim`, `emacs`, …)
- classical concept in CS (starting from Kleene, 50s)

# Implementing REs

- **input**: RE e, string s; **output**: does s match e?
- construct a **non-deterministic** or deterministic finite-state automaton (FA)

e = (ab|b)*a*a

s = abbaaba

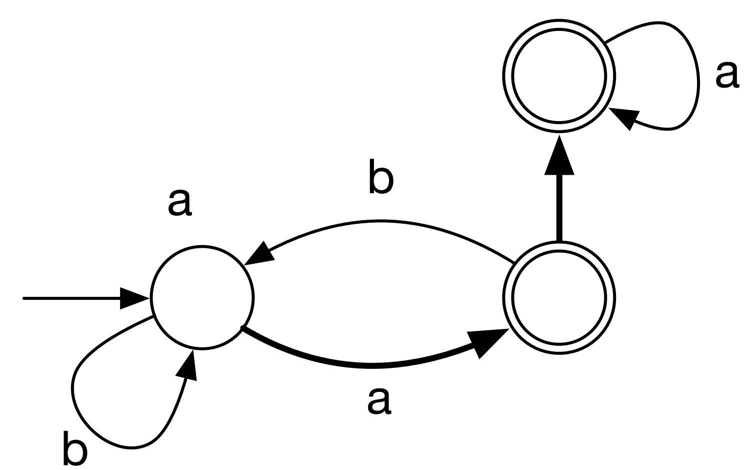


# Implementing REs

- **input**: RE e, string s; **output**: does s match e?
- construct a non-deterministic or **deterministic** automaton

e = (ab|b)*a*a

s = abbaaba



# Implementing REs

- evaluation on a deterministic FA can be done in linear time (in the size of the string $|s|$) and in constant space (size of the FA = number of states) - **how?**

# Implementing REs

- a non-deterministic FA can be transformed to a deterministic FA — but in exponential space; meaning that evaluation is not efficient
- for a deterministic FA one can build a minimal unique equivalent FA — equivalence between FAs is easy to check

# DTDs and REs

W3C requires that the RE specified in DTD must be deterministic:

- evaluation is efficient if element-type definitions are deterministic
- resulting automaton = Glushkov automaton
- states = positions of the regular expression (semantic actions); transitions = based on the "follows set"

# DTDs and REs

- **XML specification**: regular expressions are deterministic (1-unambiguos)

- unambiguous = *each word (string) is witnessed by at most one sequence of positions of symbols in the expression that matches the word* [**Brügemann-Klein, Wood 1998**]

  ambiguous: (a|b)*aa*

  equivalent unambiguous: (a|b)*a

# DTDs and REs

- Is it enough for expressions to be only unambiguous?

- **No** = an expression can be unambiguous but the matching decision has to be done by looking at more states in advance

(a|b)*a

- without looking beyond the current symbol = 1-unambiguous

# Glushkov Automaton

**Can we recognize deterministic FAs?** [Brügemann-Klein, Wood 1998]

- a regular expression is deterministic iff its Glushkov automaton is deterministic
- the Glushkov automaton can be computed in time quadratic in the size of the regular expression

# Glushkov Automaton

- character in RE = state in an automaton + one state of the beginning of the RE
- transitions show which characters can precede each other; incoming labels can only be the labels of the state
- construction is quadratic time $O(m^2)$

# Glushkov Automaton

- What is the Glushkov automaton for:

$$a(b|c)(b|d)^*$$

# DTD: Validation Using FA

General algorithm for DTD (top-down):

1. for each `<!ELEMENT…` create its deterministic automaton A

2. for each element in document *D*, match the children using its corresponding automaton

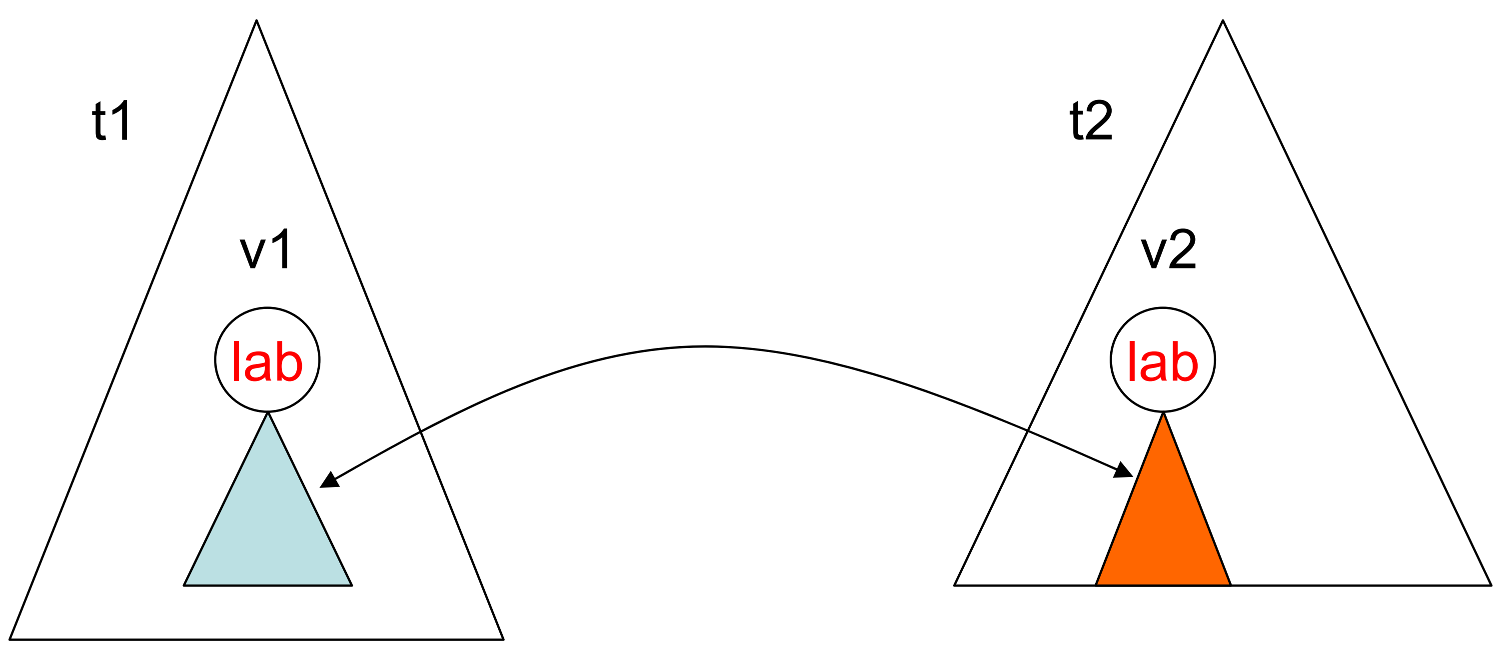
3. if one does not match = document invalid

4. if all match = document valid

# DTD: Validation Using FA

Why does this work?

- label-guarded subtree exchange property = trees obtained by exchanging the subtrees rooted at v1 and v2 are in the same languages if v1 and v2 have the same label lab



# DTD Validation: Example

```xml
<a>
  <a>
    <a />
  </a>
  <b>
    <e />
    <f />
    <g />
  </b>
  <c>
    <e />
    <e />
    <e />
    <d />
  </c>
  <b>
    <e />
    <g />
  </b>
  <b>
    <e />
    <f />
  </b>
</a>
```

```
<!ELEMENT a (a,(b|c)*)>
<!ELEMENT b (e, f?, g?)>
<!ELEMENT c (e+, d)>
<!ELEMENT d EMPTY>
<!ELEMENT e EMPTY>
<!ELEMENT f EMPTY>
<!ELEMENT g EMPTY>
```

# DTD: Limits

- DTD is compact, easy to understand, easy to validate (with the W3C restrictions…)

- But:

1. it is not in XML (dealing with another language)

2. no distinguishable types (everything is characters)

3. no value constraints (cardinality of sequences)

4. no built-in scoping (elements only used in subtrees)

# XML Schema

# XML Schema

- **W3C Standard** — schema description language that goes beyond the capabilities of the DTD
- XML Schema specifications are XML documents themselves
- XML Schema has built-in data types (based on Java and SQL types)
- control over the values a data type can assume
- users can define their own data types

# XML Schema Constructs

- declaring an element (by default, can only contain string values)

```
<xsd:element name="author" />
```

- bounded occurrences (absence of minOccurs / maxOccurs implies once)

```
<xsd:element name="address" minOccurs="1"
maxOccurs="unbounded" />
```

- types (considered atomic with respect to the schema)

```
<xsd:element name="year" type="xsd:date" />
```

other types: string, boolean, number, float, duration, time, base64binary, AnyURI, …

# XML Schema Constructs

- non atomic complex types are built from simple types using type constructors

```
<xsd:complexType name=“Persons">
 <xsd:sequence>
  <xsd:element name=“person" minOccurs=“0"
  maxOccurs=“unbounded”/>
 </xsd:sequence>
</xsd:complexType>
<xsd:element name=“persons" type=“Persons" />
```

# XML Schema Constructs

- new complex types can be derived from an existing type (see specification)

- attributes are declared within the element

```
<xsd:element name=“book”>

 <xsd:attribute name=“title" />

 <xsd:attribute name=“year" type=“xsd:gYear”/
 >

</xsd:element>
```

# XML Schema Example

- What is the schema of this XML?

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book id="1" title="Theory of Computation">
    <authors>
      <author>Michael Sipser</author>
    </author>
    <publisher>Cengage Learning</publisher>
    <year>2012</year>
    <edition>3</edition>
  </book>
  <book id="2" title="Artificial Intelligence">
    <authors>
      <author>Peter Norvig</author>
      <author>Stuart Russell</author>
    </authors>
    <publisher>Pearson</publisher>
    <year>2013</year>
    <edition>3</edition>
  </book>
</books>
```

# Tree Automata for XML Validation

# Validating XML In General

- FA on strings (words) are very good and very efficient for DTDs (and, as we will see, for XPath)

- But what about XML schema? Or any other schema language?

- Is there a formalism / structure that can validate XML in general?

# Tree Automata

Two types:

1. on ranked trees: each node has a bounded number of children; each XML can be transformed by using the first child - next sibling encoding (more later)

2. on unranked trees: no bound on the number of children; better suited (directly) to XML,

# Binary Tree Automata

## Bottom-up non-deterministic tree automata

A non-deterministic bottom-up tree automata is a 4-tuple $\mathcal{A} = (\Sigma, Q, F, \Delta)$ where

- $\Sigma$ is an alphabet. We usually distinguish between two disjoint alphabets : a leaf alphabet ($\Sigma_{leaf}$) and an internal one ($\Sigma_{internal}$).
- $Q$ is a set of states.
- $F$ is a set of accepting states $F \subseteq Q$.
- $\Delta$ is a set of transition rules having one of the forms :

$$l \rightarrow q \text{ when } l \in \Sigma_{leaf}$$
$$a(q_1, q_2) \rightarrow q \text{ when } a \in \Sigma_{internal}$$

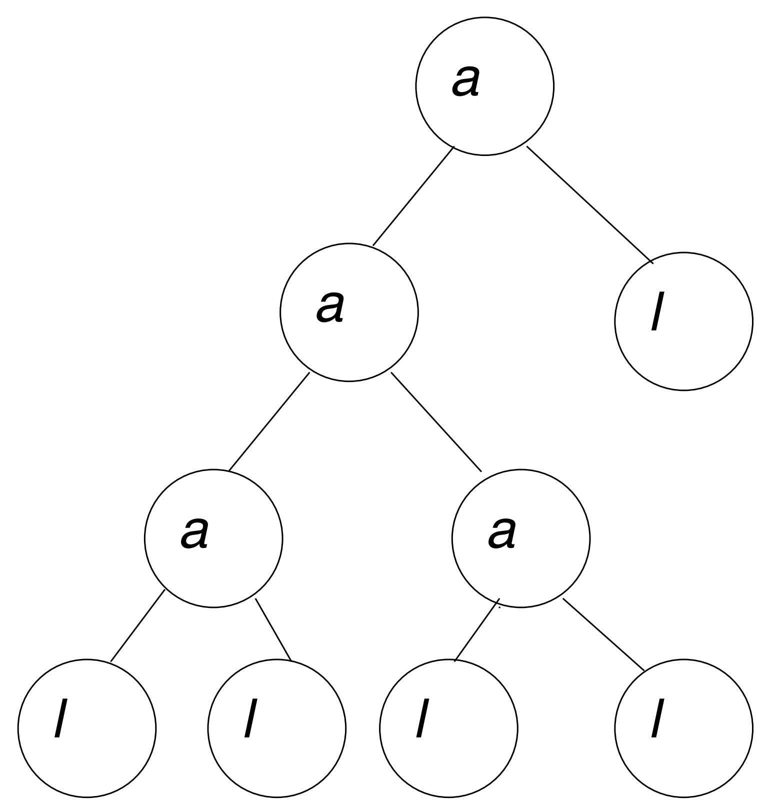
# Binary Tree Automata: Semantics

- the semantics of automata A are described in terms of a **run**
- a run = a mapping from the domain of Q (states) such that for each p we have r(p) in Q
- a run is accepting if the state of the root is one of the final states

# Automata Example

Let $\mathcal{A} = (\{a, l\}, \{q_0, q_1\}, \{q_0\}, \Delta)$ where

$$\Delta = \begin{cases} a(q_1, q_1) & \rightarrow q_0 \\ a(q_0, q_0) & \rightarrow q_1 \\ l & \rightarrow q_1 \end{cases}$$



# Tree Languages

- The language $\mathbf{L(A)}$ is the set of trees accepted by $\mathbf{A}$
- A language accepted by a bottom-up tree automaton is called a regular tree language

# Top-Down Tree Automata

**Binary top-down tree automata**

A non-deterministic top-down tree automata is a 5-tuple $\mathcal{A} = (\Sigma, Q, I, F, \Delta)$ where

- $\Sigma$ is an alphabet.
- $Q$ is a set of states.
- $I \subseteq Q$ is a set of initial states.
- $F$ is a set of accepting states $F \subseteq Q$.
- $\Delta$ is a set of transition rules having the form :
$$q \rightarrow a(q_1, q_2).$$
where $a \in \Sigma$ and $q, q_1, q_2 \in Q$

# Top-Down Tree Automata: Semantics

## Run

- A run of top-down automaton $\mathcal{A} = (\Sigma, Q, I, F, \Delta)$ on a binary tree $t$ is a mapping $r : dom(t) \rightarrow Q$ such that
  - $r(\epsilon) \in I$;
  - for each node $p$ with label $a$, rule $r(p) \rightarrow a(r(p.0), r(p.1))$ is in $\Delta$.
- A run is accepting if for all leaves $p$ we have $r(p) \in F$.

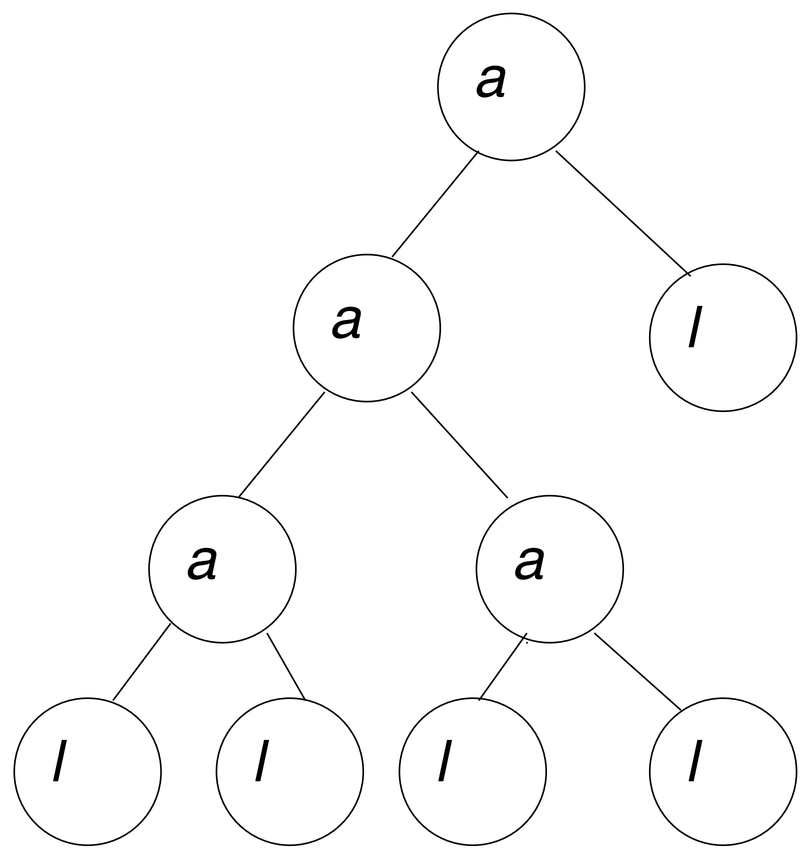
## Deterministic binary top-down automata

We say that a binary tree automaton is (top-down) deterministic **if I is a singleton and for each $a \in \Sigma$ and $q \in Q$ there is *at most* one transition rule of the form** $q \rightarrow a(q_1, q_2)$.

# Automata Example

Let $\mathcal{A} = (\{a, l\}, \{q_0, q_1\}, \{q_0\}, \{q_1\}, \Delta)$ where

$$\Delta = \begin{cases} q_0 & \rightarrow a(q_1, q_1) \\ q_1 & \rightarrow a(q_0, q_0) \end{cases}$$



# Regular Tree Languages

The following statements are equivalent:

- L is a regular tree language
- L is accepted by a non-deterministic bottom-up tree automaton
- L is accepted by a deterministic bottom-up automaton
- L is accepted by a non-deterministic top-down automaton

# Regular Tree Languages

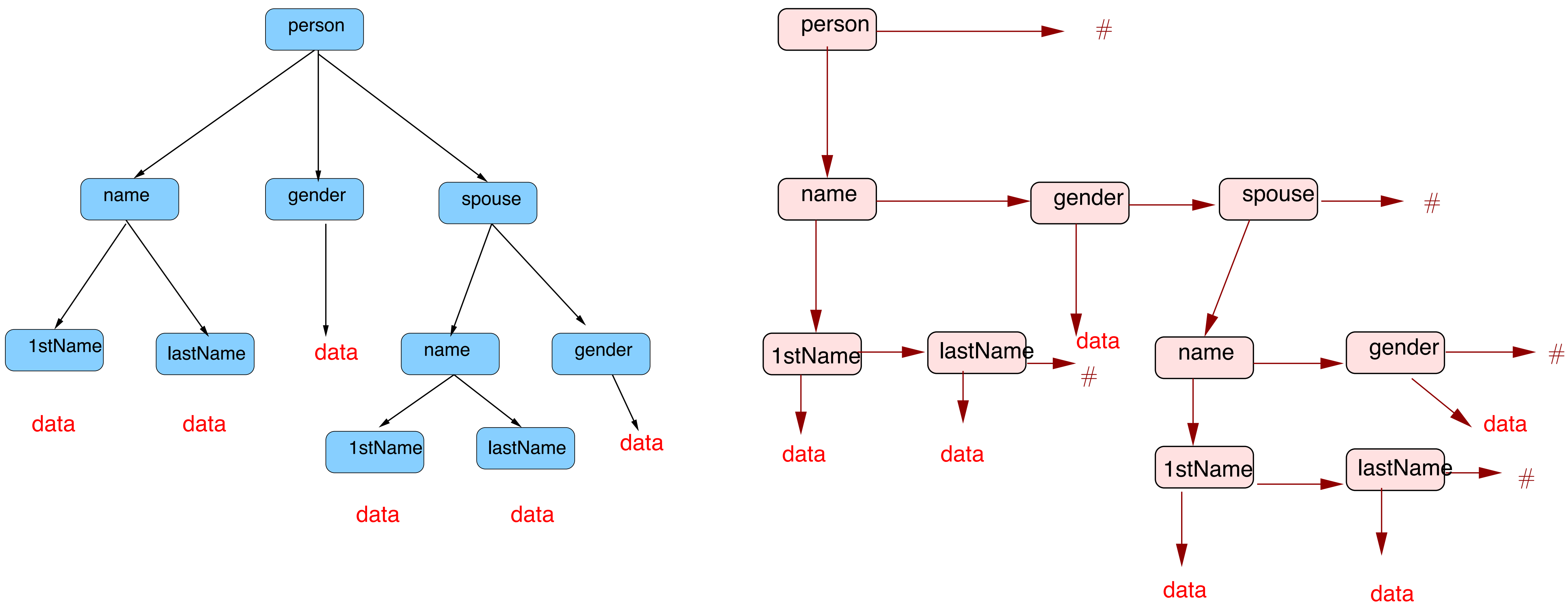
Generally, the same results as for regular word/string languages (FA):

- given a tree automaton, one can find an equivalent bottom-up automaton that is deterministic (with exponential blowup)
- regular tree languages are closed under complement, intersection and union

# Ranked Tree Automata

- We can represent any unranked tree (XML) by a binary tree where the left child is the first child and the right child is the next sibling
- Called first-child next-sibling encoding (not the only one)

# XML — Ranked Tree



# Relation Between Ranked and Unranked Tree Automata

- For each unranked tree automaton, there exists a ranked tree automaton accepting the encoding of the XML in first child — next sibling
- For each ranked tree automaton, there exists an unranked tree automaton accepting the unranked tree seconded from first child — next sibling encoding

# Unranked Bottom-Up Tree Automata

**Non-deterministic bottom-up tree automata**

A non-deterministic bottom-up tree automaton is a 4-tuple $\mathcal{A} = (\Sigma, Q, F, \Delta)$ where $\Sigma$ is an alphabet, $Q$ is a set of states, $F \subseteq Q$ is a set of final states an $\Delta$ is a set of transition rules of the form

$$a[E] \to q$$

where $a \in \Sigma$, $E$ is a regular expression over $Q$ and $q \in Q$

# Unranked Bottom-Up Tree Automata: Semantics

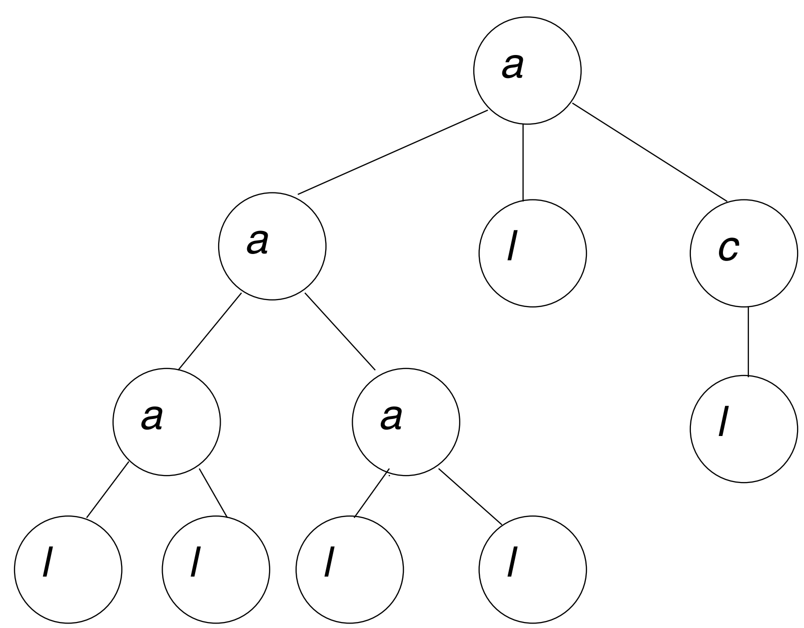
Let $\mathcal{A} = (\Sigma, Q, F, \Delta)$ be an unranked tree automata.

- The semantics of $\mathcal{A}$ is described in terms of runs
- Given an **unranked tree** $t$, a *run* of $\mathcal{A}$ on $t$ is a mapping from $dom(t)$ to $Q$ where, for each position $p$ whose children are at positions $p0, \ldots, p(n-1)$ (with $n \geq 0$), we have $r(p) = q$ if all the following conditions hold :
  - $t(p) = a \in \Sigma$,
  - the mapping $r$ is already defined for the children of $p$, *i.e.*, $r(p.0) = q_0, \ldots, r(p.(n-1)) = q_{n-1}$ and
  - the word $q_0.q_1 \ldots q_{n-1}$ is in $L(E)$.
- A run $r$ is *successful* if $r(\epsilon)$ is a final state.

# Automata Example

Let $\mathcal{A} = (\{a, l\}, \{q_a, q_c, q_l\}, \{q_a\}, \Delta)$ where

$$\Delta = \begin{cases} a\ [q_a^*.q_l^*.(q_c \mid \epsilon)] & \rightarrow q_a \\ c\ [q_l] & \rightarrow q_c \\ l\ [\epsilon] & \rightarrow q_l \quad \text{Special rule for leaves} \end{cases}$$



# Schemas and Tree Grammars

- Schemas for XML documents can be formally expressed by Regular Tree Grammars (RTG)

### Regular Tree Grammar (RTG)

A *regular tree grammar* (RTG) is a 4-tuple $G = (N, T, S, P)$, where :

- $N$ is a finite set of *non-terminal symbols* ;
- $T$ is a finite set of *terminal symbols* ;
- $S$ is a set of *start symbols*, where $S \subseteq N$ and
- $P$ is a finite set of *production rules* of the form $X \rightarrow a\,[R]$, where $X \in N$, $a \in T$, and $R$ is a regular expression over $N$.

(We say that, for a production rule, $X$ is the left-hand side, $a\,R$ is the right-hand side, and $R$ is the content model.)

# Grammar Example

| $P_1$ |
| --- |
| $\mathit{Dir} \to \mathit{directory}[\mathit{Person}^*]$ |
| $\mathit{Person} \to \mathit{student}[\mathit{DirA} \mid \mathit{DirB}])$ |
| $\mathit{Person} \to \mathit{professor}[\mathit{DirB}]$ |
| $\mathit{DirA} \to \mathit{direction}[\mathit{Name}.\mathit{Number}?.\mathit{Add}?]$ |
| $\mathit{DirB} \to \mathit{direction}[\mathit{Name}.\mathit{Add}?.\mathit{Phone}^*]$ |

# Competing Non-Terminals

Two different non-terminals A and B (of the same grammar G) are said to be competing with each other if:

- a production rule has A in the left-hand side,
- another production rule has B in the left-hand side, and
- these two production rules share the same terminal symbol in the right-hand side.

  Same definition for automata — states are competing if they have the same label and different transition rules

# Grammar Example

| $P_1$ | $\Delta_1$ |
|---|---|
| $Dir \to directory[Person^*]$ | $directory[q^*_{person}] \to q_{dir}$ |
| $Person \to student[DirA \mid DirB])$ | $student[q_{dirA} \mid q_{dirB}] \to q_{person}$ |
| $Person \to professor[DirB]$ | $professor[q_{dirB}] \to q_{person}$ |
| $DirA \to direction[Name.Number?.Add?]$ | $direction[q_{name}.q_{number}?.q_{add}?] \to q_{dirA}$ |
| $DirB \to direction[Name.Add?.Phone^*]$ | $direction[q_{name}.q_{add}?.q^*_{phone}] \to q_{dirB}$ |

# Local Tree Grammar

- A local tree grammar (LTG) is a regular tree grammar that does not have competing non-terminals
- A local tree language (LTL) is a language that can be generated by at least one LTG

# Grammar Example

| $P_3$ | $\Delta_3$ |
|---|---|
| $Dir \rightarrow directory[Student^*.Professor^*]$ | $directory[q_{stud}^*.q_{prof}^*] \rightarrow q_{dir}$ |
| $Student \rightarrow student[Name.Number?.Add?]$ | $student[q_{name}.q_{number}?.q_{add}?] \rightarrow q_{stud}$ |
| $Professor \rightarrow professor[Name.Add?.Phone^*]$ | $professor[q_{name}.q_{add}?.q_{phone}^*] \rightarrow q_{prof}$ |

# Single-Type Tree Grammar

A single type tree grammar (STTG) is a regular tree grammar, where:

- for each production rule, non terminals in its regular expression do not compete with each other, and
- start symbols do not compete with each other.

A single-type tree language (STTL) is a language that an be generated by at least one STTG.

# Grammar Example

| $P_2$ | $\Delta_2$ |
|---|---|
| $Dir \rightarrow directory[Person^*]$ | $directory[q^*_{person}] \rightarrow q_{dir}$ |
| $Person \rightarrow student[DirA])$ | $student[q_{dirA}] \rightarrow q_{person}$ |
| $Person \rightarrow professor[DirB]$ | $professor[q_{dirB}] \rightarrow q_{person}$ |
| $DirA \rightarrow direction[Name.Number?.Add?]$ | $direction[q_{name}.q_{number}?.q_{add}?] \rightarrow q_{dirA}$ |
| $DirB \rightarrow direction[Name.Add?.Phone^*]$ | $direction[q_{name}.q_{add}?.q^*_{phone}] \rightarrow q_{dirB}$ |

# Classes of Regular Languages

## LTL $\subset$ STTL $\subset$ RTL

- LTL and STTL are closed under intersection but not union; RTL closed under union, intersection and difference

# XML Schema Languages

| Grammar | Schema Language |
| --- | --- |
| LTG | DTD |
| STTG | XML Schema |
| RTG | RelaxNG |

# General Validation Algorithm

A run associates to each position **p** in the XML document a set of states in **Q** such that:

1. there exists a transition rule to a state in **Q** from a label **a**

2. the label at **p** is **a**

3. the string of the children labels matches the RE in the transition rule

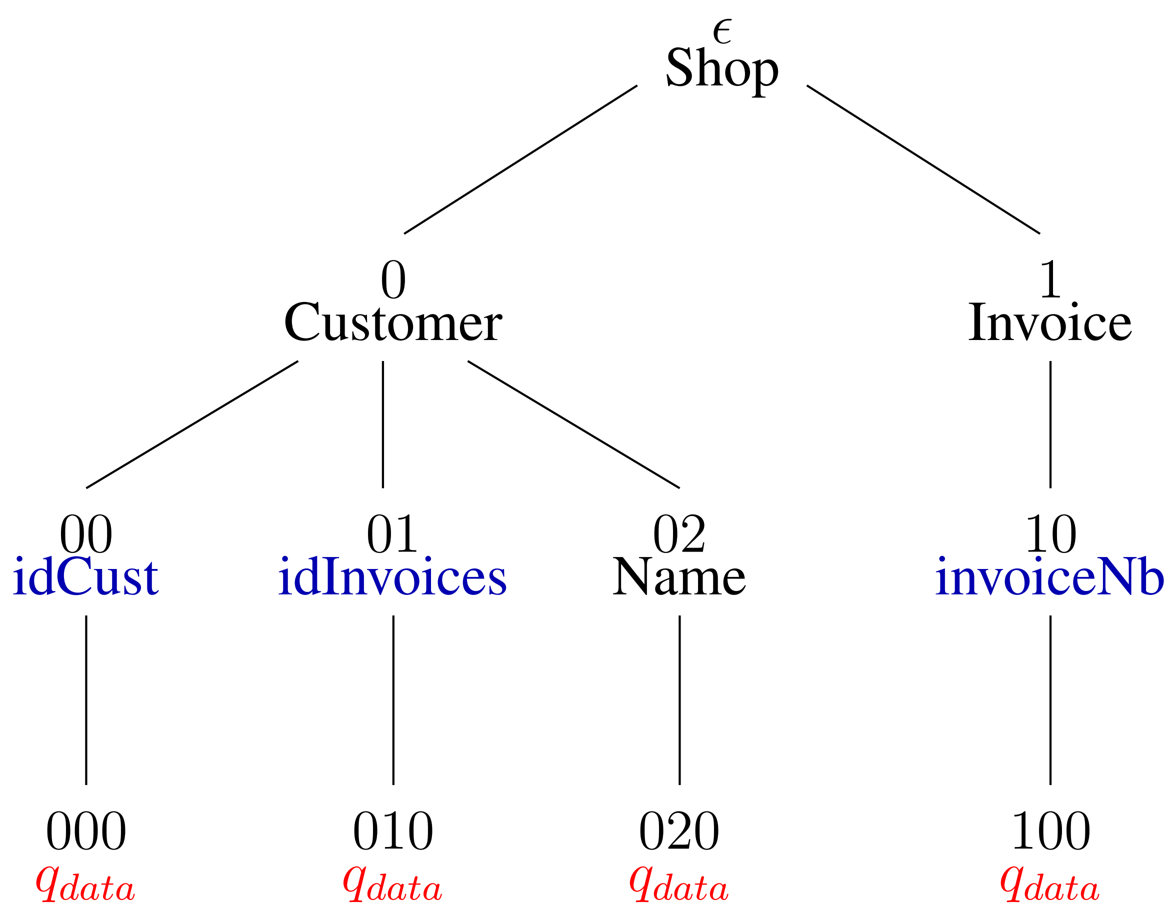
A run is successful if it contains at least one final state.

# Simplified Versions for LTG and STTG

- LTG: the sets of states are always singletons, only one rule for each label

- STTG: the results of a run can consider just a single type for each node of the tree

# Validation Example




$$Shop, (\emptyset, \emptyset), q^*_{Customer} q^*_{Invoice} \rightarrow q_{Shop}$$

$$Customer, (\{q_{idCust}\}, \{q_{idInvoices}\}), q_{Name} \rightarrow q_{Customer}$$

$$Invoice, (\{q_{invoiceNb}\}, \emptyset), \emptyset \rightarrow q_{Invoice}$$

$$idCust, (\emptyset, \emptyset), q_{data} \rightarrow q_{idCust}$$

$$idInvoices, (\emptyset, \emptyset), q_{data} \rightarrow q_{idInvoices}$$

$$Name, (\emptyset, \emptyset), q_{data} \rightarrow q_{Name}$$

$$invoiceNb, (\emptyset, \emptyset), q_{data} \rightarrow q_{invoiceNb}$$

# Slide Credits

- Validation Using Trees: structure&examples from Mirian Hayfield Ferrari
- Figures & examples in slides 8, 12, 13, 24 from C. Maneth's course