XML and Databases

Lecture 5
XML Validation using Automata

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2009

Outline

1. Recap: deterministic Reg Expr's
   / Glushkov Automaton

2. Complexity of DTD validation

3. Beyond DTDs: XML Schema and RELAX NG

4. Static Methods, based on Tree Automata

Previous Lecture

XML type definition languages

want to specify a certain subset of XML doc's = a "type" of XML documents

Remember
The specification/type definition should be **simple**, so that

→ a *validator* can be built automatically (and efficiently)
→ the *validator* runs efficient on any XML input

(similar demands as for a *parser*)

➔ Type def. language must be SIMPLE!

(similarly: parser generators use EBNF or smaller subclasses: LL / LR)

O(n^3) parsing

XML Type Definition Languages

DTD (Document Type Definition, W3C)
Originated from SGML. Now part of XML

→DTD may appear at the beginning of an XML document

Reg Exprs
must be
deterministic
(=1-unambiguous)

same!!

“*Unique*
Particle Attribution”

XML Schema (W3C)
Now at version 1.1
HUGE language, many built-in simple types

→Schemas themselves: written in XML

See the “Schema Primer” at http://www.w3.org/TR/xmlschema-0/

RELAX NG (Oasis)
For tree structure definition, more powerful than Schemas&DTDs

XML Type Definition Languages

DTD (Document Type Definition)

<!**DOCTYPE** root-element [doctype declaration …]>

<!**ELEMENT** element-name content-model>

content-models

- EMTPY
- ANY
- (#PCDATA | elem-name_1 | … | elem-name_n)*
- deterministic Reg Expr over element names

<!**ATTLIST** element-name attr-name attr-type attr-default ..>

Types: CDATA, (v1|..), ID, IDREFs
Defaults: #REQUIRED, #IMPLIED, “value”, #FIXED

XML Type Definition Languages

DTD (Document Type Definition)

```
<!DOCTYPE root-element [ doctype declaration …]>
```

```
<!ELEMENT element-name content-model>
```

content-models
- EMTPY
- ANY
- (#PCDATA | elem-name_1 | … | elem-name_n)*
- deterministic Reg Expr

Most interesting / challenging aspect of DTDs

```
<!ATTLIST element-name attr-name attr-type attr-default ..>
```

```
Types: CDATA, (v1|..), ID, IDREFs
Defaults: #REQUIRED, #IMPLIED, “value”, #FIXED
```

Summary

In order to check whether a (large) document
is **valid** wrt to a given DTD ("it validates")
you need to

→ check if children lists match the given Reg Expr's

This can be done *efficiently*, using **finite-automata (FAs)**!

To check if a Reg Expr e is **allowed in a DTD**
we have to construct a particular finite automaton: the $\underbrace{\textbf{Glushkov automaton}}_{\text{Glu(e)}}$.

Glu(e) must be *deterministic*.

Note If Glu(e) is *deterministic*, then its size (# transitions) is *linear* in size(e)!

Summary

In order to check whether a (large) document is **valid** wrt to a given DTD ("it validates") you need to

→ check if children lists match the given Reg Expr's

This can be done *efficiently*, using **finite-automata (FAs)**!

To check if a Reg Expr e is **allowed in a DTD** we have to construct a particular finite automaton: the $\underbrace{\textbf{Glushkov automaton}}_{\text{Glu(e)}}$.

Glu(e) must be *deterministic*.

Note If Glu(e) is *deterministic*, then its size (# transitions) is *linear* in size(e)!

Question Can you explain why this is the case?

Summary

In order to check whether a (large) document
is **valid** wrt to a given DTD ("it validates")
you need to

→ check if children lists match the given Reg Expr's

This can be done *efficiently*, using **finite-automata (FAs)**!

To check if a Reg Expr e is **allowed in a DTD**
we have to construct a particular finite automaton: the $\underbrace{\textbf{Glushkov automaton}}_{\text{Glu(e)}}$.

Glu(e) must be *deterministic*.

Note If Glu(e) is *deterministic*, then its size (# transitions) is ~~*linear*~~ in size(e)!

Question Can you explain why this is the case?

not correct:
linear in **size(e) * #letters(e)**

More Notes

(1) From a *deterministic* FA you can**not** necessarily obtain a deterministic (= 1-unambiguous) regular expression!!

Example: e = (a | b)* a (a | b) ← NO 1-unambigous reg exp exists for e

---

To check if a Reg Expr e is **allowed in a DTD**
we have to construct a particular finite automaton: the $\underbrace{\textbf{Glushkov automaton}}_{\text{Glu(e)}}$.

Glu(e) must be *deterministic*.

Note If Glu(e) is *deterministic*, then its size (# transitions) is ~~*linear*~~ in size(e)!

not correct:
linear in **size(e) * #letters(e)**

Question Can you explain why this is the case?

More Notes

For more details:
See paper by Brüggemann-Klein,
Linked from the course web-page.

(1) From a *deterministic* FA you can**not** necessarily obtain a deterministic (= 1-unambiguous) regular expression!!

Example: e = (a | b)* a (a | b) ← NO 1-unambigous reg exp exists for e

(2) Glu(e) is closely related to → Thomson(e) [remove ε-transitions]
and to → Berry/Sethi(e) [same]
and → Brzozowski(e)

To check if a Reg Expr e is **allowed in a DTD**
we have to construct a particular finite automaton: the **Glushkov automaton**.

Glu(e)

Glu(e) must be *deterministic*.

Note If Glu(e) is *deterministic*, then its size (# transitions) is ~~*linear*~~ in size(e)!

Question Can you explain why this is the case?

not correct:
linear in **size(e) * #letters(e)**

Glushkov automaton Glu(e)

Each letter-position in the Reg Expr e becomes one state of Glu; plus, Glu has one extra begin state.

FIRST(e) = all possible begin positions of words matching e

e.g. FIRST(R (E | G) (EX)*) = { R_1 }

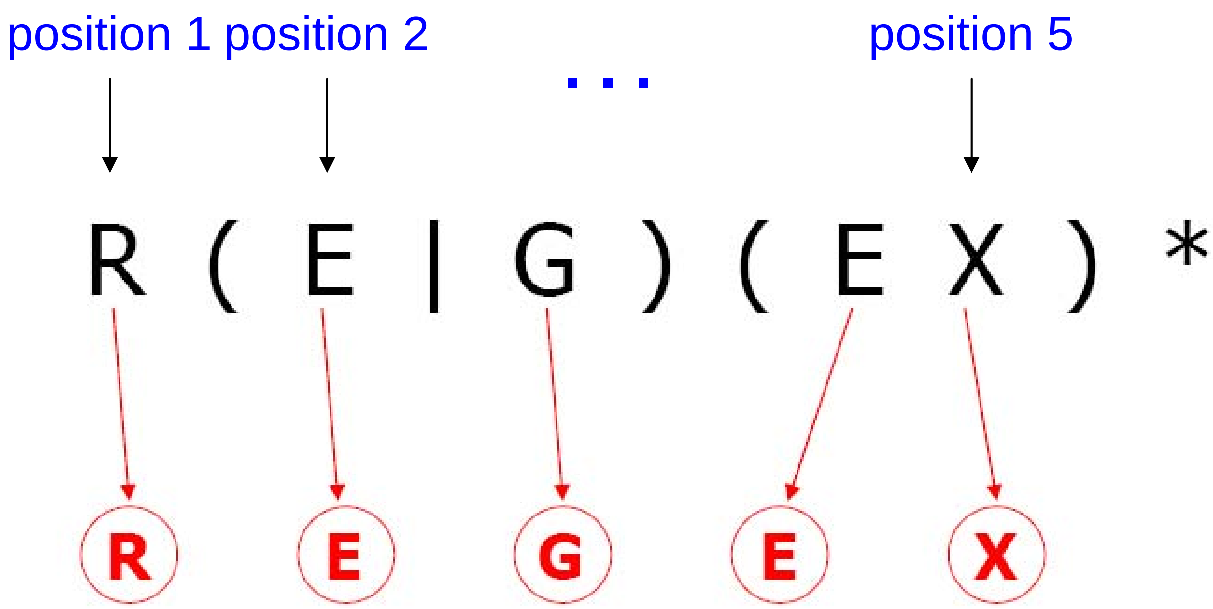
Glushkov's automaton

R (E | G) (E X) *

Following slides from: http://www.cs.ut.ee/~varmo/tday-rouge/tammeoja-slides.pdf

Glushkov's automaton

- Character in RE = **state** in automaton



Glushkov's automaton

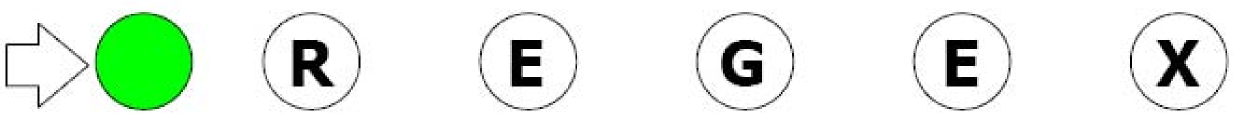
- Character in RE = **state** in automaton
  + one state for the beginning of the RE

R (E | G) (E X) *

Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

R (E | G) (E X) *

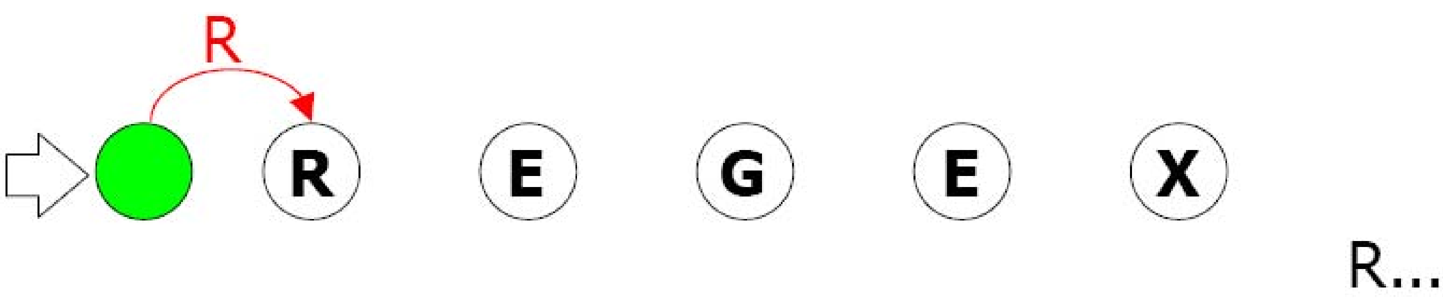


R...

Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

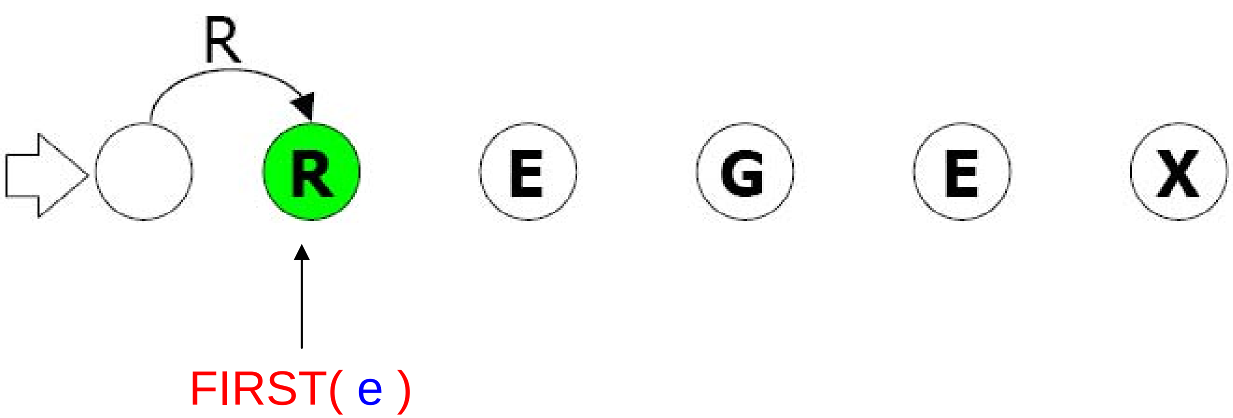
R (E | G) (E X) *



Glushkov's automaton

- Character in RE = **state** in automaton
  \+ one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

R (E | G) (E X) *



R...

Glushkov automaton G(e)

Each position in the Reg Expr e becomes one state of G;
plus, G has one extra begin state.

FIRST(e) = all possible begin positions of words matching e

e.g. FIRST(R (E | G) (EX)*) = { R_1 }

FOLLOW(e, x) = all possible positions following position x in e

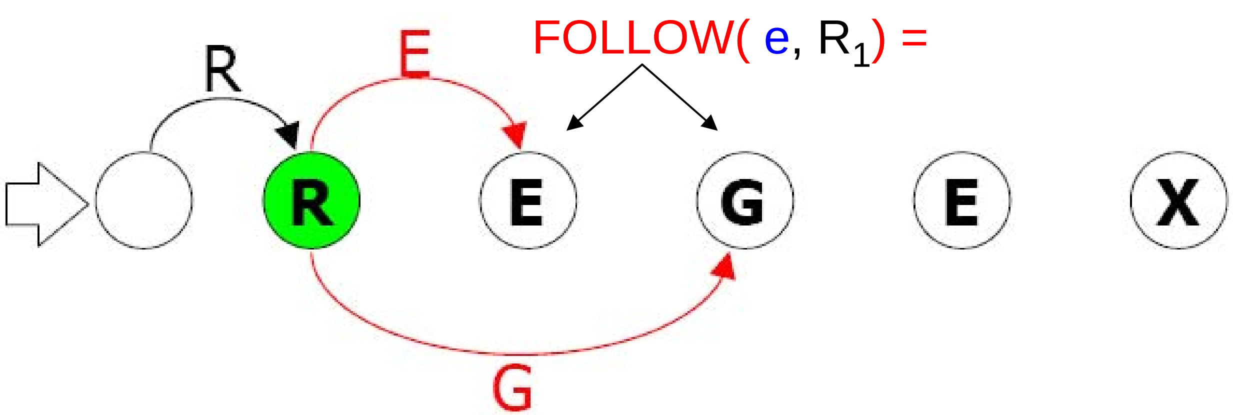
e.g. FOLLOW(R (E | G) (EX)*, R_1) = { E_2, G_3 }

➔ From state "R_1": add E-transition to E_2
G-transition to G_3

Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

R (E | G) (E X) *

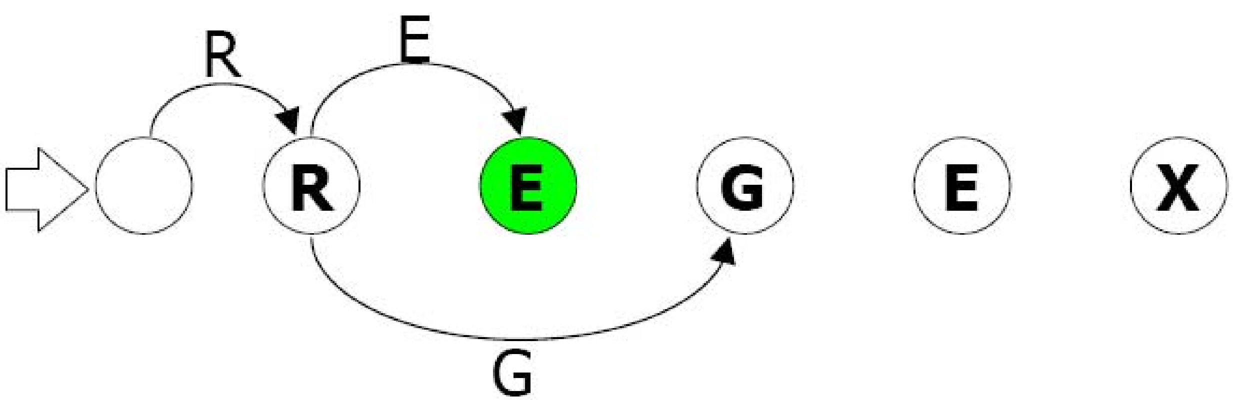


RE...
RG...

Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

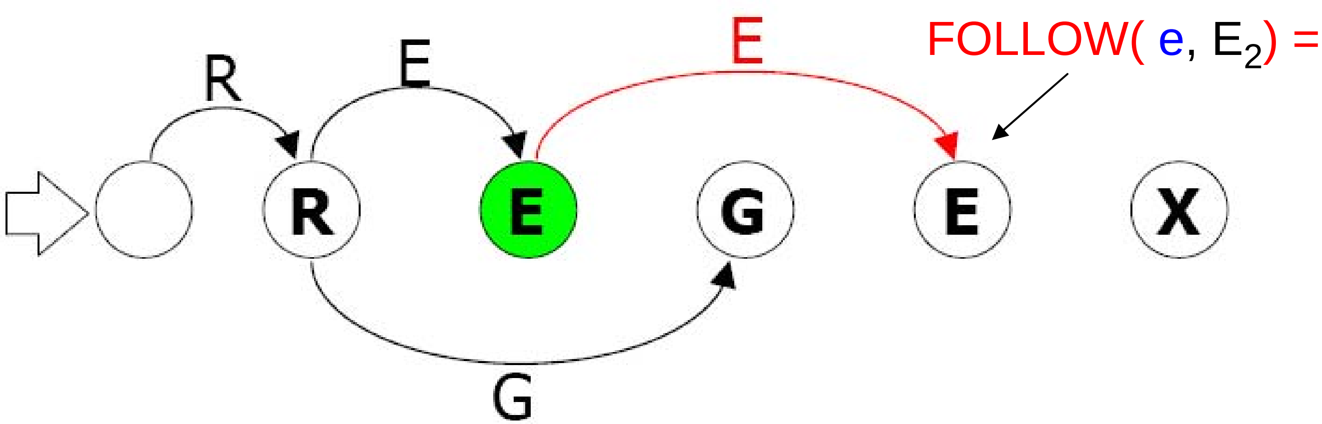
R (E | G) (E X) *



Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

R (E | G) (E X) *

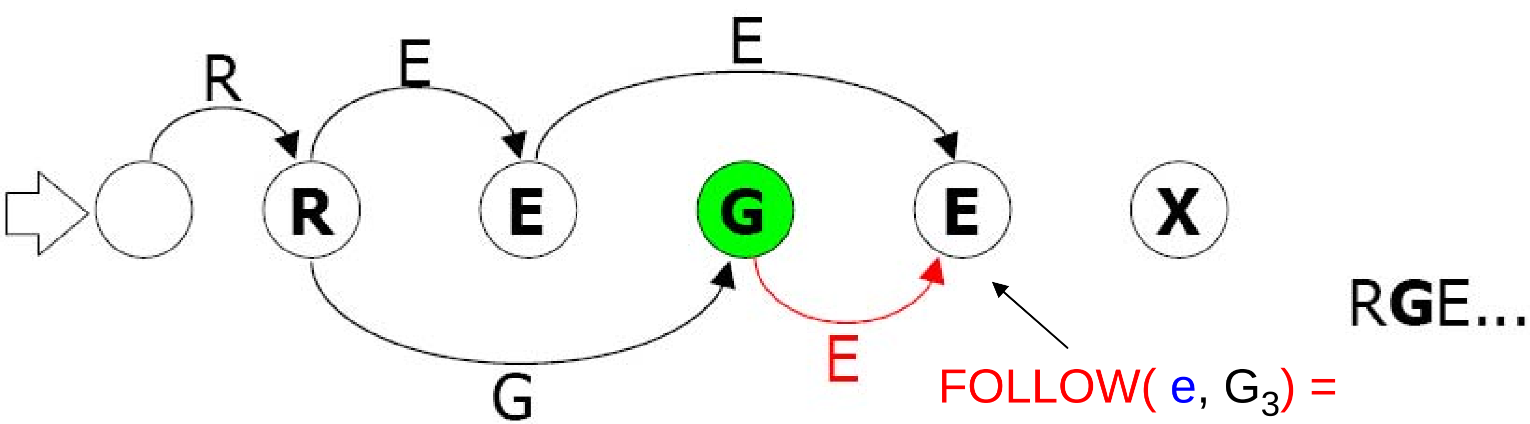


REE...

Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

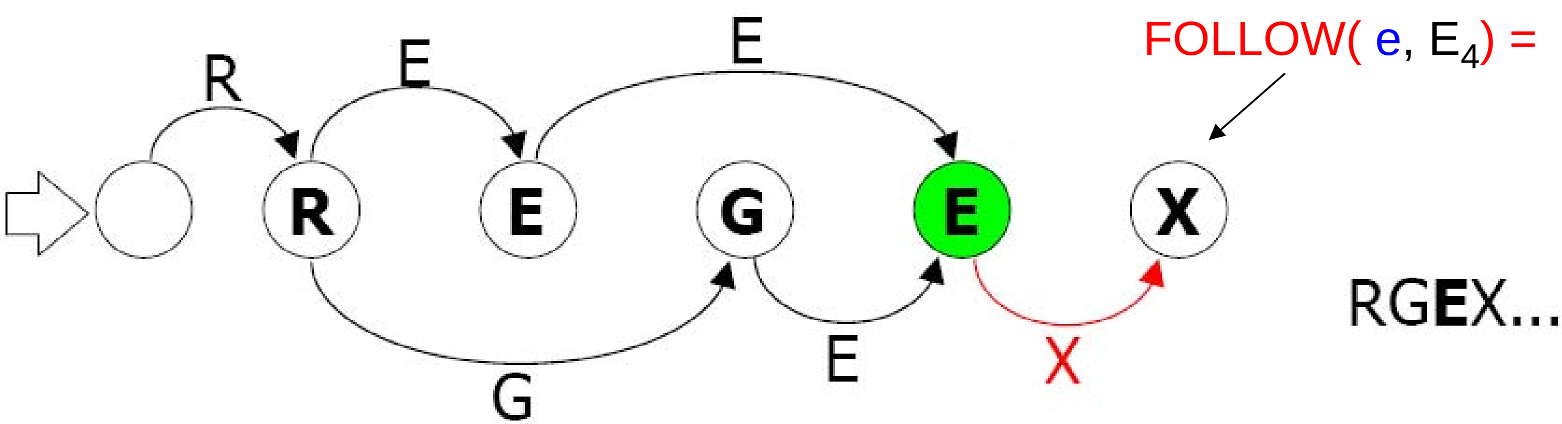
R (E | G) (E X) *



Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

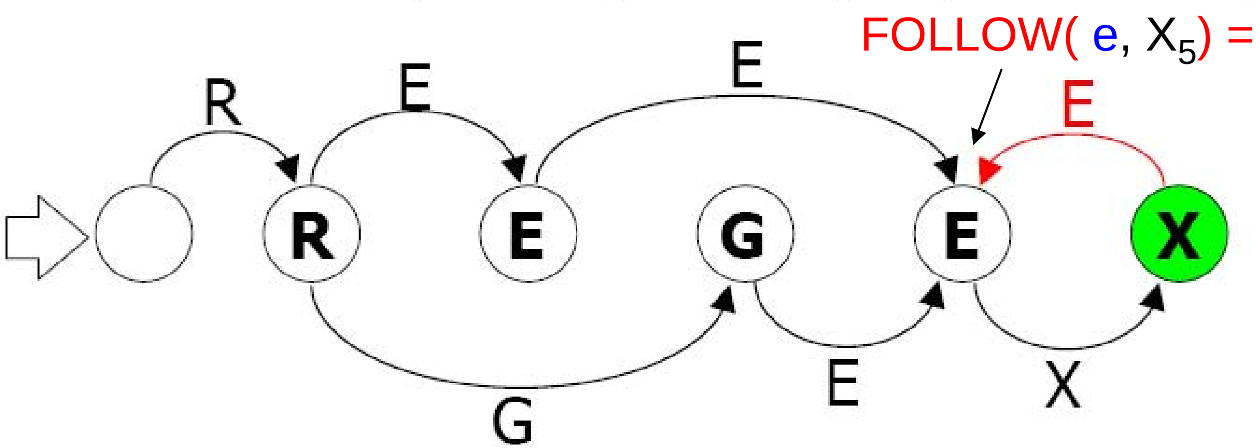
R (E | G) (E X) *



Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

R (E | G) (E X) *



RGE**X**E...

Glushkov automaton G(e)

Each position in the Reg Expr e becomes one state of G;
plus, G has one extra begin state.

FIRST(e) = all possible begin positions of words matching e

e.g. FIRST(R (E | G) (EX)*) = { R_1 }

FOLLOW(e, x) = all possible positions following position x in e

e.g. FOLLOW(R (E | G) (EX)*, R_1) = { E_2, G_3 }

➔ From state "R_1":   add   E-transition to E_2
G-transition to G_3

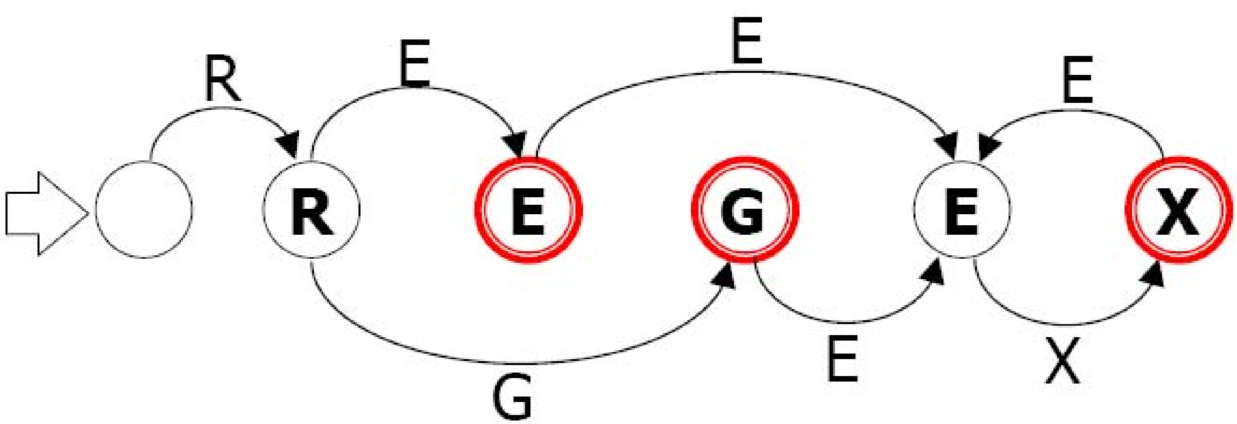
LAST(e) = all possible *end* positions of words matching e

e.g.   LAST(R (E | G) (EX)*) = { E_2, G_3, X_5 }

Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

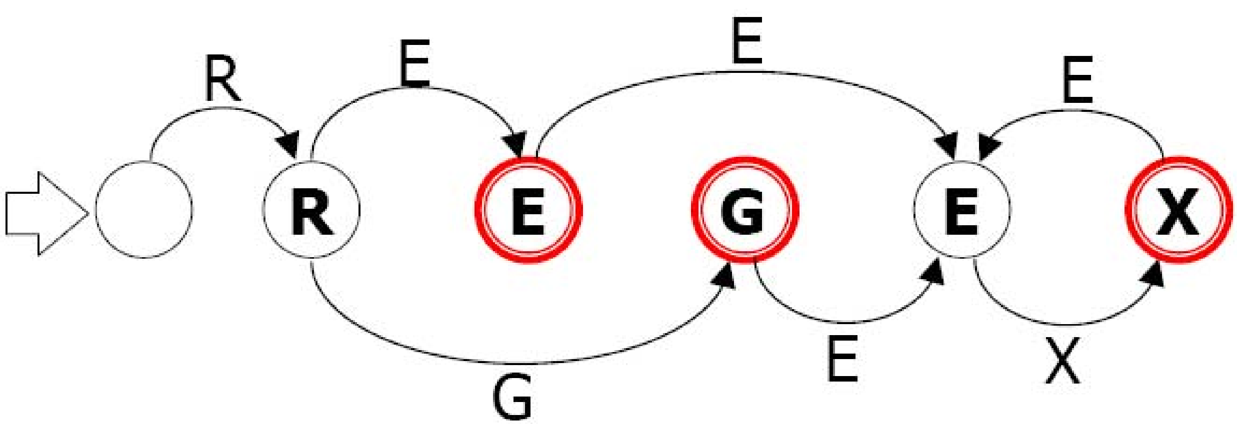
R (E | G) (E X) *



Glushkov's automaton

- Character in RE = **state** in automaton
  + one state for the beginning of the RE
- **Transitions** show which characters/positions can precede each other

R (E | G) (E X) *



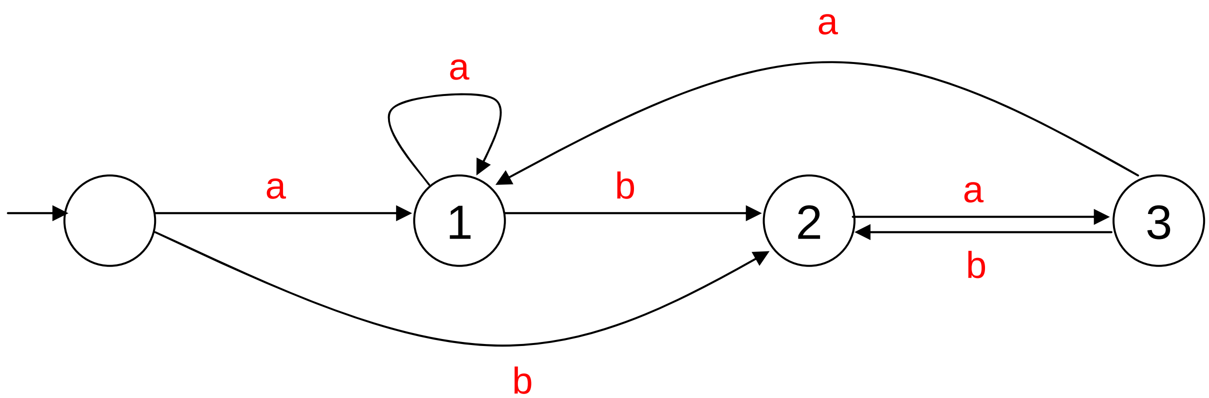
Is this automaton *deterministic* ??

Glushkov automaton G(e)

Another example

(a* | ba)*

This FA
is deterministic.



Which of these is deterministic?

→ (ab) | (ac)
→ a (b | c)
→ a(a | b)*ac

Glushkov automaton G(e)

Each position in the Reg Expr e becomes one state of G;
plus, G has one extra begin state.

FIRST(e) = all possible *begin* positions of words matching e

e.g. FIRST(R (E | G) (EX)*) = { R_1 }

FOLLOW(e, x) = all possible positions *following* position x in e

LAST(e) = all possible *end* positions of words matching e

Naïve implementation: O(n^3) time, where n = size(e)

(for each position: computing FOLLOW goes through every position
at each step, needs to compute *union* ➔ O(n*n*n)

Glushkov automaton G(e)

Each position in the Reg Expr e becomes one state of G;
plus, G has one extra begin state.

FIRST(e) = all possible *begin* positions of words matching e

e.g. FIRST(R (E | G) (EX)*) = { R_1 }

FOLLOW(e, x) = all possible positions *following* position x in e

LAST(e) = all possible *end* positions of words matching e

Naïve implementation: O(n^3) time, where n = size(e)

(for each position: computing FOLLOW goes through every position
at each step, needs to compute *union* ➔ O(n*n*n)

Not really needed. Can be improved to O(n^2)

Glushkov automaton G(e)

Each position in the Reg Expr e becomes one state of G;
plus, G has one extra begin state.

FIRST(e) = all possible *begin* positions of words matching e

e.g. FIRST(R (E | G) (EX)*) = { R_1 }

FOLLOW(e, x) = all possible positions *following* position x in e

LAST(e) = all possible *end* positions of words matching e

Naïve implementation: O(n^3) time, where n = size(e)

(for each position: computing FOLLOW goes through every position
at each step, needs to compute *union* ➔ O(n*n*n)

Can be improved to
O(size(e) + size(G(e)))

Not really needed. Can be improved to O(n^2)

Glushkov automaton G(e)

Note If G(e) is *deterministic*, then its size (# transitions) is *quadratic* in size(e)!

Linear in size(e) * #letters(e), if G(e) is deterministic!

➔ O(size(e) * #letters(e))

Naïve implementation: O(n^3) time, where n = size(e)

(for each position: computing FOLLOW goes through every position
at each step, needs to compute *union* ➔ O(n*n*n)

Can be improved to
O(size(e) + size(G(e)))

Not really needed. Can be improved to O(n^2)

To avoid these expensive running times

DTD requires that `FA=G(e)` must be *deterministic*!

n = length(w)
m = size(e)

Total Running time $O(n + m)$

If s = #letters(e) is assumed fixed (not part of the input)

Otherwise: $O(n + ms)$

How can you **implement** a regular expression?

Input: Reg Expr e, string w
Question: Does w match e?

deterministic FA: run on w takes time **linear** in length(w)

Unrestricted Reg Expr e →

Algorithm

```
FA = BuildFA(e);
DFA = BuildDFA(FA);
```

Size of FA is linear in size(e)=m
Size of DFA is exponential in m

Total Running time $O(n + 2^m)$

→ Other alternative: $O(nm)$

Summary

Deterministic (1-unambiguous) content models give rise to *efficient matching algorithms*.

(they avoid O(nm)
or O(n+2^m) complexities)

Disadvantages

→ Hard to know whether given reg expr is OK (deterministic)

→ Det. reg exprs are NOT closed under union. (not so nice..)

Question Can you see why?

Hint: find det. reg. exprs. e1 and e2 such that their union is equal to (a | b)* a (a | b)

Now that we know how the check all the different content-models (in particular det. Reg Expr's) how to build full validator for a DTD?

elem-name_1 → RegExpr_1
elem-name_2 → RegExpr_2
…
elem-name_k → RegExpr_k

Automata A_1, A_2, …, A_k

The Validation Problem
Given a DTD T and a document D, is D valid wrt T?

Top-Down Implementation
→ at element node w. label elem-name_i, run automaton A_i

→ check attribute constraints
→ check ID/IDREF constraints

(Given A_1, A_2, …, A_k)

Total Running time *linear in the sum of sizes of the DTD and the document*. O(size(T) + size(D))

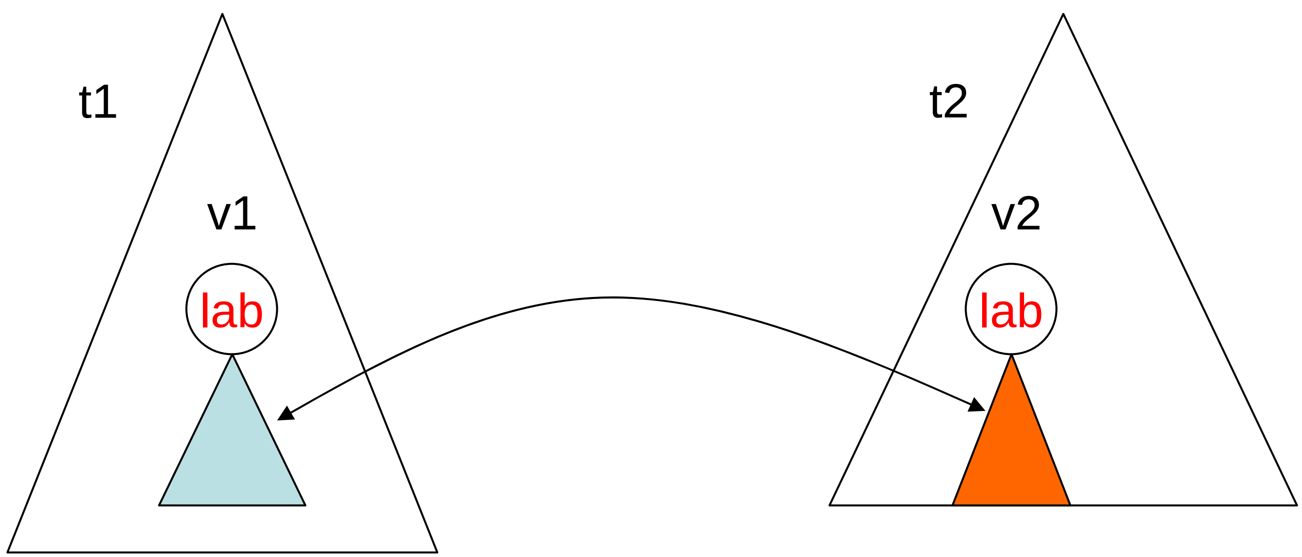
DTDs have the

“**label-guarded subtree exchange**” property:

t1, t2 trees in a DTD language T
v1 node in t1, labeled “lab”
v2 node in t2, labeled “lab”

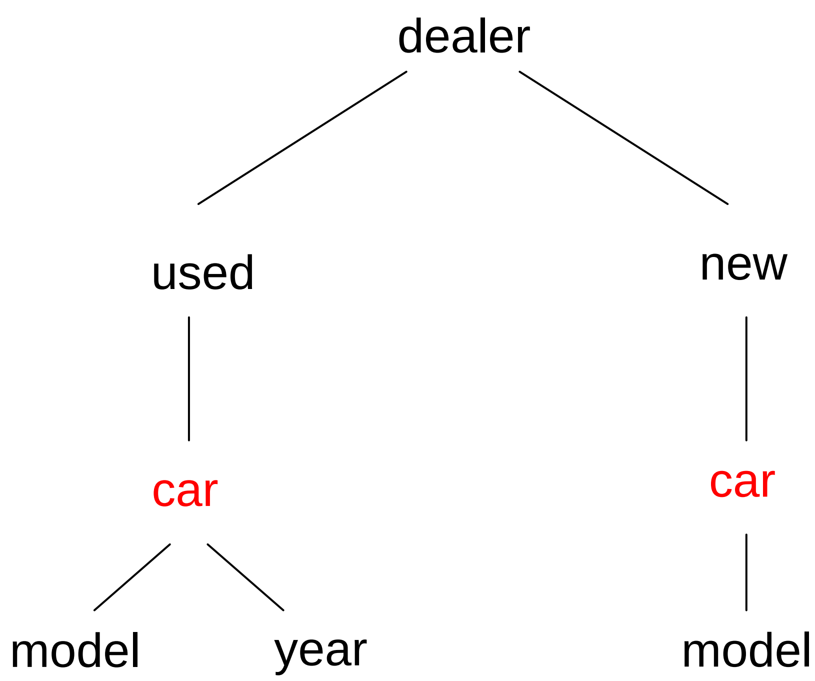
trees obtained by exchanging the subtrees rooted at v1 and v2 are also in T

aka “local”
→ content model only depends on label of parent



Beyond DTDs

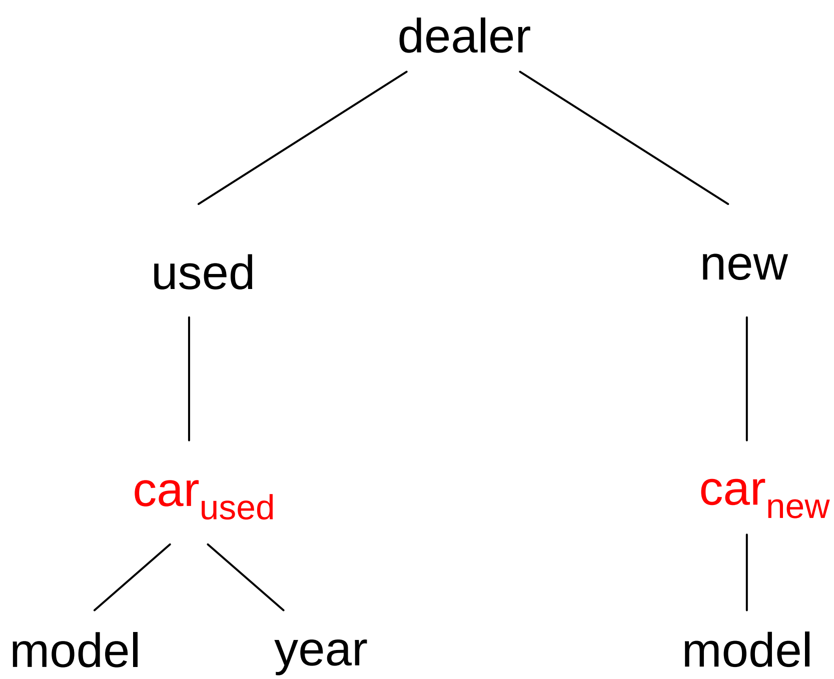
Often, the expressive power of DTDs is *not sufficient*.
Problem each element name has precisely one content-model in a DTD.
Would like to distuingish, depending on the context (parent).



car has different structure, in different contexts.

Beyond DTDs

Often, the expressive power of DTDs is *not sufficient*.
Problem each element name has precisely one content-model in a DTD.
Would like to distuingish, depending on the context (parent).

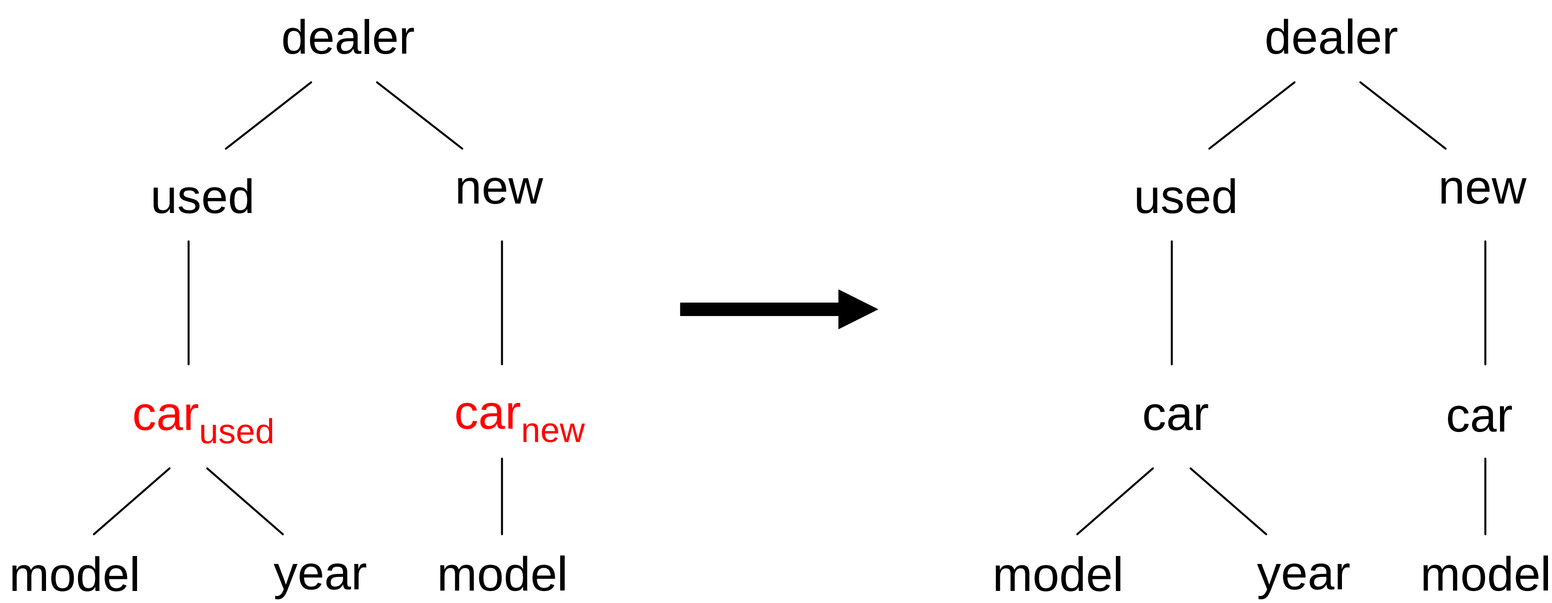


“specialization”

car has different structure, in different contexts.

Specialized DTDs

dealer → used, new
used → $(car_{used})^*$
new → $(car_{new})^*$
car_{used} → model, year
car_{new} → model

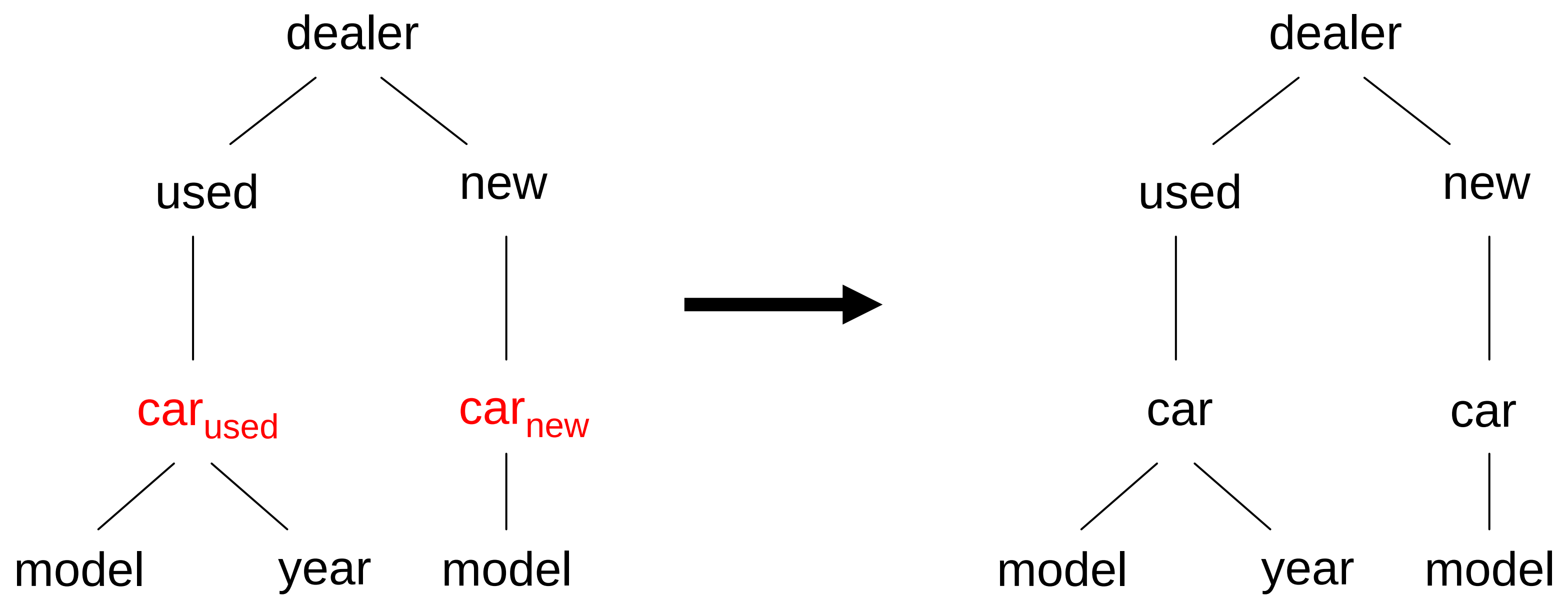


Specialized DTDs

dealer → used, new
used → (car_{used})*
new → (car_{new})*
car_{used} → model, year
car_{new} → model

New notation. Use *capitalized* TYPE Names

Dealer → dealer [Used, New]
Used → used [(Car_{used})*]
New → new [(Car_{new})*]
Car_{used} → car [Model, Year]
Car_{new} → car [Model]



New notation. Use *capitalized* TYPE Names

Dealer → dealer [Used, New]
Used → used [(Car_{used})*]
New → new [(Car_{new})*]
Car_{used} → car [Model, Year]
Car_{new} → car [Model]

Not local

Let us call this new concept a "grammar".

the "*local*" restriction

A grammar G is **local**, if
for any label[RegExpr_1], label[RegExpr_2] present in G
it holds that RegExpr_1 = RegExpr_2.

By definition: Every DTD is a local grammar, and vice versa.

New notation. Use *capitalized* TYPE Names

Dealer → dealer [Used, New]
Used → used [(Car_{used})*]
New → new [(Car_{new})*]
Car_{used} → car [Model, Year]
Car_{new} → car [Model]

Not local

Let us call this new concept a "grammar".

the "*local*" restriction

A grammar G is **local**, if
for any label[RegExpr_1], label[RegExpr_2] present in G
it holds that RegExpr_1 = RegExpr_2.

By definition: Every DTD is a local grammar, and vice versa.

~~A grammar G is **single-type**, if~~
~~for any label[RegExpr_1], label[RegExpr_2] occurring *in the same rule of G*~~
~~it holds that RegExpr_1 = RegExpr_2.~~

the "*single-type*" restriction

WRONG

New notation. Use *capitalized* TYPE Names

	Dealer	→	dealer [Used, New]
	Used	→	used [(Car_{used})*]
	New	→	new [(Car_{new})*]
competing	Car_{used}	→	**car** [Model, Year]
	Car_{new}	→	**car** [Model]

Alternatively:

Call two TYPE Names T1 and T2 "competing"
if they have the same element name (but not identical rules)

Classes of Grammars

local no competing TYPE names! (DTDs)

single-type TYPE names in the *same content model* do not compete!
(XML Schema's)

regular no restriction… (RELAX NG)

New notation. Use *capitalized* TYPE Names

Dealer → dealer [Used, New]
Used → used [(Car_{used})*]
New → new [(Car_{new})*]
competing Car_{used} → **car** [Model, Year]
Car_{new} → **car** [Model]

Question Are there single-type grammars (XML Schemas) which cannot be expressed by local grammars (DTDs).

Classes of Grammars

local no competing TYPE names! (DTDs)

single-type TYPE names in the *same content model* do not compete! (XML Schema's)

regular no restriction… (RELAX NG)

New notation. Use *capitalized* TYPE Names

competing

but are not
in same
content model!

Person → person [PersonName, Gender, Spouse?, Pet*]
PersonName → name [First, Last]
Pet → pet [Kind, PetName]
PetName → name [#PCDATA]
…

Question Are there single-type grammars (XML Schemas) which cannot be expressed by local grammars (DTDs).
YES!

Classes of Grammars

local no competing TYPE names! (DTDs)

single-type TYPE names in the *same content model* do not compete! (XML Schema's)

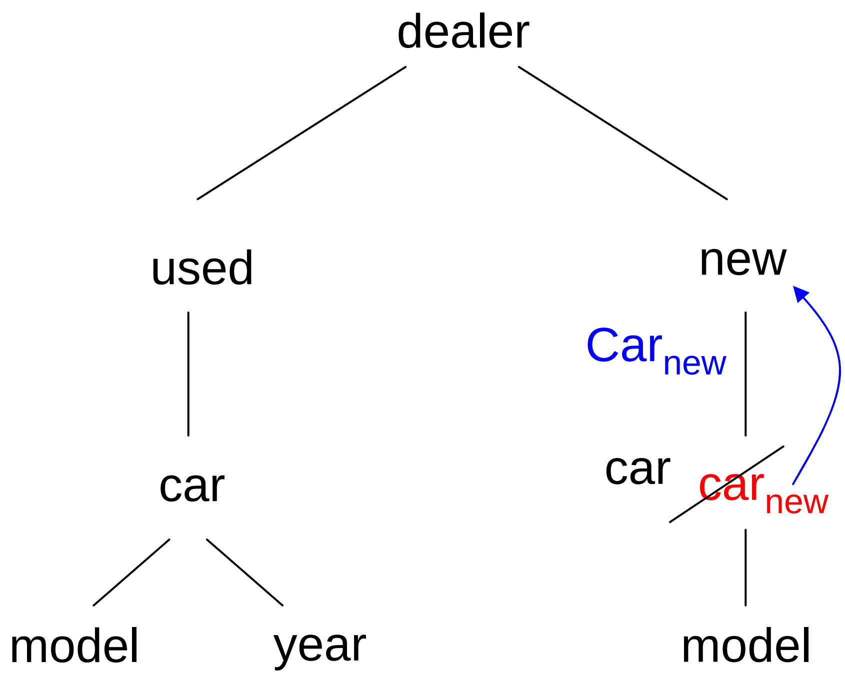
regular no restriction… (RELAX NG)

New notation. Use *capitalized* TYPE Names

competing

Dealer → dealer [Used, New]
Used → used [(Car_{used})*]
New → new [(Car_{new})*]
Car_{used} → **car** [Model, Year]
Car_{new} → **car** [Model]

Through the use of TYPE Names (nonterminals / states) you can distinguish **deep context**!



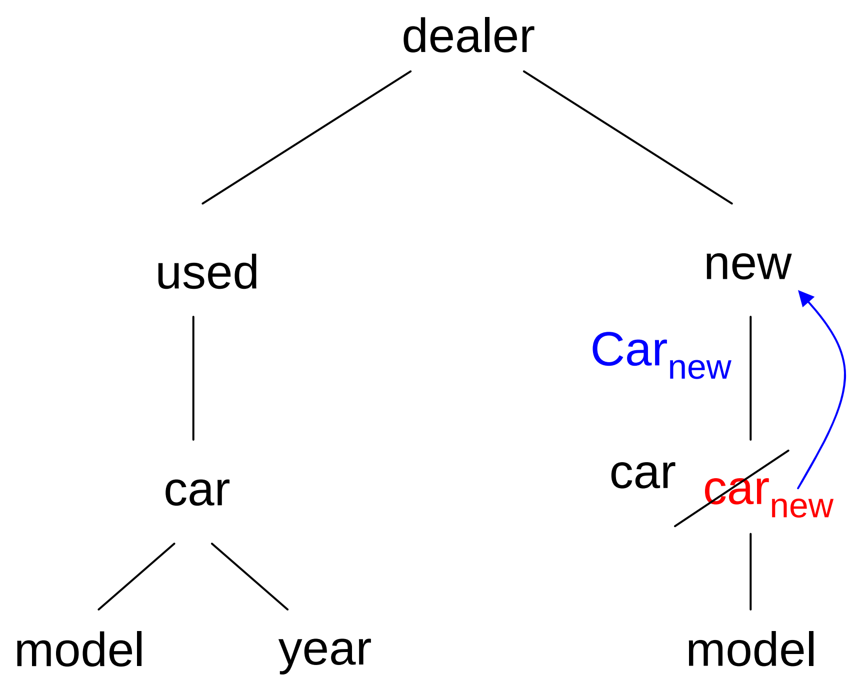
"specialization"
through parent

New notation. Use *capitalized* TYPE Names

Dealer → dealer [Used, New]
Used → used [(Car_{used})*]
New → new [(Car_{new})*]
competing
Car_{used} → **car** [Model, Year]
Car_{new} → **car** [Model]

Through the use of TYPE Names (nonterminals / states) you can distinguish **deep context**!

Can we model context that is far away from the specialized node?



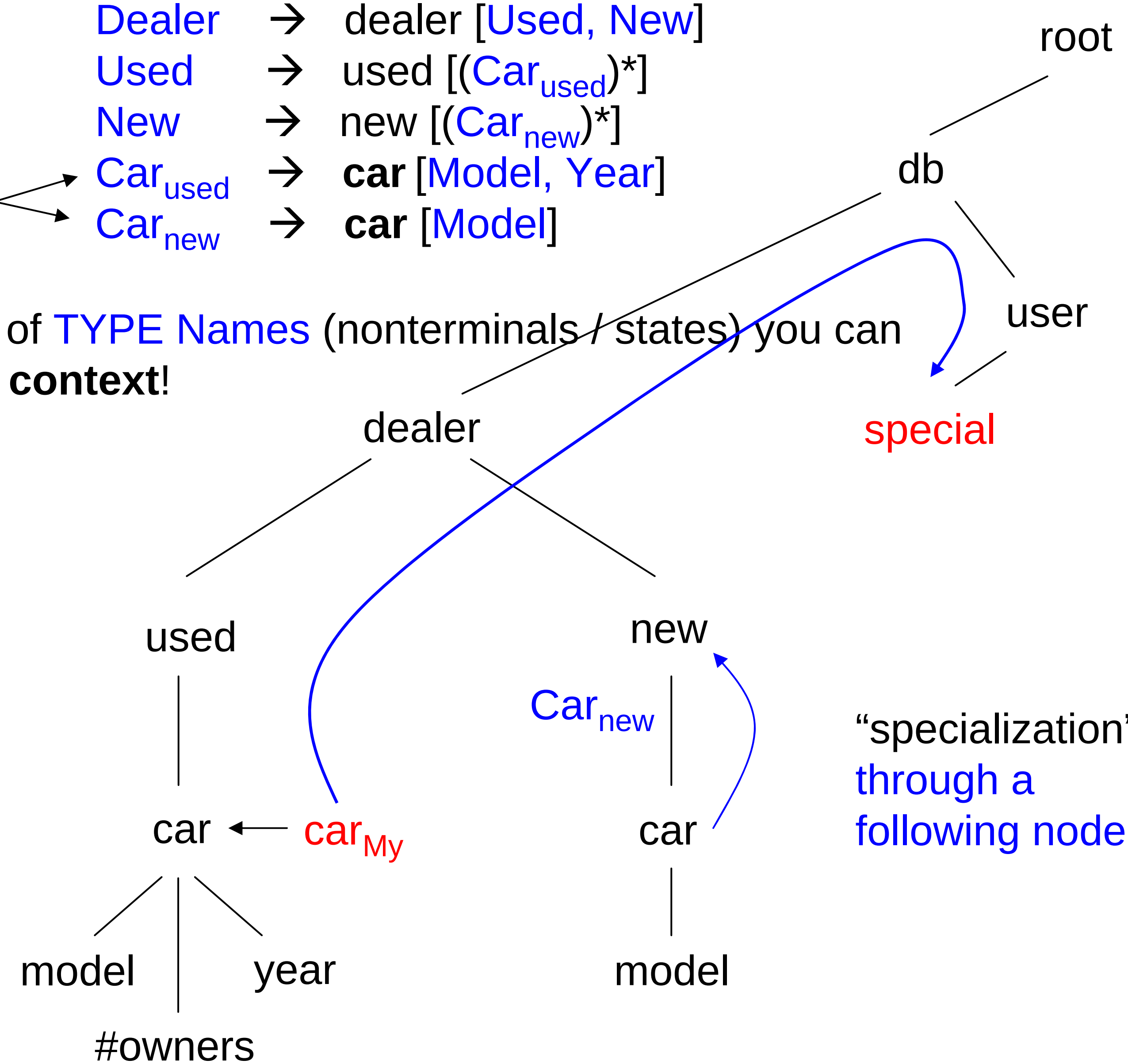
“specialization” through parent

New notation. Use *capitalized* TYPE Names

Dealer → dealer [Used, New]
Used → used [(Car_{used})*]
New → new [(Car_{new})*]
Car_{used} → **car** [Model, Year]
Car_{new} → **car** [Model]

competing (Car_{used}, Car_{new})

Through the use of TYPE Names (nonterminals / states) you can distinguish **deep context**!



Can we model context that is far away from the specialized node?

Sure!

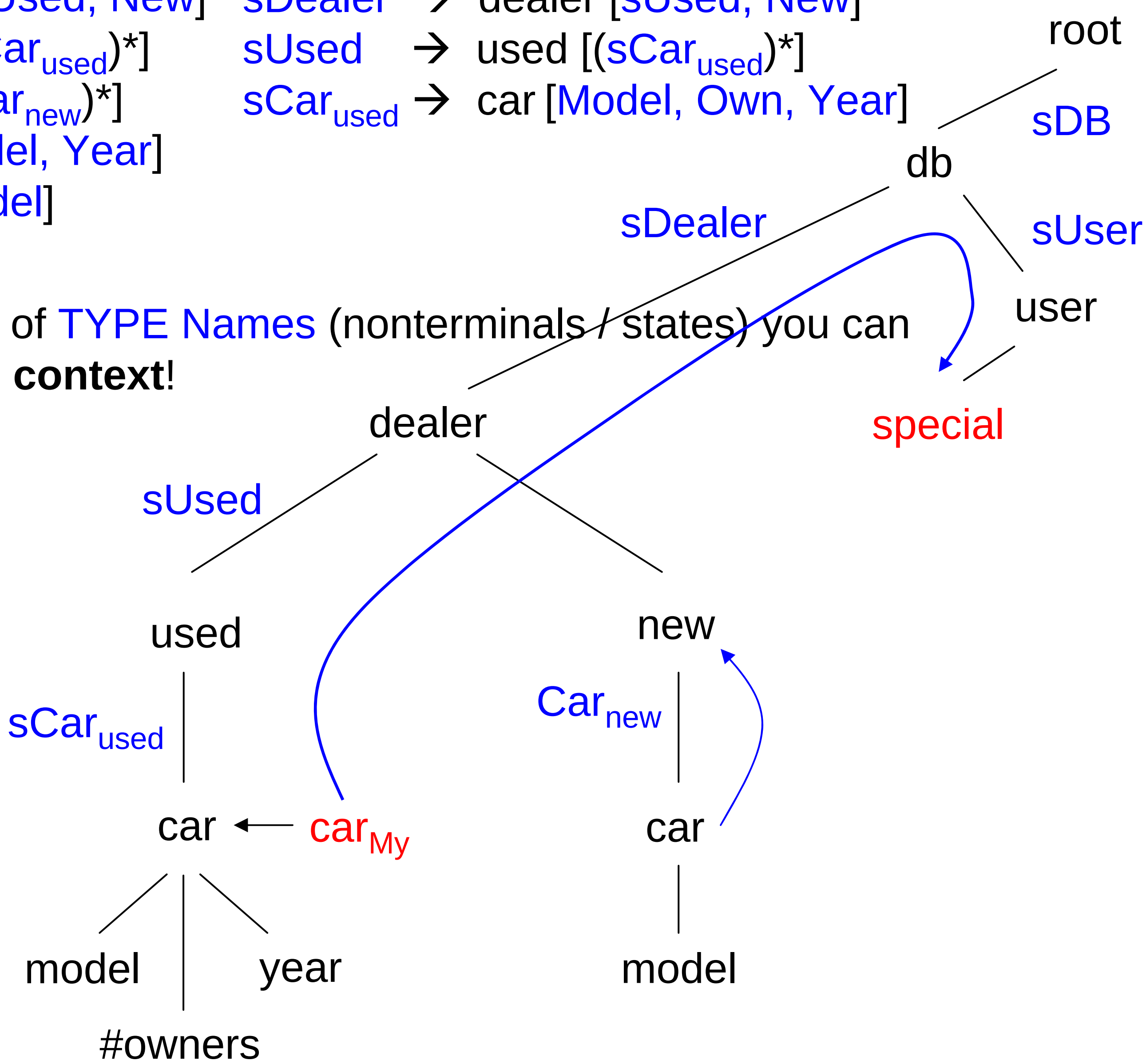
“specialization” through a following node…

Doc $\rightarrow$ root [DB | sDB]

DB $\rightarrow$ db [Dealer, User]
Dealer $\rightarrow$ dealer [Used, New]
Used $\rightarrow$ used [(Car$_{used}$)*]
New $\rightarrow$ new [(Car$_{new}$)*]
Car$_{used}$ $\rightarrow$ car [Model, Year]
Car$_{new}$ $\rightarrow$ car [Model]

sDB $\rightarrow$ db [sDealer, sUser]
sDealer $\rightarrow$ dealer [sUsed, New]
sUsed $\rightarrow$ used [(sCar$_{used}$)*]
sCar$_{used}$ $\rightarrow$ car [Model, Own, Year]

Through the use of TYPE Names (nonterminals / states) you can distinguish **deep context**!



Can we model context that is far away from the specialized node?

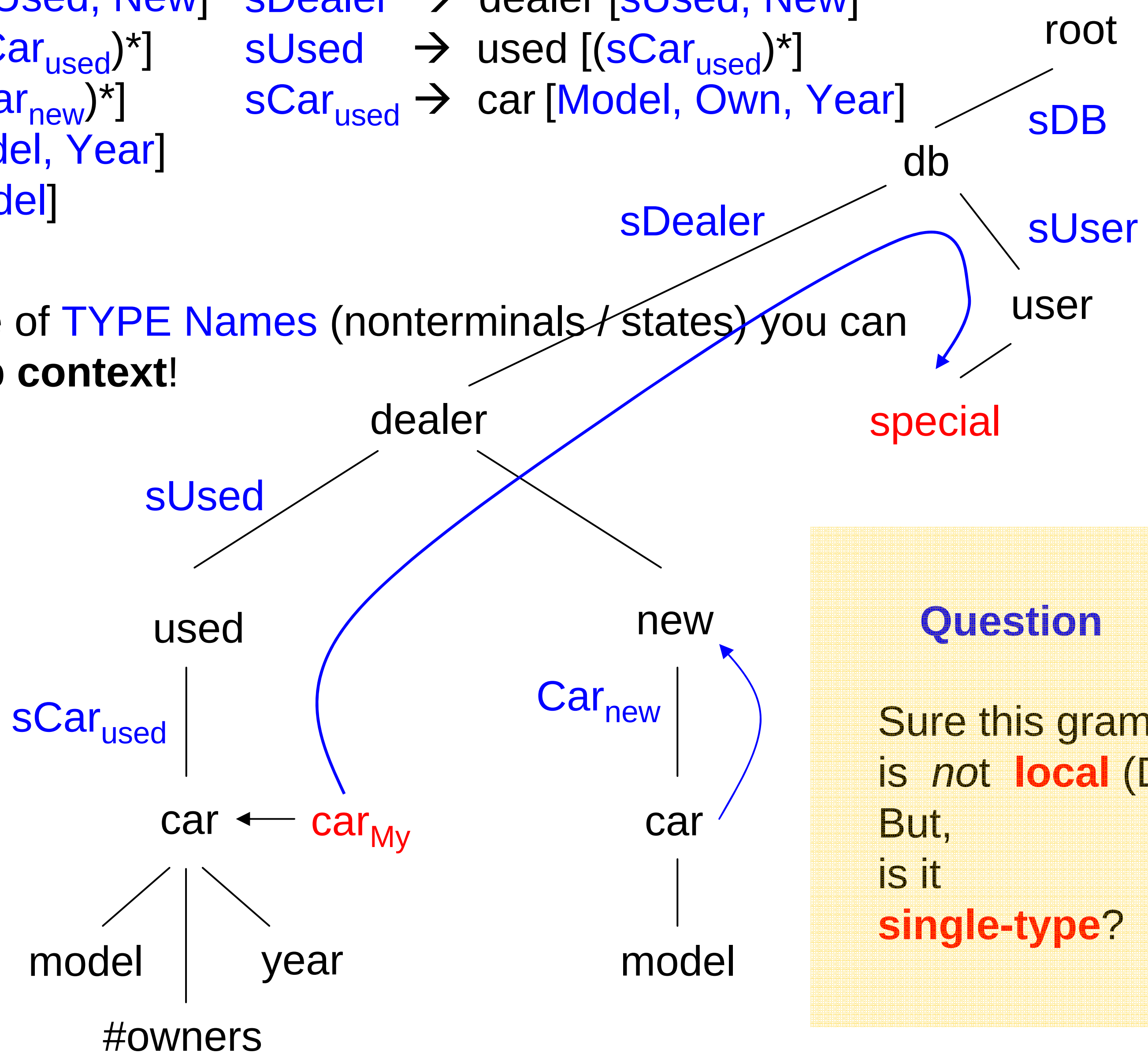
Sure!

Doc → root [DB | sDB]

DB → db [Dealer, User]
Dealer → dealer [Used, New]
Used → used [(Car_{used})*]
New → new [(Car_{new})*]
Car_{used} → car [Model, Year]
Car_{new} → car [Model]

sDB → db [sDealer, sUser]
sDealer → dealer [sUsed, New]
sUsed → used [($sCar_{used}$)*]
$sCar_{used}$ → car [Model, Own, Year]

Through the use of TYPE Names (nonterminals / states) you can distinguish **deep context**!

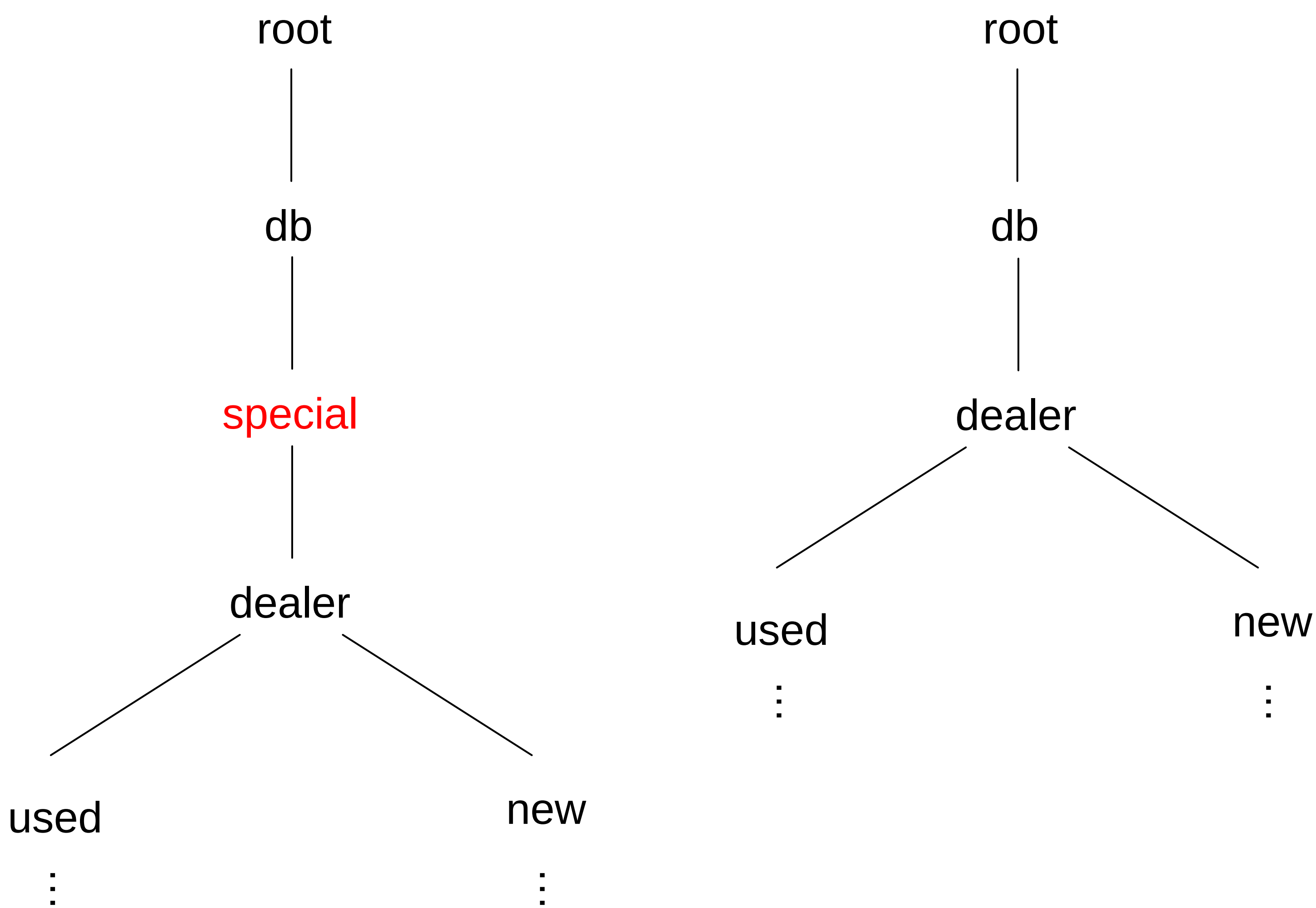


Can we model context that is far away from the specialized node?

Sure!

Question

Sure this grammar is *not* **local** (DTD).
But,
is it
single-type?



Question Is this grammar **single-type**?

prev. example:
probably, *not expressable* in single-type (XML Schema).

Other example:

Person → MPerson | FPerson
MPerson → person[Name, gender[Male], FSpouse?, Children?]
FPerson → person[Name, gender[Female], MSpouse?, Children?]
Male → male[]
Female → female[]
FSpouse → spouse[Name, gender[Female]]
MSpouse → spouse[Name, gender[Male]]
Children → children[Person+]

a person’s spouse must have
opposite gender.

Note This example and the Pet-example are taken from Hosoya’s book (see course web page).

prev. example:
probably, *not expressable* in single-type (XML Schema).

Other example:

competing

Reg Expr
… but ***not*** in a content …

Person → MPerson | FPerson
MPerson → **person**[Name, gender[Male], FSpouse?, Children?]
FPerson → **person**[Name, gender[Female], MSpouse?, Children?]
Male → male[]
Female → female[]
FSpouse → spouse[Name, gender[Female]]
MSpouse → spouse[Name, gender[Male]]
Children → children[Person+]

a person's spouse must have
opposite gender.

BUT, is this even a "grammar" in our sense?

prev. example:
probably, *not expressable* in single-type (XML Schema).

Reg Expr
… **in** a content …

Other example:

competing

Person → root[MPerson | FPerson]
MPerson → person[Name, gender[Male], FSpouse?, Children?]
FPerson → person[Name, gender[Female], MSpouse?, Children?]
Male → male[]
Female → female[]
FSpouse → spouse[Name, gender[Female]]
MSpouse → spouse[Name, gender[Male]]
Children → children[Person+]

a person's spouse must have
opposite gender.

BUT, is this even a "grammar" in our sense?
NO!
→ Reg Expr only allowed inside a content ("under an element name").

Classes XML Type Formalisms

local no competing TYPE names! (**DTDs**)

single-type TYPE names in the *same content model* do not compete! (**XML Schema's**)

regular no restriction… (**RELAX NG**)

Increasing Expressivness
of defining sets of trees ("tree languages")

Questions

Given two DTDs D1 and D2 can we check if
→ all documents valid for D1 are also valid for D2? (DTD inclusion problem)
→ D1 and D2 describe the same set of documents? (DTD equality problem)

Given a Relax NG grammar G, can we check if
→ there exists any document that is valid for G? (emptiness problem)
→ there is a document valid for G *and* valid for G2? (intersection & emptiness)

Classes XML Type Formalisms

local no competing TYPE names! (**DTDs**)

single-type TYPE names in the *same content model* do not compete! (**XML Schema's**)

regular no restriction… (**RELAX NG**)

Increasing Expressivness
of defining sets of trees ("tree languages")

Questions

Given two DTDs D1 and D2 can we check if
→ all documents valid for D1 are also valid for D2? (DTD inclusion problem)
→ D1 and D2 describe the same set of documents? (DTD equality problem)

Given a Relax NG grammar G, can we check if
→ there exists any document that is valid for G? (emptiness problem)
→ there is a document valid for G *and* valid for G2? (intersection & emptiness)

If we can do it for **regular tree** grammars, then also works for single-type/local!!

All of the checks can be done automatically, for **regular tree** grammars!

↑

equivalent to tree **automata**

Tree Automata: very powerful framework,

→ Have all the good properties of string automata!
→ Yet, they are more expressive!

Questions

Given two DTDs D1 and D2 can we check if
→ all documents valid for D1 are also valid for D2? (DTD inclusion problem)
→ D1 and D2 describe the same set of documents? (DTD equality problem)

Given a Relax NG grammar G, can we check if
→ there exists any document that is valid for G? (emptiness problem)
→ there is a document valid for G *and* valid for G2? (intersection & emptiness)

⇒ If we can do it for **regular tree** grammars, then also works for single-type/local!!

All of the checks can be done automatically, for **regular tree** grammars!

equivalent to tree **automata**

Tree Automata: very powerful framework,

→ Have all the good properties of string automata!
→ Yet, they are more expressive!

Note

String automata are **not** sufficient to check DTDs / Schemas!
Even if we only consider well-bracketed strings!

Example 1

c → c[a, c, b]
a → empty
b → empty
c → empty

Example 2

a → a[c, a]
a → a[a, b]
a / b / c → empty

All of the checks can be done automatically, for **regular tree** grammars!

constant memory computation

equivalent to tree **automata**

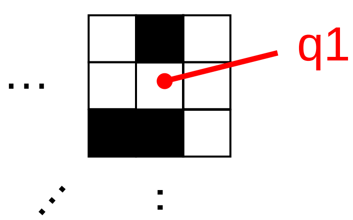
Finite-state automata are important:

→ Think you are in a maze, with only fixed memory and you can only read the maze (cannot mark anything).

wall

Model by finite automaton. In state q1, (to [N|S|E|W], ■) → (q2, [N|S|E|W])
q2, (to [N|S|E|W], □) → (q3, [N|S|E|W])

empty



Can an automaton search the maze?

All of the checks can be done automatically, for **regular tree** grammars!

equivalent to tree **automata**

constant memory computation

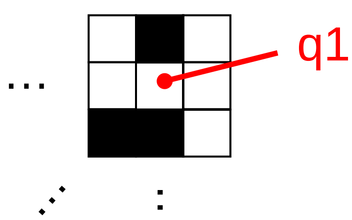
Finite-state automata are important:

→ Think you are in a maze, with only fixed memory and you can only read the maze (cannot mark anything).

wall

Model by finite automaton. In state q1, (to [N|S|E|W], ■) → (q2, [N|S|E|W])
q2, (to [N|S|E|W], □) → (q3, [N|S|E|W])

empty



Can an automaton search the maze?

No!! → need markers ("pebbles").
How many? 5? 2?

All of the checks can be done automatically, for **regular tree** grammars!

constant memory computation

equivalent to tree **automata**

Finite-state automata are important:

In our context, e.g., for

→ KMP (efficient string matching) [Knuth/Morris/Pratt] generalization using automata. Used, e.g., in `grep`

→ Compression

→ Static analysis of schemas & queries (= “everything you can do *before* *before* running over the actual data”)

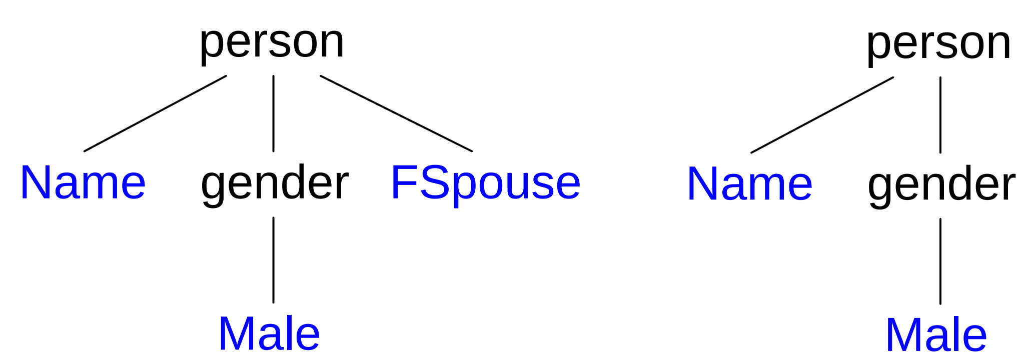
4. Static Methods, based on Tree Automata

Person → MPerson | FPerson
MPerson → person [Name, gender[Male], FSpouse?]
FPerson → person [Name, gender[Female], MSpouse?]

Regular Tree Grammar

Leaves may be labeled by TypeNames

Rules of the form TypeName → Tree



Alternatively, regular tree languages are defined by **Tree Automata**.

state, element-name → state1, state2

conventionally, defined for *binary/ranked* trees.

4. Static Methods, based on Tree Automata

Given grammars D1 and D2 can we check if
→ all documents valid for D1 are also valid for D2? (inclusion problem)
→ D1 and D2 describe the same set of documents? (equality problem)
→ does there exists any document that is valid for D1? (emptiness problem)
→ there is a document valid for D1 *and* valid for D2? (intersection & emptiness)

ALL these checks are possible for ***regular tree grammars****!!*

➔ hence, they are also solvable for DTDs / XML Schemas / RELAX NG's

(1) use binary tree encodings
(2) translate XML Type Definition to a Tree Grammar (easy)

Alternatively, regular tree languages are defined by **Tree Automata**.

state, element-name → state1, state2 ← conventionally, defined for *binary/ranked* trees.

4. Static Methods, based on Tree Automata

Given grammars D1 and D2 can we check if
→ all documents valid for D1 are also valid for D2? (*inclusion problem*)
→ D1 and D2 describe the same set of documents? (equality problem)
→ does there exists any document that is valid for D1? (emptiness problem)
→ there is a document valid for D1 *and* valid for D2? (intersection & emptiness)

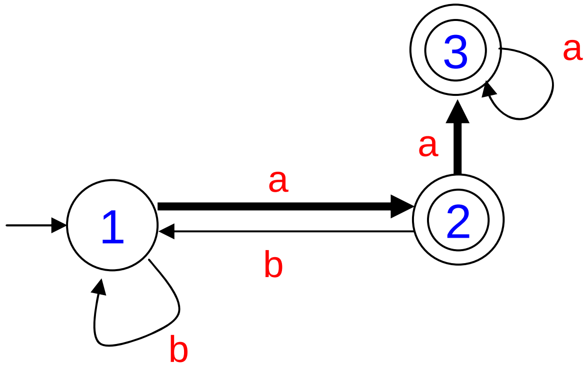
ALL these checks are possible for ***regular tree grammars****!!*

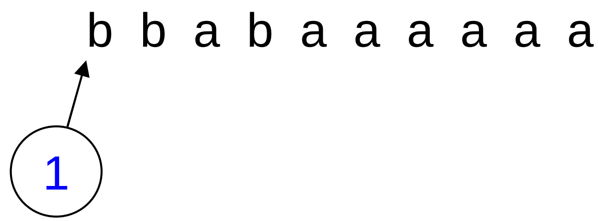
➔ The checks above give rise to
very powerful optimization procedures for XML Databases!

For example:
documents d_1, d_2, …, d_n are valid for your schema “Small_xhtml”.

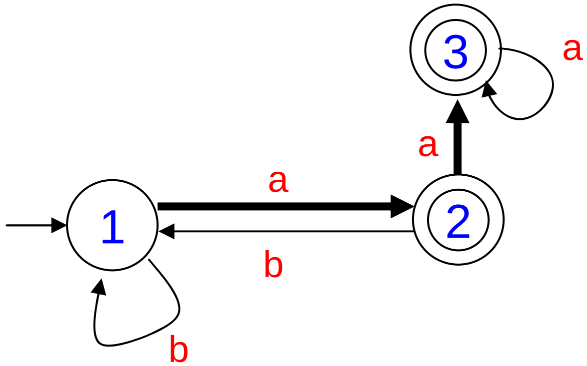
Are they also valid for schema XHTML?

➔ Check *inclusion problem* for Small_html and XHTML!

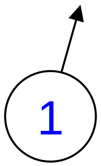

3
a
a
a
1
2
b
b


b b a b a a a a a a
1

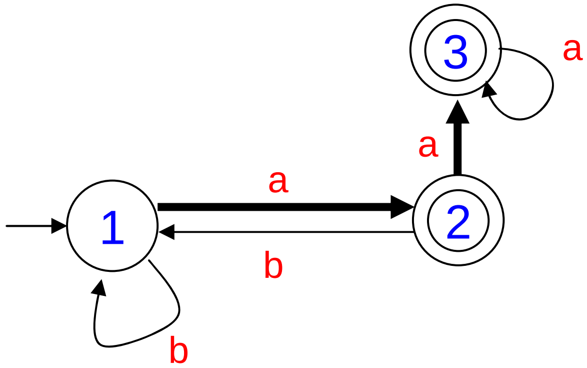
Automata on words

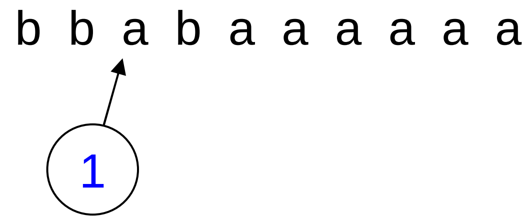


b b a b a a a a a a

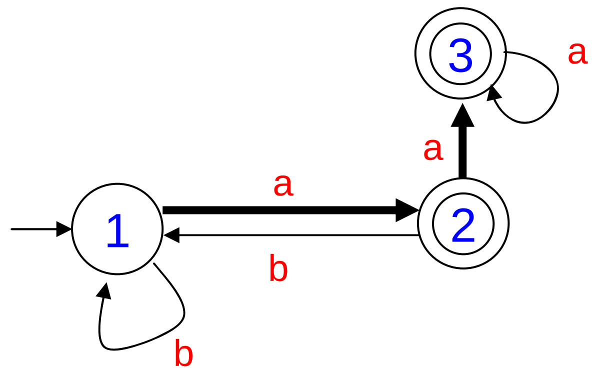


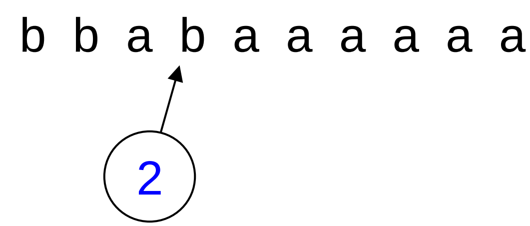
Automata on words



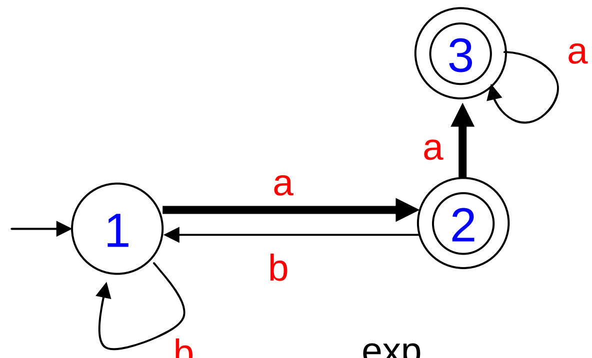


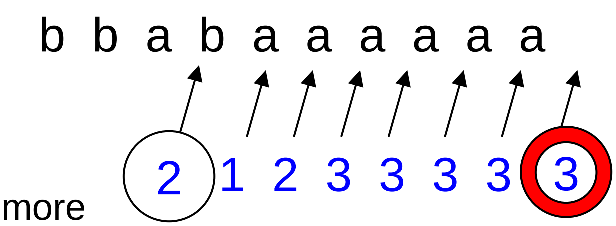
Automata on words





Automata on words





word is accepted by the automaton

Expressiveness

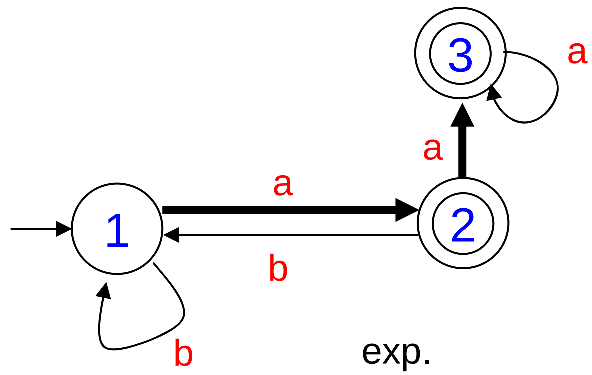
exp. blow-up

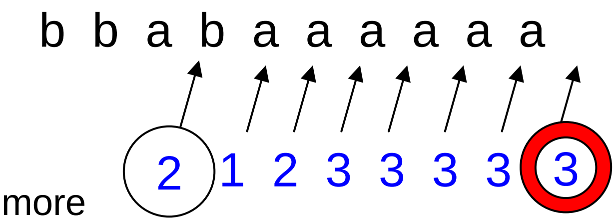
exp. more succinct

deteterministic = nondeterministic

left-to-right = right-to-left

Automata on words





word is accepted
by the automaton

exp. blow-up

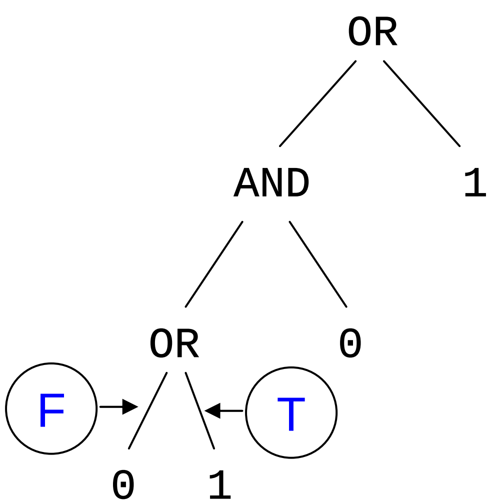
exp. more succinct

Expressiveness deteterministic = nondeterministic

left-to-right = right-to-left

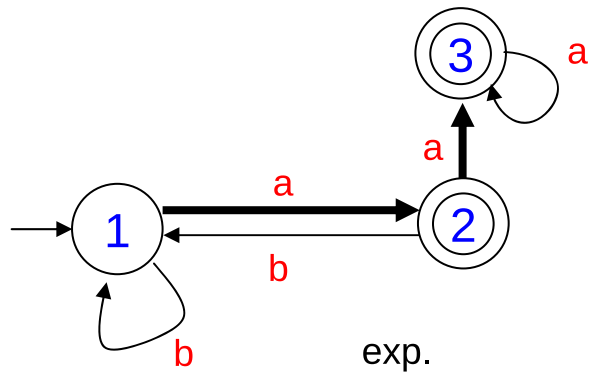
Automata on trees

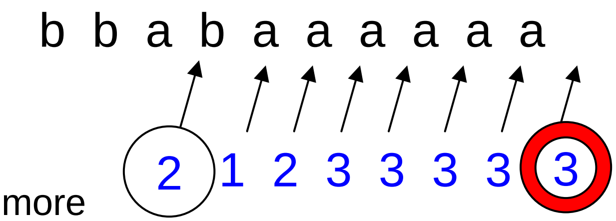
1. *bottom-up* LABEL(state1, state2) → state



0() → F
1() → T
OR(F, F) → F
OR(F, T) → T
...

Automata on words





word is accepted by the automaton

exp. blow-up

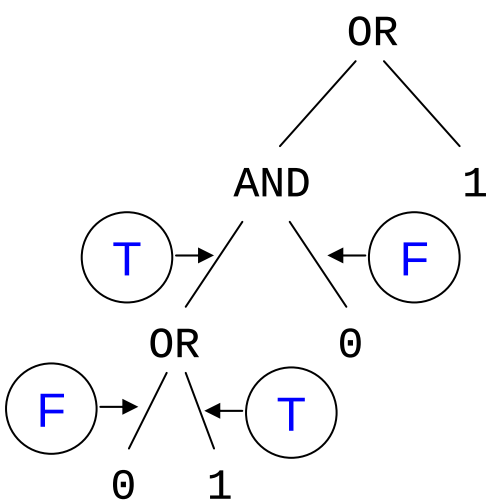
exp. more succinct

Expressiveness deteterministic = nondeterministic

left-to-right = right-to-left

Automata on trees

1. *bottom-up* LABEL(state1, state2) → state



0() → F

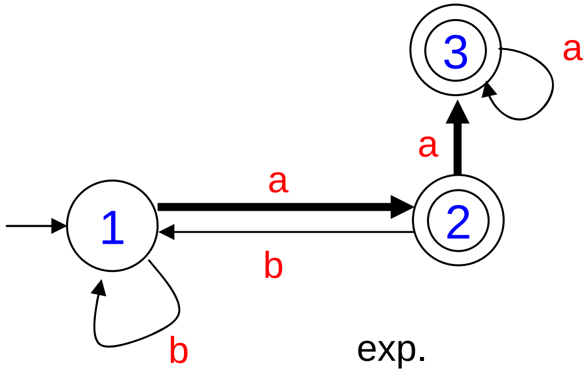
1() → T

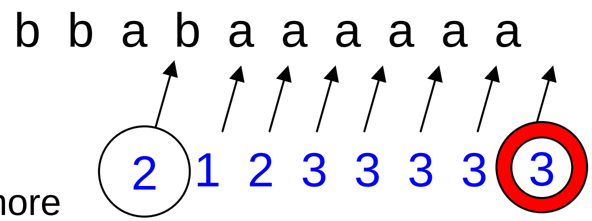
OR(F, F) → F

OR(F, T) → T

...

Automata on words





exp. blow-up

exp. more succinct

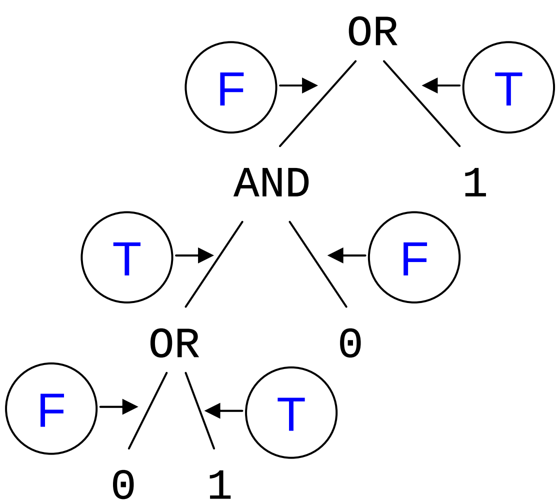
Expressiveness deteterministic = nondeterministic

left-to-right = right-to-left

word is accepted by the automaton

Automata on trees

1. *bottom-up* LABEL(state1, state2) → state



0() → F

1() → T

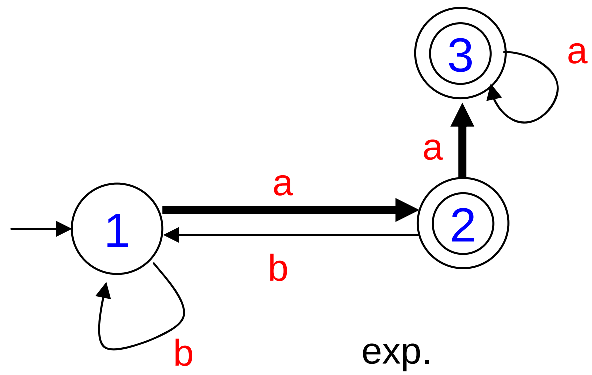
OR(F, F) → F

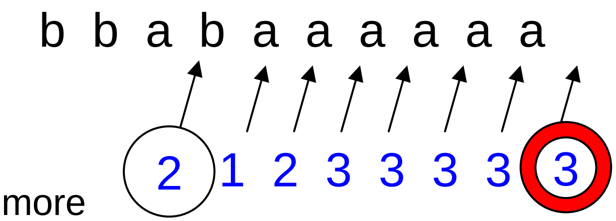
OR(F, T) → T

...

AND(T, F) → F

Automata on words





word is accepted by the automaton

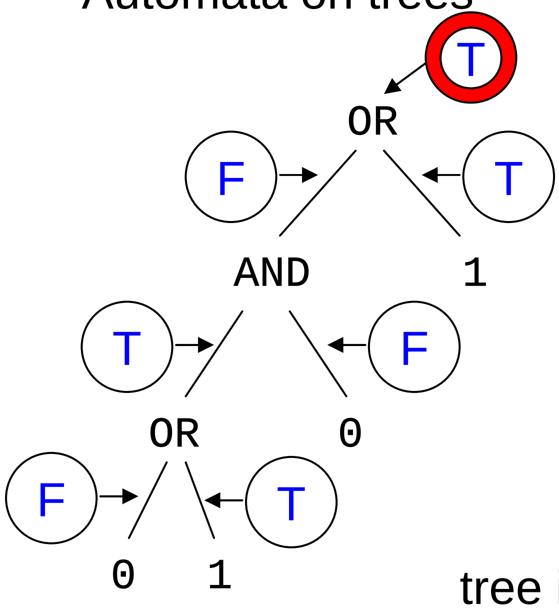
exp. blow-up

exp. more succinct

Expressiveness deteterministic = nondeterministic

left-to-right = right-to-left

Automata on trees



1. *bottom-up* LABEL(state1, state2) → state

0() → F

1() → T

OR(F, F) → F

OR(F, T) → T

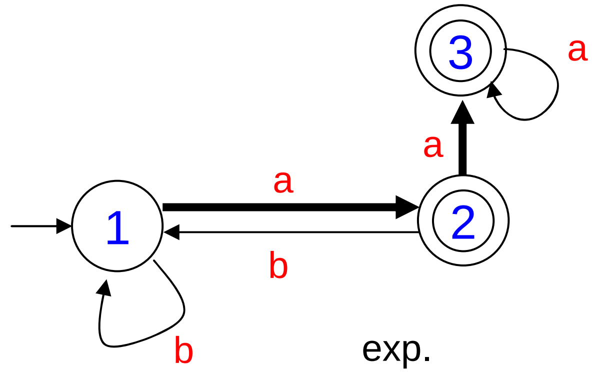
...

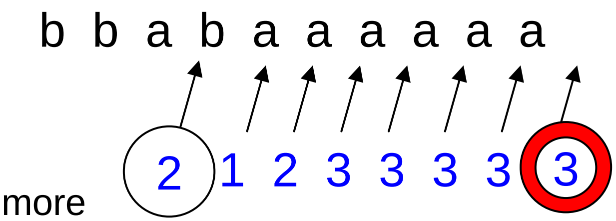
AND(T, F) → F

Accepting States = { T }

tree is accepted by the automaton

Automata on words





word is accepted by the automaton

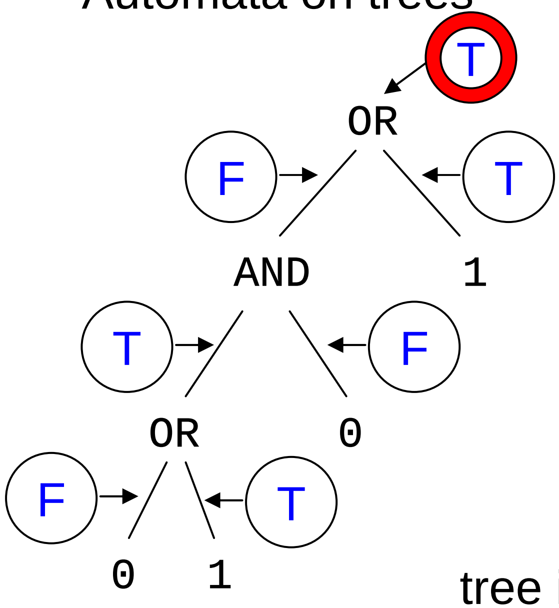
exp. blow-up

exp. more succinct

Expressiveness deteterministic = nondeterministic

left-to-right = right-to-left

Automata on trees



1. *bottom-up* LABEL(state1, state2) → state

0() → F
1() → T
OR(F, F) → F
OR(F, T) → T
...
AND(T, F) → F

Accepting States = { T }

This automaton is *deterministic*.

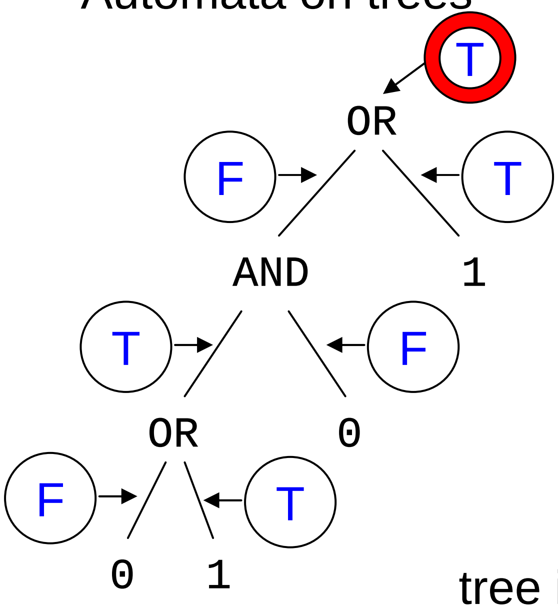
nondeterminism
LABEL(st1, st2) → { st3, … }

tree is accepted by the automaton

Question

How much memory do you need exactly, to run such a bottom-up tree automaton?

Automata on trees



1. *bottom-up* LABEL(state1, state2) → state

0() → F
1() → T
OR(F, F) → F
OR(F, T) → T
...
AND(T, F) → F

Accepting States = { T }

This automaton is *deterministic*.

nondeterminism
LABEL(st1, st2) → { st3, … }

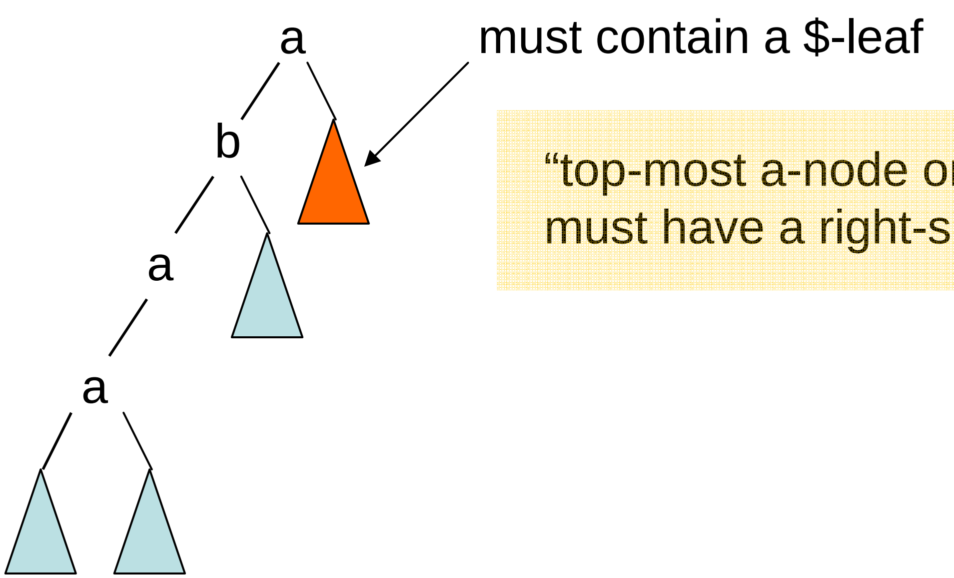
tree is accepted by the automaton

Similarly as for word automata:

> For every nondeterministic bottom-up tree automaton
> there is an equivalent deterministic bottom-up tree automaton.

Again, the construction can cause exponential size blow-up.

2. *top-down* state, LABEL → (state1, state2)



a,b = binary node labels
e,$ = leaf node labels

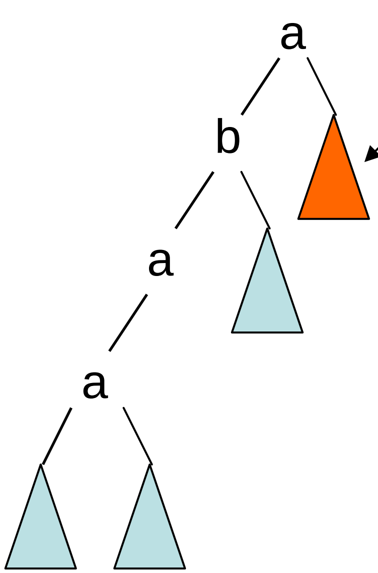
"top-most a-node on the left-most path
must have a right-subtree which contains a $-node."

Similarly as for word automata:

> For every nondeterministic bottom-up tree automaton
> there is an equivalent deterministic bottom-up tree automaton.

Again, the construction can cause exponential size blow-up.

2. *top-down* state, LABEL → (state1, state2)



must contain a $-leaf

a,b = binary node labels
e,$ = leaf node labels

"top-most a-node on the left-most path
must have a right-subtree which contains a $-node."

begin, a	→	(any, find$)
begin, b	→	(begin, any)
find$, a/b	→	{ (find$, any), (any, find$) }
find$, $	→	ACC
find$, e	→	REJ
any, a/b	→	(any,any)
any, $/e	→	ACC

nondeterministic

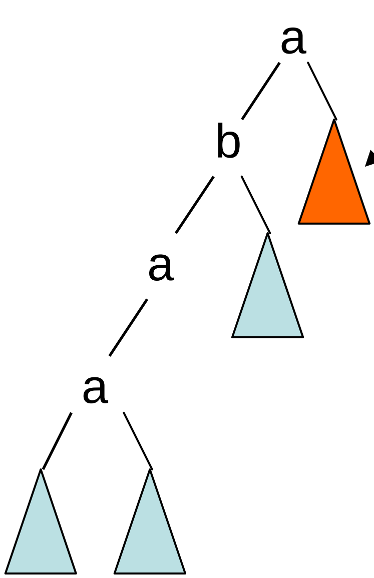
→ accept tree if there exists an accepting run
(= all leaves go to ACC)

For every nondeterministic bottom-up tree automaton
there is an equivalent deterministic bottom-up tree automaton.

Question
Can you find an equivalent *bottom-up* automaton for this example?

---

2. *top-down* state, LABEL → (state1, state2)



must contain a $-leaf

a,b = binary node labels
e,$ = leaf node labels

“top-most a-node on the left-most path
must have a right-subtree which contains a $-node.”

begin, a → (any, find$)
begin, b → (begin, any)
find$, a/b → { (find$, any), (any, find$) }
find$, $ → ACC
find$, e → REJ
any, a/b → (any,any)
any, $/e → ACC

nondeterministic

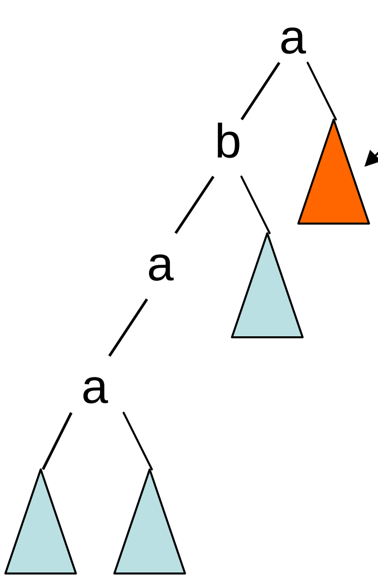
→ accept tree if there exists an accepting run

For every nondeterministic bottom-up tree automaton
→ there is an equivalent deterministic bottom-up tree automaton, and
→ there is an equivalent nondeterministic *top-down* tree automaton.

➔ Yes! you can… ☺

---

2. *top-down* state, LABEL → (state1, state2)



must contain a $-leaf

a,b = binary node labels
e,$ = leaf node labels

"top-most a-node on the left-most path
must have a right-subtree which contains a $-node."

begin, a → (any, find$)
begin, b → (begin, any)
find$, a/b → { (find$, any), (any, find$) }
find$, $ → ACC
find$, e → REJ
any, a/b → (any,any)
any, $/e → ACC

nondeterministic

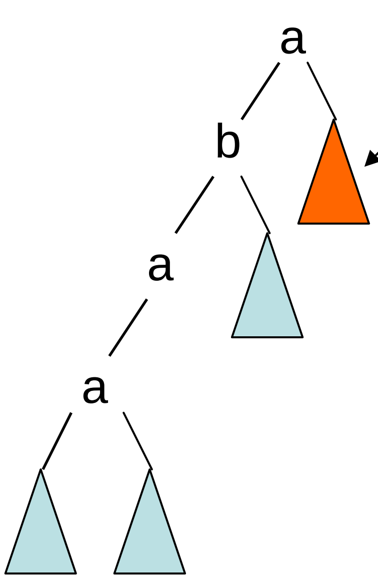
→ accept tree if there exists an accepting run

For every nondeterministic bottom-up tree automaton
→ there is an equivalent deterministic bottom-up tree automaton, and
→ there is an equivalent nondeterministic *top-down* tree automaton.

Question
Is there an equivalent deterministic top-down automaton??

---

2. *top-down* state, LABEL → (state1, state2)



must contain a $-leaf

a,b = binary node labels
e,$ = leaf node labels

"top-most a-node on the left-most path
must have a right-subtree which contains a $-node."

begin, a	→	(any, find$)
begin, b	→	(begin, any)
find$, a/b	→	{ (find$, any), (any, find$) }
find$, $	→	ACC
find$, e	→	REJ
any, a/b	→	(any,any)
any, $/e	→	ACC

nondeterministic

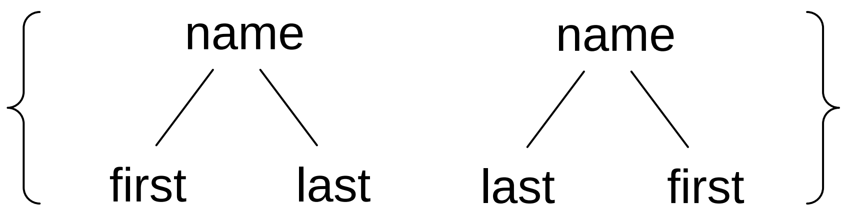
→ accept tree if there exists an accepting run

For every nondeterministic bottom-up tree automaton
→ there is an equivalent deterministic bottom-up tree automaton, and
→ there is an equivalent nondeterministic *top-down* tree automaton.

Question

Is there an equivalent deterministic top-down automaton??

➔ NO! ☹



This set of two trees canNOT be recognized
by any determinstic top-down tree automaton!!

Why?

For every nondeterministic bottom-up tree automaton
→ there is an equivalent deterministic bottom-up tree automaton, and
→ there is an equivalent nondeterministic *top-down* tree automaton.

Question

Is there an equivalent deterministic top-down automaton??

➔ NO! ☹

Questions

What about **local** tree languages (defined by DTDs).
→ Can they be accepted by deterministic top-down automata?

What about **single-type** tree languages (defined by XML Schema's)
→ Can they be accepted by deterministic top-down automata?

For every nondeterministic bottom-up tree automaton
→ there is an equivalent deterministic bottom-up tree automaton, and
→ there is an equivalent nondeterministic *top-down* tree automaton.

Question
Is there an equivalent deterministic top-down automaton??

→ NO! ☹

Questions

What about **local** tree languages (defined by DTDs).
→ Can they be accepted by deterministic top-down automata?

What about **single-type** tree languages (defined by XML Schema's)
→ Can they be accepted by deterministic top-down automata?

Yes!
Hence, there is **no DTD / Schema** for { name[first,last], name[last,first] }

For every deterministic bottom-up tree automaton
there exists a minimal unique equivalent one!

→ Equivalence is decidable

In fact, YOU have already produced
minimal bottom-up tree automata!

The minimal DAG of a tree t can be seen as the minimal unique
tree automaton that only accepts the tree t.

For every deterministic bottom-up tree automaton
there exists a minimal unique equivalent one!

→ Equivalence is decidable

In fact, YOU have already produced
minimal bottom-up tree automata!

The minimal DAG of a tree t can be seen as the minimal unique
tree automaton that only accepts the tree t.

Question

How expensive (complexity) to find mininmal one?

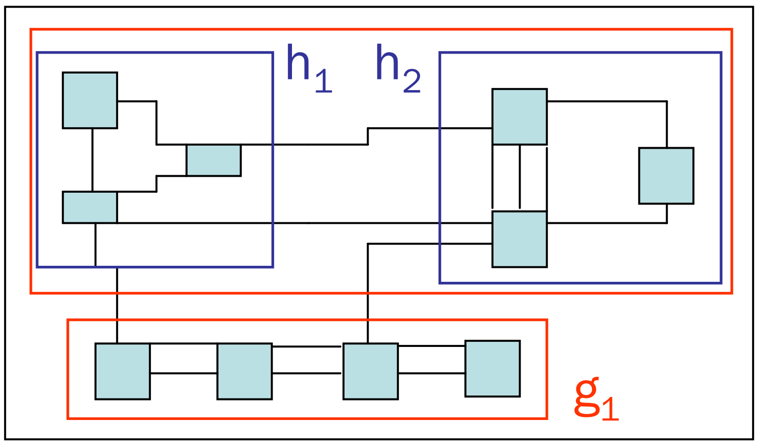
→Same as for word automata?

Tree Automata are a very useful concept in CS!

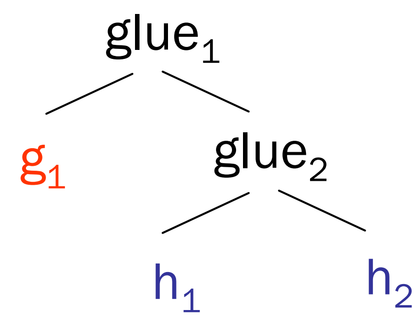
→ Heavily used in verification

“Derive a property of a complex object

from the properties of its constituents…”



$g = \text{glue}_1(\, g_1, \text{glue}_2(\, h_1, h_2\,)\,)$



➔ Do all graphs / chip-layouts produced in this way, have property P?

Use the hierarchical construction history of an object, in order to work on a “parse” tree instead of a complex graph.
From there, use tree automata. ☺

Many NP-complete graph problems become

tractable on “bounded-treewidth “ graphs!

XML Tree Automata play crucial rule for

➔ Efficient validators against XML Types

→ Optimizations If doc1 is of TYPE1, then no need to validate against TYPE2, if we know TYPE2 included in TYPE1

- if only “slightly different” then only need to validate “there”
- incremental validation against updates
- etc, etc.

➔ Efficient query evaluators, use richer automata which can select nodes and produce query answers

→ Optimizations If answer of QUERY1 is in cache, then no need to evaluate QUERY2, if “included” in QUERY1.

- if every possible answer set to QUERY1 (of TYPE X) is EMPTY, then no need to evaluate on the real data!

➔ XML Type Checking for Programming Languages

The Future

In 5-10 years from now: ☺

You can write a function in Programming Language X

```
Function foo(XML document D: TYPE1):  TYPE2
{
    traverse D
        & compute output;
    .
    .
    .
    return output
}
```

Compiler (XML Type Checker) will complain, if your function does not compute documents of TYPE2.

➔ If no complaint, then **guaranteed**:
ALL outputs are ALWAYS of correct type!!)

The Future

Experimental PL's
In this direction:
→CDuce
→XDuce

In 5-10 years from now: ☺

You can write a function in Programming Language *X*

```
Function foo(XML document D: TYPE1):  TYPE2
{
    traverse D
        & compute output;
    .
    .
    .
    return output
}
```

Compiler (XML Type Checker) will complain, if your function does not compute documents of TYPE2.

➔ If no complaint, then correct type **guaranteed**.

Compilers will **have** to be able to give *static guarantees* about input/output behaviour of program!

END
Lecture 5