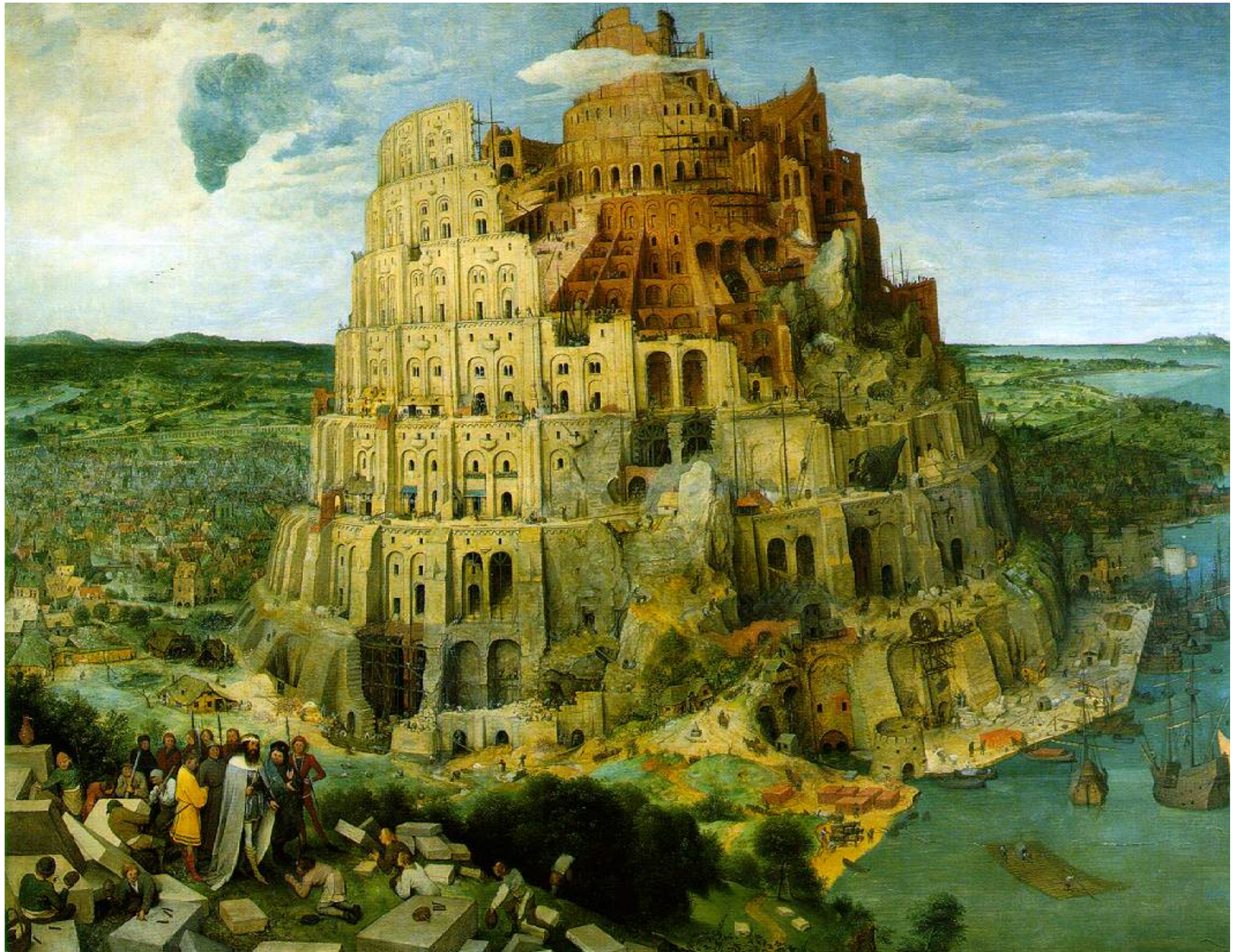


# Foundations of XML Processing

Sebastian Maneth  
EPFL

# Outline

- Motivation of XML
- Data Models
- Type Formalisms
  
- XML Processing
  - Type Checking
  - Implementation
  
- Future Research
- Conclusion



# Motivation

→ XML is a **Data Exchange Format**

- 1974 SGML (Charles Goldfarb at IBM Research)
- 1989 HTML (Tim Berners-Lee at CERN/Geneva)
- 1994 Berners-Lee founds Web Consortium (W3C)
- 1996 **XML** (W3C draft, v1.0 in 1998)

# XML = data

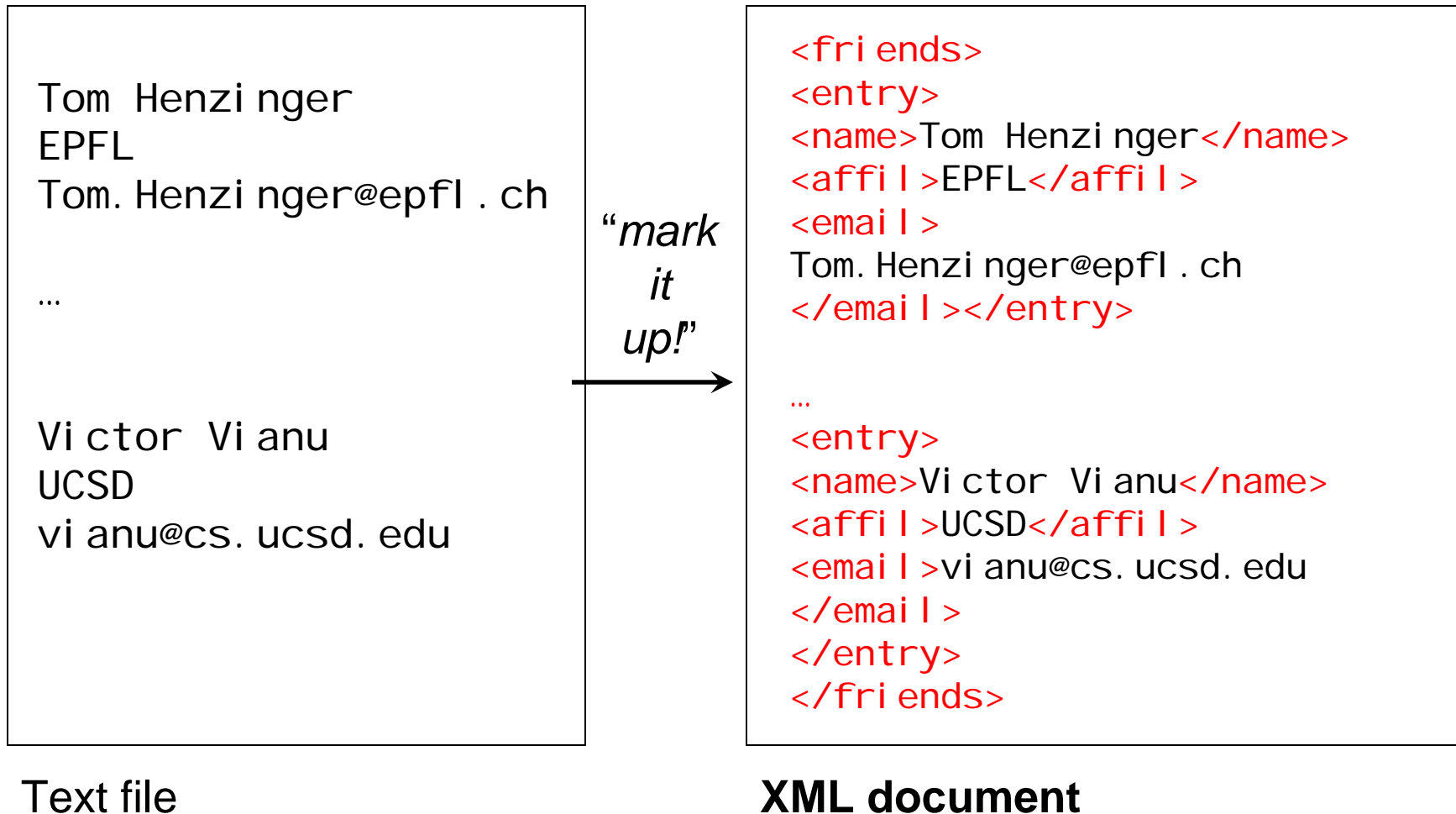
```
Tom Henzinger  
EPFL  
Tom.Henzinger@epfl.ch
```

...

```
Victor Vianu  
UCSD  
vianu@cs.ucsd.edu
```

Text file

# XML = data + structure



# XML Documents

- Ordinary text files (UTF-8, UTF-16, UCS-4 ...)
- Idea of labeled brackets for structure is not new!  
(already used by Chomsky in the 1960's)
- Brackets describe a tree structure
- **Allows applications from different vendors to exchange data!**
- **standardized, extremely widely accepted!**

# XML has many friends:

## Query Languages

Xpath, XSLT, Xquery, fxt, ... (mostly by W3C)

## Implementations (Parsers, Validators, Translators)

SAX, Xalan, Galax, Xerxes, ...

(by IBM/Apache, Microsoft, Oracle, Sun...)

---

## Current Issues

- DB/PL support (“data binding”, JBind, Castor, Zeus...)
- storage support (compression, data optimization)

# Data Models

**Streaming** (memory-less)

```
<friends><name>Tom Henzinger</name><affiliation> ... </friends>
```

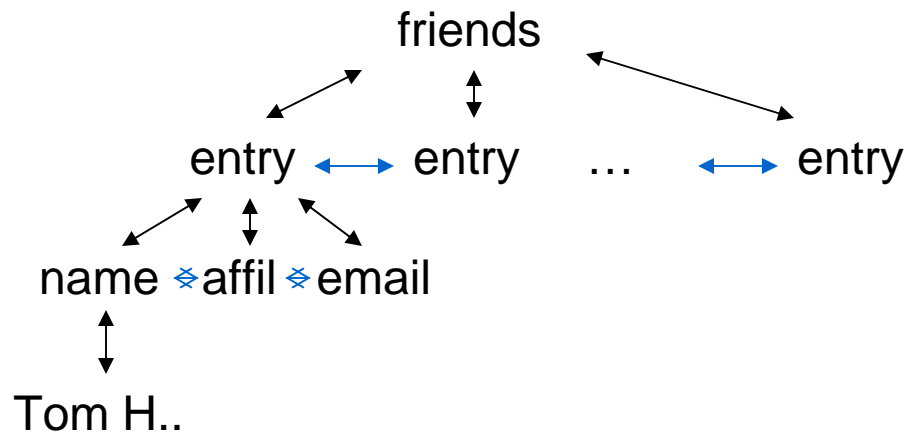
# Data Models

## Streaming (memory-less)

<friends><name>Tom Henzinger</name><affiliation> ... </friends>

## DOM Tree

(unranked,  
ordered)

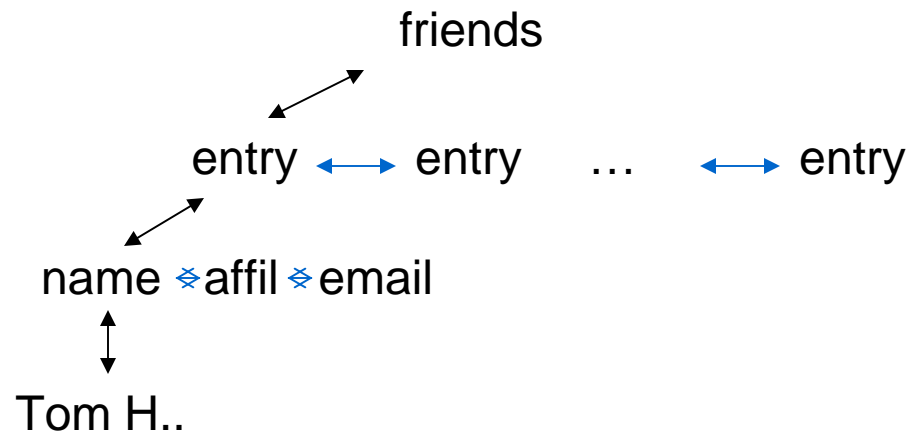


# Data Models

## Streaming (memory-less)

<friends><name>Tom Henzinger</name><affiliation> ... </friends>

## Binary Tree

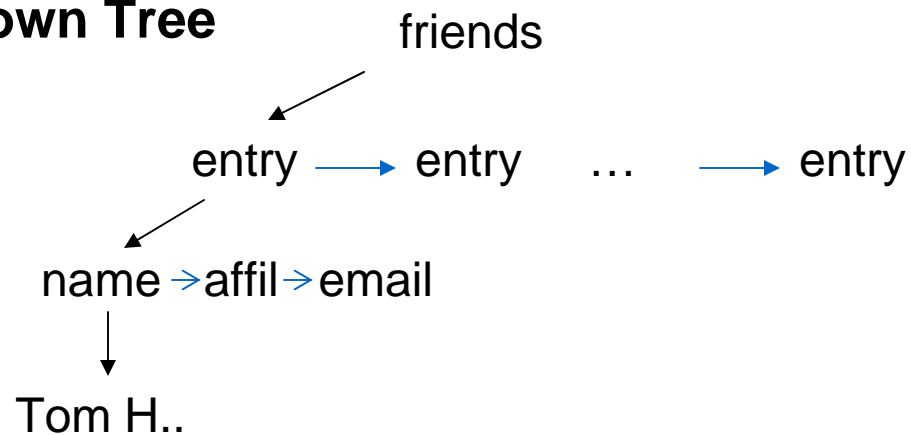


# Data Models

## Streaming (memory-less)

<friends><name>Tom Henzinger</name><affiliation> ... </friends>

## Binary Top-Down Tree

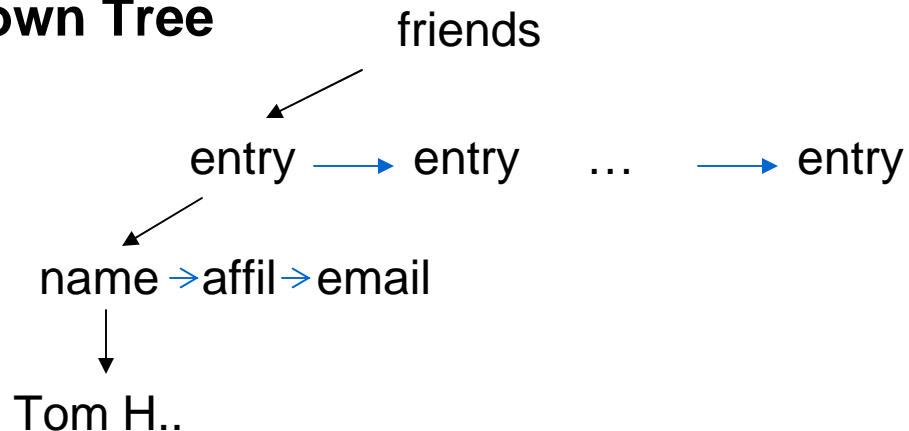


# Data Models

## Streaming (memory-less)

<friends><name>Tom Henzinger</name><affiliation> ... </friends>

## Binary Top-Down Tree



## Relational

|                     |      |       |     |        |
|---------------------|------|-------|-----|--------|
| friends/entry/name  | Tom  | Jerry | ... | Victor |
| friends/entry/affil | EPFL | MIT   |     | USCD   |
| friends/entry/email | ...  |       |     |        |



# Type Formalisms

Document Type Definition: (DTD)      TYPE Friends = ELEMENT friends(Entries)  
TYPE Entries = (ELEMENT entry(Entry))\*  
TYPE Entry = (ELEMENT name(CDATA))  
(ELEMENT affil(CDATA))  
(ELEMENT email(CDATA))

DTD's originate from SGML

regexp's over tag(T)

Newer:      XML Schema      (W3C)  
RELAX NG      (Oasis)

---

In terms of tree languages, all these formalisms  
are included in the

**→ regular tree languages (REGT).**

# Type Formalisms

**Regular Tree languages (REGT).**

Many characterizations: *Reg. Tree Grammars*  
*Tree Automata*  
*MSO Logic*

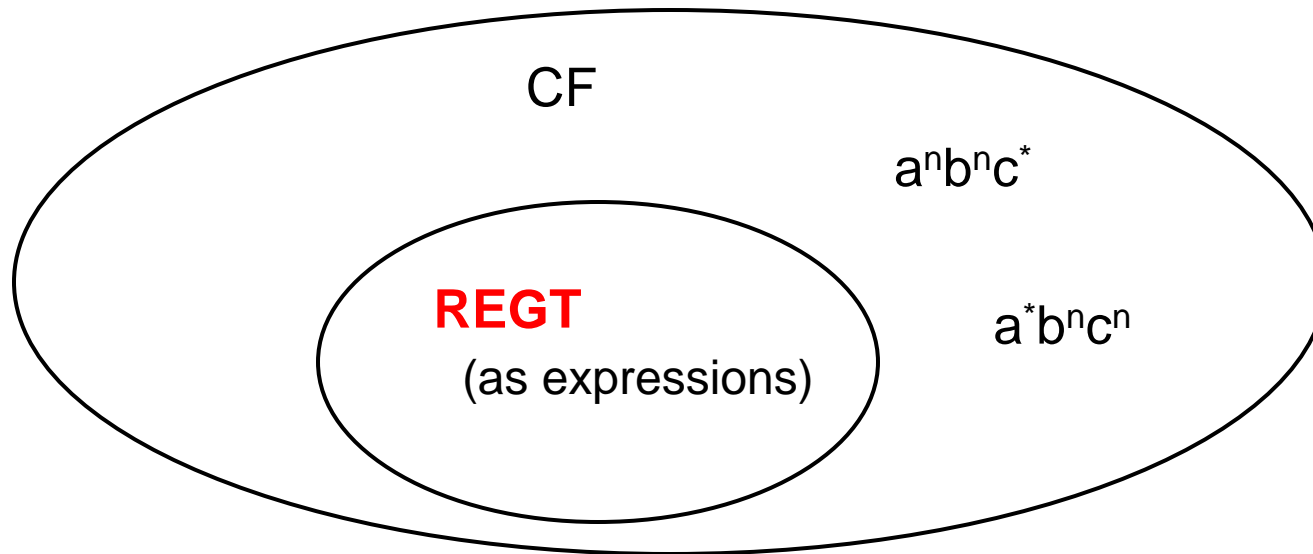
Nice properties: *Closed under intersection (union, complement)*  
*Decidable equivalence*

# Type Formalisms

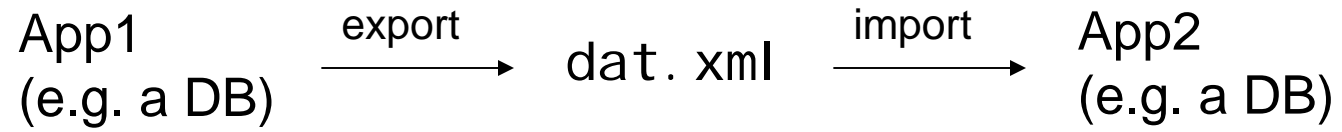
**Regular Tree languages (REGT).**

Many characterizations: *Reg. Tree Grammars*  
*Tree Automata*  
*MSO Logic*

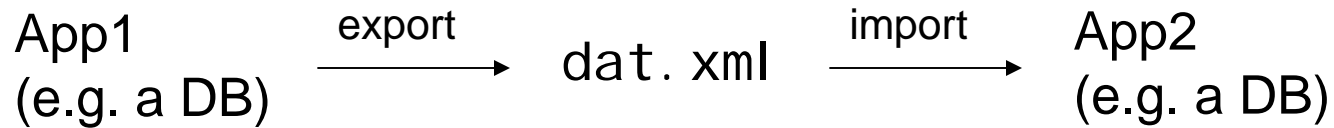
Nice properties: *Closed under intersection (union, complement)*  
*Decidable equivalence*



# XML Processing

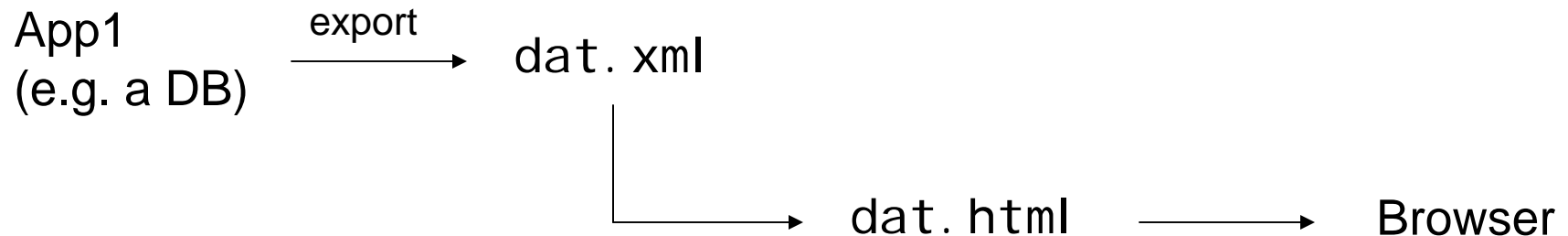


# XML Processing

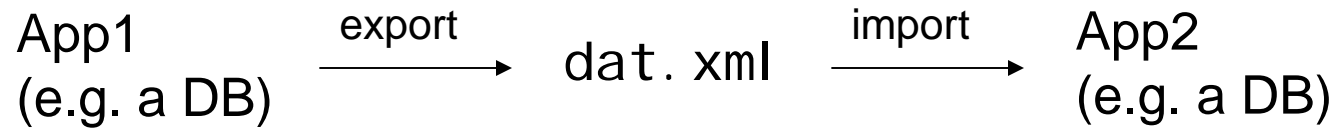


What if App2 expects a different type of XML documents?  
Or, if it has no XML-import functionality?

E.g. App2 is a web browser that expects HTML.



# XML Processing



What if App2 expects a different type of XML documents?  
Or, if it has no XML-import functionality?

E.g. App2 is a web browser that expects HTML.



How to specify this?  
What are formal models?

# XML Processing

Given a language that allows to specify a XML-to-XML (or HTML) transformation, you would like to

- (1) **Type check the transformation**
- (2) **Implement it efficiently**

If the transformation type checks, then it will **ONLY** generate XML of the correct output type for all possible inputs (of the correct type).  
(e.g. HTML)

# XML Processing

Given a language that allows to specify a XML-to-XML transformation, you would like to

- (1) **Type check the transformation**
- (2) **Implement it efficiently**

If the transformation type checks, then it will **ONLY** generate XML of the correct output type for all possible inputs (of the correct type).  
(e.g. HTML)

---

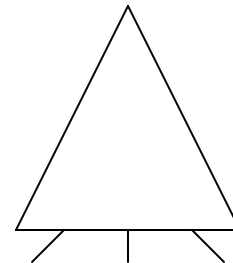
Clearly, solutions to (1) and (2) heavily depend on the specification language and its expressive power!

# XML Processing

How to specify XML transformations?

- (1) select nodes
- (2) construct output and process recursively

<select pattern> →



apply patterns

**Pattern Matching**

---

In Query/Transformation languages:

For (1), use XPath 2.0 [W3C], e.g., `//entry/name[text="Victor Vianu"]`

In Programming Languages, use regexp patterns (Xduce, Xtatic, Scala), or reg. tree languages (fxt).

# XML Processing

E.g., in XSLT [W3C], this is a possible rule:

```
//x@entry/name[text="Victor Vianu"] → <results>  
                                     <apply x>  
                                     </results>
```

# XML Processing

## The XML Type Checking Problem:

Input: XML input and output types  $T_{in}$ ,  $T_{out}$  (in REGT)  
An XML transformation  $t$

Question: Is  $t(d) \in T_{out}$  for all  $d \in T_{in}$ ?

---

# XML Processing

## The XML Type Checking Problem:

Input: XML input and output types  $T_{in}$ ,  $T_{out}$  (in REGT)  
An XML transformation  $t$

Question: Is  $t(d) \in T_{out}$  for all  $d \in T_{in}$ ?

---

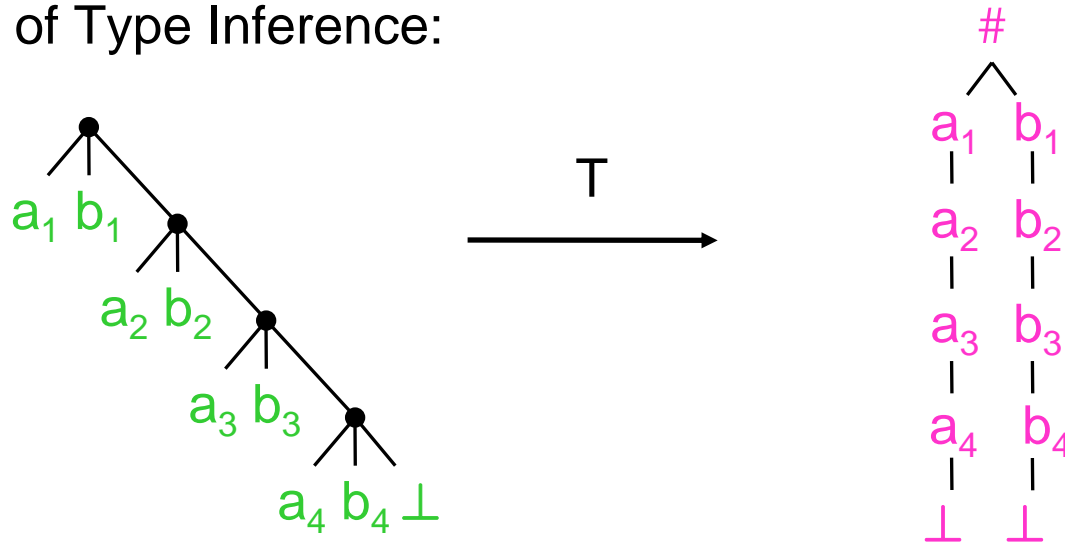
Usually type checking is solved by **type inference**:

1. Infer the type  $K$  of  $t(T_{in})$
2. Check if  $K \subseteq T_{out}$

→ Remember that XML types are in REGT!

# XML Type Checking

Limitations of Type Inference:



XQuery infers the output type:  $K = \#(a_i^* \perp, b_i^* \perp)$

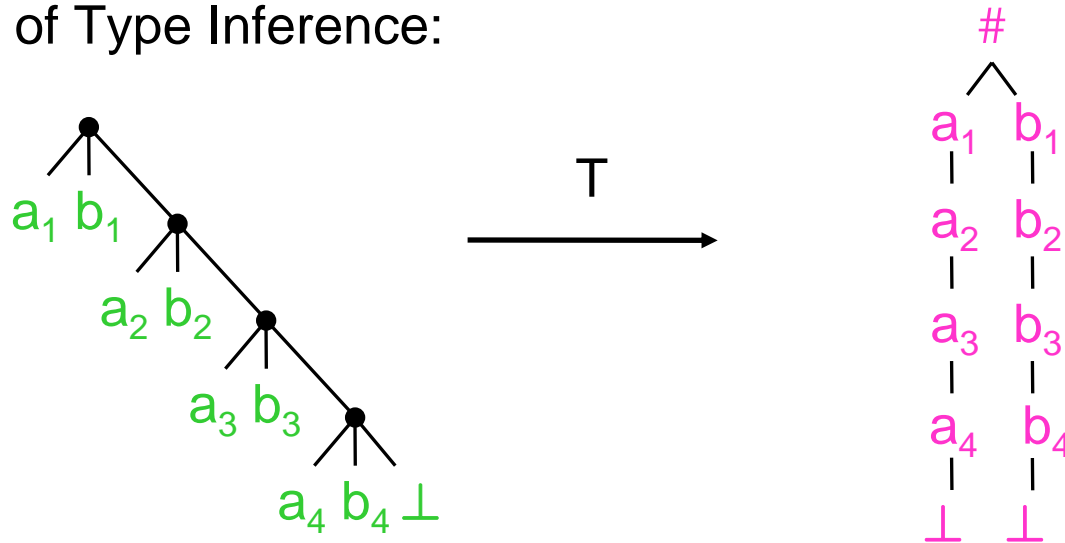
---

User needs XML docs of type  $L = \#(\perp, \perp) \mid \#(a_i a_i^*, b_i b_i^*)$

Clearly, outputs of  $T$  are in  $L$ . (“ $T$  type checks wrt  $L$ ”)

# XML Type Checking

Limitations of Type Inference:



XQuery infers the output type:  $K = \#(a_i^* \perp, b_i^* \perp)$

---

User needs XML docs of type  $L = \#(\perp, \perp) \mid \#(a_i a_i^*, b_i b_i^*)$

Clearly, outputs of  $T$  are in  $L$ . (“ $T$  type checks wrt  $L$ ”)

**BUT type-checker rejects, because  $K \not\subseteq L$  !!!**

# XML Type Checking

Outputs of transformations are in general NOT in REGT.  
→ type can only be approximated.

---

[M.BerleaPerstSeidl,PODS'05]

For the XML transformation language “TL”, **exact type checking** can be performed **effectively**.

TL can simulate most XML query and transformation languages.

---

Earlier results:

[MiloSuciuVianu,PODS'00]: “all XML query and transf. languages can be simulated by pebble tree transducers (PTTs)”

[EngelfrietM.'03]: upper bound to type check k-PTTs: non-elementary!

# XML Type Checking

Idea of the type checking algorithm:

- (1) Compile the transformation in a composition  $M1; M2; M3$  of Macro Tree Transducers (MTTs)
- (2) Determine the REGT  $W = (M1; M2; M3)^{-1}(\overline{T_{out}})$
- (3) Test whether  $(T_{in} \cap W) = \emptyset$ .

---

Based on this result we are currently implementing an exact static type checker for fxt, an XML transformation system developed at TU Munich, Germany.

# Implementation

→ Pattern matching constructs are compiled into tree automata.

CAVEAT:

Single match vs. all match semantics.



Can cause ambiguities.

pat1 → action1  
pat2 → action2  
pat3 → action3

**Challenge:** determine match (+var bindings)  
of pattern with highest priority, with only  
TWO runs through the input tree.

Even all possible matches can be determined by only two traversals through the input, using intricate automata constructions.

# Implementation

→ Pattern matching constructs are compiled into tree automata.

We are implementing these ideas into the pattern matcher of the programming language Scala.

Currently: regexp patterns with single match policy (right-longest)

Scala is developed by Martin Odersky and his PL group at EPFL.  
It is aimed towards component abstraction and web programming,  
and compiles to JVM or .NET  
It smoothly fuses FP and OOP.

# Implementation

## **Problem:**

Loading an XML file into memory needs HUGE amount of memory!!

In many Java-based implementations, loading a file takes up to 10-times more memory than size of the file!!!!

# Implementation

For XML support to be **practical** it must support

- representing large XML files in memory  
(>50MB, sometimes as large as 500MB!)
- efficient querying/processing of representations

Approaches to handle large XML files:

- paging techniques
- compression of XML

# Data Optimization

- Even in clever implementations, DOM trees usually take 4-5 times more memory than size of XML file!!

(why? Imagine `<a/>`, which needs 4 bytes, as compared to a tree node, which needs at least 16 bytes.)

- 
- **Idea** Cut off data values and put them in a table
  - Do hash consing (sharing of common subtrees) on tree skeleton.

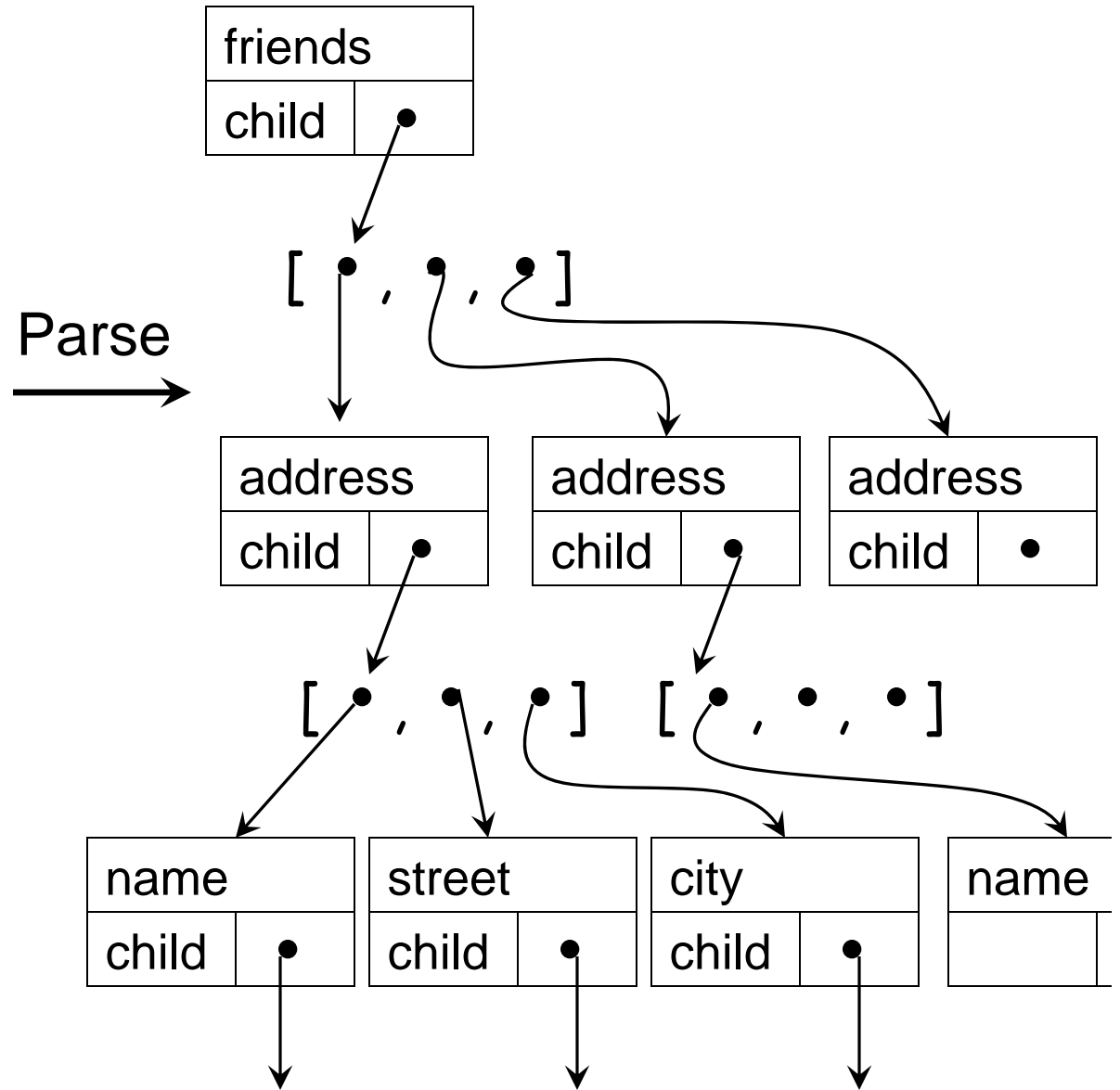
→ for common XML files, tree skeletons compress to  
≈ **10% of original size!** [[Buneman/Grohe/Koch, VLDB'03](#)]:

# Data Optimization

```

<friends>
<address>
<name> Giorgio Busatto </name>
<street> Agnes-Miegel Str.
33 </street>
<city> 28279 Bremen </city>
</address> .....
<address>
<name> Agnes Meier </name>
</friends>
    
```

XML file of addresses

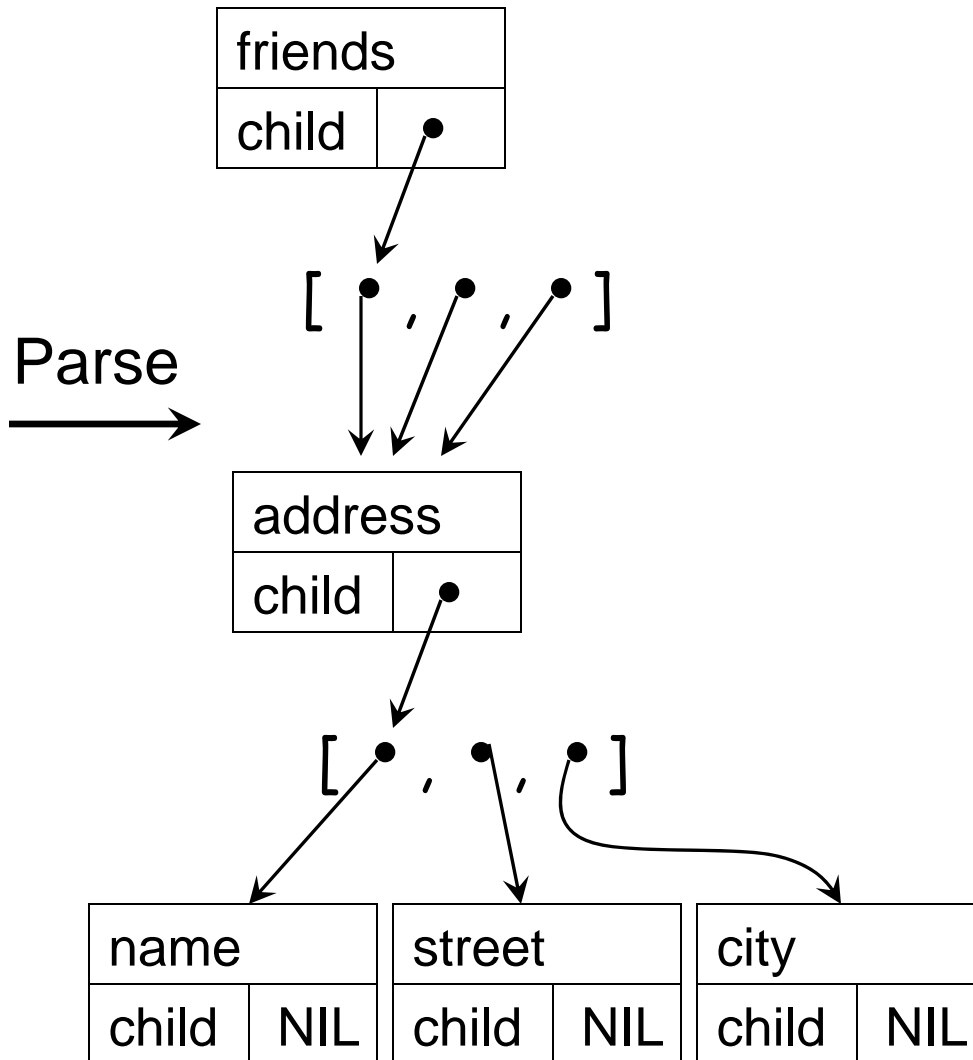


# Data Optimization

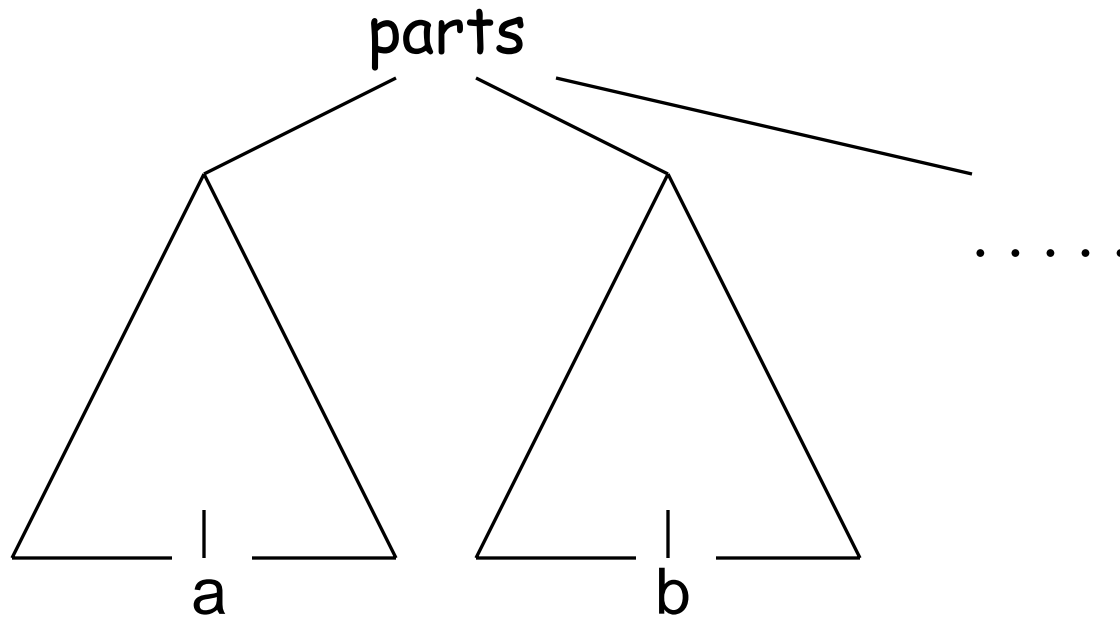
```

<friends>
<address>
<name> Giorgio Busatto </name>
<street> Agnes-Miegel Str.
33 </street>
<city> 28279 Bremen </city>
</address> .....
<address>
<name> Agnes Meier </name>
</friends>
    
```

XML file of addresses



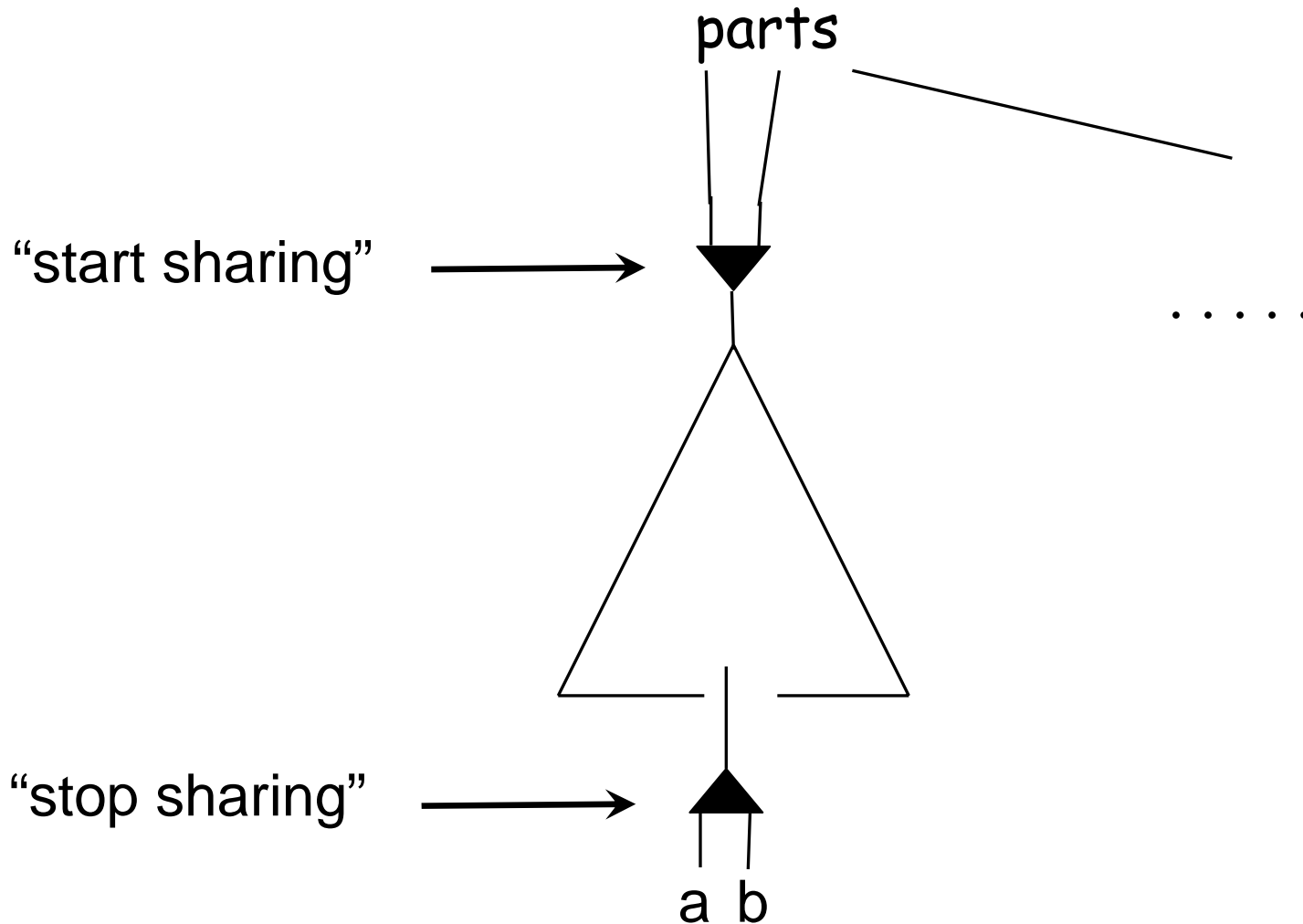
# Data Optimization: one step further



Can NOT be shared, different in one node.

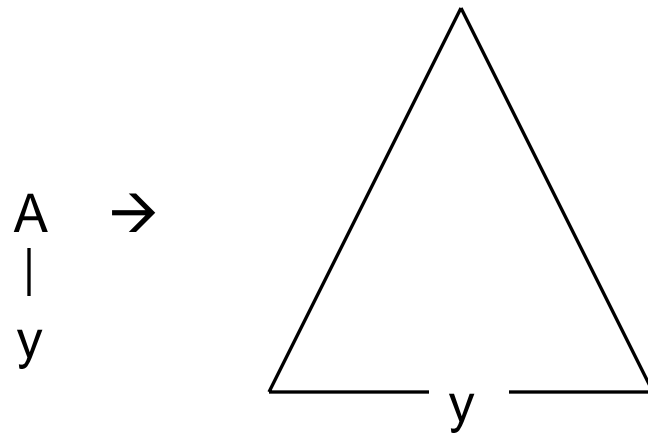
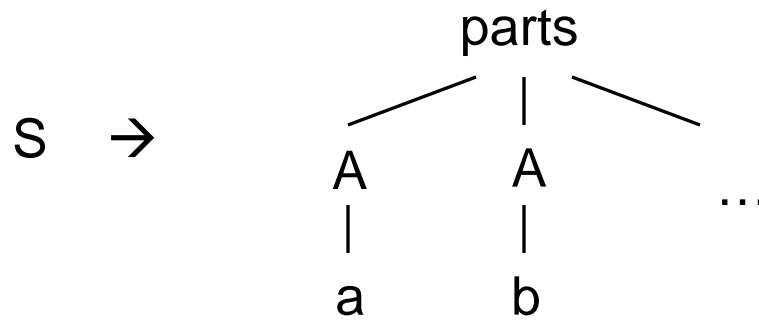
# Data Optimization: one step further

Idea: use Lamping's **sharing graphs**



# Data Optimization: one step further

A sharing graph can be seen as a  
“straight-line context-free tree grammar”.



# Data Optimization: one step further

Given a tree  $t$ , its **minimal DAG**

- is unique
- can be computed in linear time (at parse time!).

(folklore, known from 60's LISP times. “hash consing”)

---

Its **minimal Sharing Graph** is

- not unique
- finding one is NP-complete [[BusattoM.04](#)]

(because finding a minimal context-free grammar for a given string is NP-complete [[LehmanShelat,SODA'02](#)])

# Data Optimization: one step further

In [\[BusattoLohreyM.04\]](#) we describe BPLEX, a linear-time approx. algorithm for finding a small sharing graph for a given tree.

- Similar to LZ compression
- use scanning window to search for tree patterns (up to a certain size)

As far as we know, BPLEX generates the most efficient pointer based tree representation currently available.

# Data Optimization: one step further

| input file              | size of tree | min. binary DAG size |       | min. unranked mDAG size |       | BPLEX output size |       |
|-------------------------|--------------|----------------------|-------|-------------------------|-------|-------------------|-------|
|                         |              |                      |       |                         |       |                   |       |
| SwissProt (457,4 MB)    | 10,903,568   | 1,437,445            | 13.2% | 1,100,648               | 10.1% | 311,328           | 2.9%  |
| DBLP (103.6 MB)         | 2,611,931    | 533,183              | 20.4% | 222,754                 | 8.5%  | 115,902           | 4.4%  |
| Trebank (55.8 MB)       | 2,447,727    | 1,454,494            | 59.4% | 1,301,688               | 53.2% | 519,542           | 21.2% |
| 1998statistics (657 KB) | 28,306       | 2,403                | 8.5%  | 726                     | 2.6%  | 410               | 1.4%  |
| catalog-02 (104M)       | 2,240,231    | 52,392               | 2.3%  | 32,267                  | 1.4%  | 26,774            | 1.2%  |
| catalog-01 (11M)        | 225,194      | 6,990                | 3.1%  | 8,503                   | 2.8%  | 3,817             | 1.7%  |
| dictionary-02 (104M)    | 2,731,764    | 681,130              | 24.9% | 441,322                 | 16.2% | 160,329           | 5.9%  |
| dictionary-01 (11M)     | 277,072      | 77,554               | 28.0% | 46,993                  | 17.0% | 20,150            | 7.3%  |
| JST_snp.chr1 (36M)      | 655,946      | 40,663               | 6.2%  | 25,047                  | 2.3%  | 12,858            | 1.8%  |
| JST_gene.chr1 (11M)     | 216,401      | 14,606               | 6.7%  | 5,658                   | 2.6%  | 4,000             | 1.8%  |
| NCBI_snp.chr1 (190M)    | 3,642,225    | 809,394              | 22.2% | 15                      | <0.1% | 59                | <0.1% |
| NCBI_gene.chr1 (24M)    | 360,350      | 14,356               | 4.0%  | 11,767                  | 3.3%  | 7,160             | 2.0%  |

# Data Optimization: one step further

It is straightforward to implement a DOM-proxy that works on BPLEX outputs.

For certain tasks like XML type validation, it is even possible to obtain algorithms that are provably more (time) efficient on BPLEX outputs than on the original tree!

→ Core Xpath evaluation has same complexity on BPLEX output as it has on DAGs.[\[LohreyM.05\]](#)

# Future Research

- Memory benchmarks for full XML files in different representations.
  
- Data access benchmarks for
  - top-down access
  - DOM access
  - real queries, hand crafted / automatically generated
  
- INCREMENTAL algorithm to find small sharing graphs!  
Implement it into Java/.NET.
  
- Updates on sharing graphs
  
- Incorporate BPLEX into XML file compression tools like XMill.

# Conclusions

In the context of XML processing there is tremendous amount of applications for

- tree automata and tree transducers

- type checking

- memory optimization

- query optimization

Most probably also other areas, e.g., verification of XML based protocols can use tree automata and transducer theory.

# The END



☺... XML, a great chance to make the world better... ☺