A Mini Course on Trees, Automata and XML

Paris
June 2004

Thomas
Schwentick

Why XML?

Why XML?

Answer

Have a look into the ≥ 20 XML papers at SIGMOD/PODS

Why Trees?

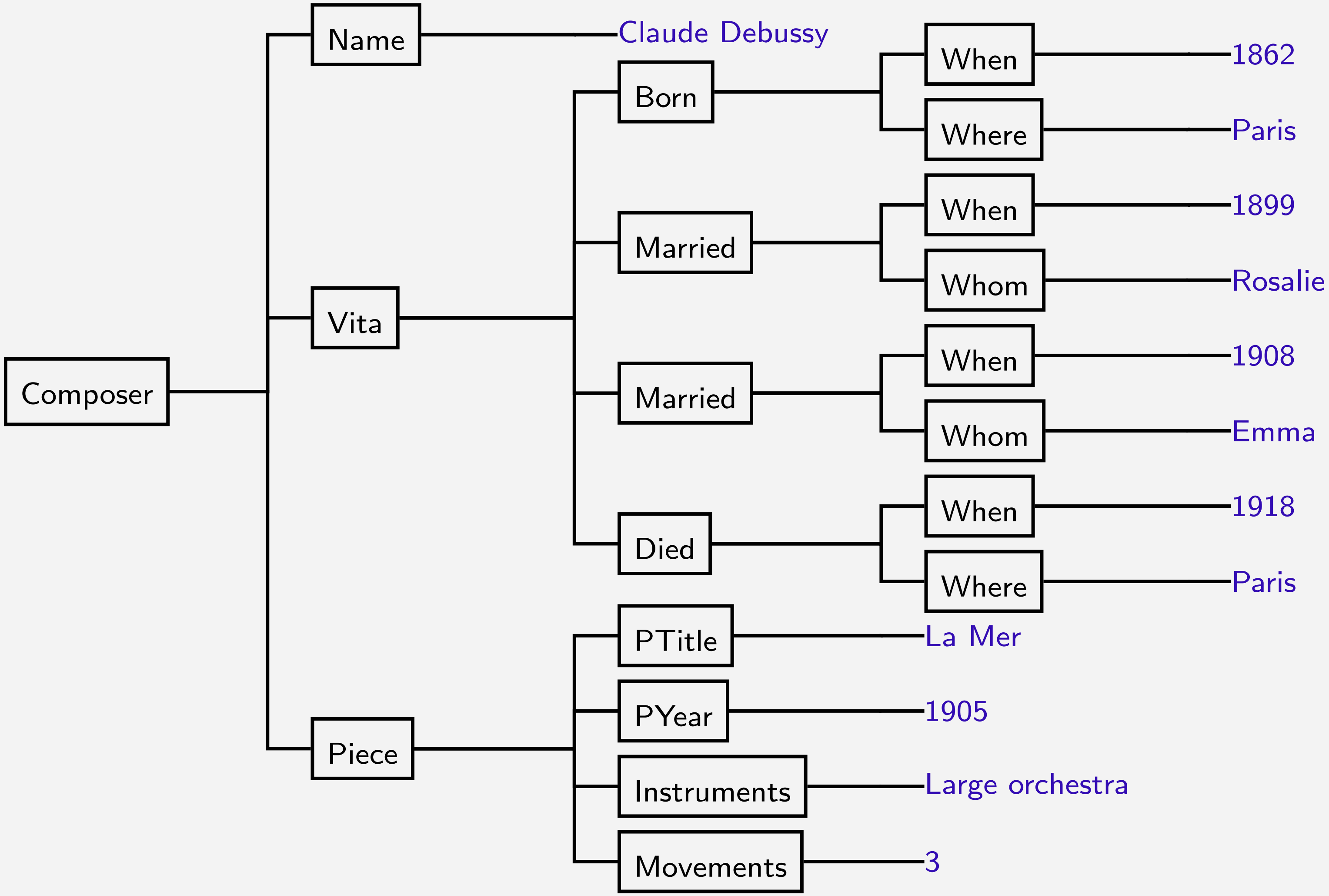
Why Trees?

Look at this:

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Why Trees?

Look at this:



- Composer
  - Name — Claude Debussy
  - Vita
    - Born
      - When — 1862
      - Where — Paris
    - Married
      - When — 1899
      - Whom — Rosalie
    - Married
      - When — 1908
      - Whom — Emma
    - Died
      - When — 1918
      - Where — Paris
  - Piece
    - PTitle — La Mer
    - PYear — 1905
    - Instruments — Large orchestra
    - Movements — 3

Why Automata?

Why Automata?

Answer

That's our topic for the remaining 88 minutes

I

- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II

- Schemas
- Node Selecting Queries
- XPath Query Containment

III

- Extensions
- Conclusion

Question: Why is XML appealing for Theory people?

Question: Why is XML appealing for Theory people?

Years ago...

- Theoretical Computer Science for Database Theorists: Logics, Complexity, Algorithms,...
- Database Theory for Theoretical Computer Scientists:

Question: Why is XML appealing for Theory people?

Years ago...

- Theoretical Computer Science for Database Theorists: Logics, Complexity, Algorithms,...
- Database Theory for Theoretical Computer Scientists: terra incognita

Question: Why is XML appealing for Theory people?

Years ago...

- Theoretical Computer Science for Database Theorists: Logics, Complexity, Algorithms,...
- Database Theory for Theoretical Computer Scientists: terra incognita

After the advent of XML

Many connections between Formal Languages & Automata Theory and XML & Database Theory

Question: Why trees?

Question: Why trees?

A Natural Answer

- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

Question: Why trees?

A Natural Answer

- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

Limitations

- But trees can not model all aspects of XML (e.g., IDREFs, data values)

⇒ Sometimes extensions are needed

- E.g., directed graphs instead of trees

Question: Why trees?

A Natural Answer

- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

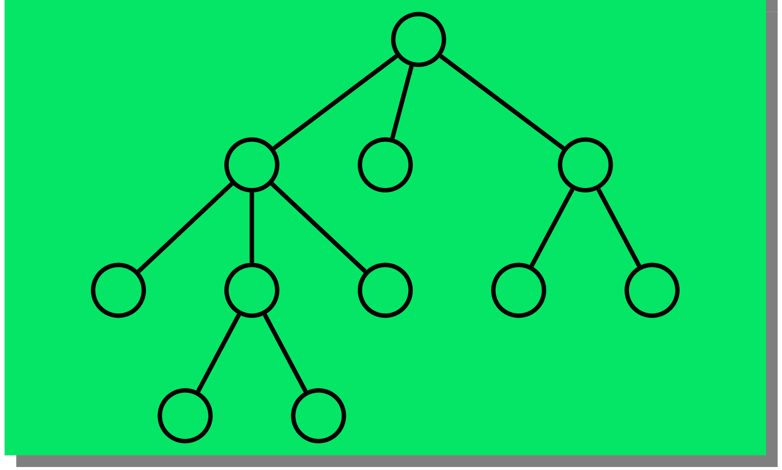
Limitations

- But trees can not model all aspects of XML (e.g., IDREFs, data values)

⇒ Sometimes extensions are needed

- E.g., directed graphs instead of trees

Example



Question: Why trees?

A Natural Answer

- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

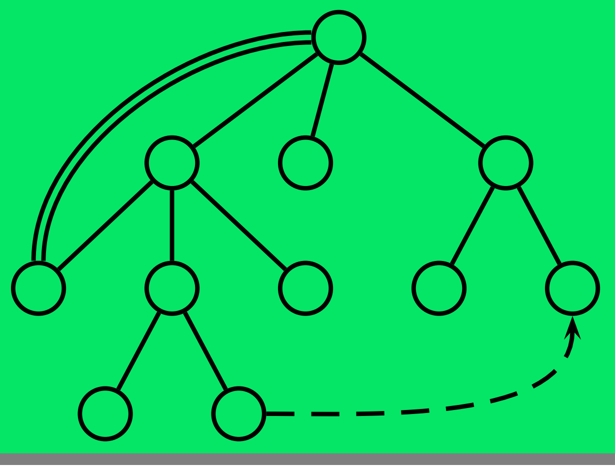
Limitations

- But trees can not model all aspects of XML (e.g., IDREFs, data values)

⇒ Sometimes extensions are needed

- E.g., directed graphs instead of trees

Example



Question: Why trees?

A Natural Answer

- Trees reflect the hierarchical structure of XML
- Underlying data model of XML is tree based

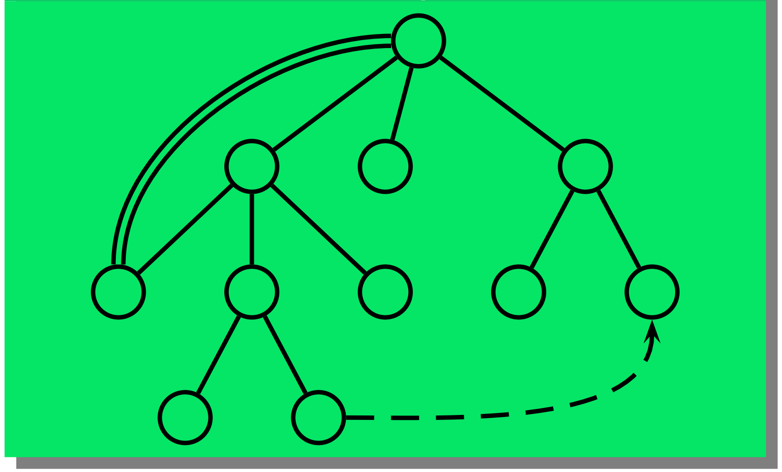
Limitations

- But trees can not model all aspects of XML (e.g., IDREFs, data values)

⇒ Sometimes extensions are needed

- E.g., directed graphs instead of trees

Example



Nevertheless

In this tutorial we will concentrate on the tree view at XML

Question: Why automata?

Ingredients of XML

Concepts from formal languages are obviously present around XML:

- Labelled trees
- `DTD`: context-free grammars
- `DTD`: regular expressions
- `XPath`: regular path expressions

We will see

Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation

Question: Why automata?

Ingredients of XML

Concepts from formal languages are obviously present around XML:

- Labelled trees
- `DTD`: context-free grammars
- `DTD`: regular expressions
- `XPath`: regular path expressions

We will see

Automata turn out to be useful as:

→ a means to define robust classes with clear semantics

- a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation

Question: Why automata?

Ingredients of XML

Concepts from formal languages are obviously present around XML:

- Labelled trees
- `DTD`: context-free grammars
- `DTD`: regular expressions
- `XPath`: regular path expressions

We will see

Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- → a tool for proofs
- an algorithmic tool for static analysis
- a tool for query evaluation

Question: Why automata?

Ingredients of XML

Concepts from formal languages are obviously present around XML:

- Labelled trees
- `DTD`: context-free grammars
- `DTD`: regular expressions
- `XPath`: regular path expressions

We will see

Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- a tool for proofs

→ an algorithmic tool for static analysis

- a tool for query evaluation

Question: Why automata?

Ingredients of XML

Concepts from formal languages are obviously present around XML:

- Labelled trees
- `DTD`: context-free grammars
- `DTD`: regular expressions
- `XPath`: regular path expressions

We will see

Automata turn out to be useful as:

- a means to define robust classes with clear semantics
- a tool for proofs
- an algorithmic tool for static analysis

→ a tool for query evaluation

Four important kinds of XML processing

Validation

Check whether an XML document is of a given type

Navigation

Select a set of positions in an XML document

Querying

Extract information from an XML document

Transformation

Construct a new XML document from a given one

Four important kinds of XML processing . . . . and their languages

Validation `DTD, XML Schema`

Check whether an XML document is of a given type

Navigation `XPath`

Select a set of positions in an XML document

Querying `XQuery`

Extract information from an XML document

Transformation `XSLT`

Construct a new XML document from a given one

Example document

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

DTD

DTDs describe types of XML documents

Example document

⟨Composer⟩
⟨Name⟩ Claude Debussy ⟨/Name⟩
⟨Vita⟩
⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
⟨/Vita⟩
⟨Piece⟩
⟨PTitle⟩ La Mer ⟨/PTitle⟩
⟨PYear⟩ 1905 ⟨/PYear⟩
⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
⟨Movements⟩ 3 ⟨/Movements⟩
...
⟨/Piece⟩
...
⟨/Composer⟩
...

DTD

DTDs describe types of XML documents

Example document

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instrum
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Example

```
<!DOCTYPE Composers [
    <!ELEMENT Composers (Composer*)>
    <!ELEMENT Composer (Name, Vita, Piece*)>
    <!ELEMENT Vita (Born, Married*, Died?)>
    <!ELEMENT Born (When, Where)>
    <!ELEMENT Married (When, Whom)>
    <!ELEMENT Died (When, Where)>
    <!ELEMENT Piece (PTitle, PYear,
         Instruments, Movements)>
    ]>
```

Example document

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

`XPath`

XPath expressions select sets of nodes of XML documents by specifying navigational patterns

Example doc

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

`XPath`

XPath expressions select sets of nodes of XML documents by specifying navigational patterns

Example doc

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Example query

`//Vita/Died/*`

`XPath`

XPath expressions select sets of nodes of XML documents by specifying navigational patterns

Example doc

```
⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
    ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
    ⟨Movements⟩ 3 ⟨/Movements⟩
    ...
  ⟨/Piece⟩
  ...
⟨/Composer⟩
...
```

Example query

`//Vita/Died/*`

Example document

⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
    ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
    ⟨Movements⟩ 3 ⟨/Movements⟩
    ...
  ⟨/Piece⟩
  ...
⟨/Composer⟩
...

`XQuery`

`XQuery` is a full-fledged XML query language

Example document

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

`XQuery`

`XQuery` is a full-fledged XML query language

Example document

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Example query

```
for $x in doc(‘composers.xml’)/Composer
where $x/Vita/Died/Where = ‘Paris’
return $x/Name
```

Result

```
⟨Name⟩ Claude Debussy ⟨/Name⟩
⟨Name⟩ Eric Satie ⟨/Name⟩
⟨Name⟩ Hector Berlioz ⟨/Name⟩
⟨Name⟩ Camille Saint-Saëns ⟨/Name⟩
⟨Name⟩ Frédéric Chopin ⟨/Name⟩
⟨Name⟩ Maurice Ravel ⟨/Name⟩
⟨Name⟩ Jim Morrison ⟨/Name⟩
⟨Name⟩ César Franck ⟨/Name⟩
⟨Name⟩ Gabriel Fauré ⟨/Name⟩
⟨Name⟩ George Bizet ⟨/Name⟩
...
```

XQuery

`XQuery` is a full-fledged XML query language

nt

here⟩ Paris ⟨/Where⟩⟨/Born⟩
om⟩ Rosalie⟨/Whom⟩⟨/Married⟩
om⟩ Emma ⟨/Whom⟩⟨/Married⟩
re⟩ Paris ⟨/Where⟩ ⟨/Died⟩

⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
⟨Movements⟩ 3 ⟨/Movements⟩
...
⟨/Piece⟩
...
⟨/Composer⟩
...

Example query

```
for $x in doc('composers.xml')/Composer
where $x/Vita/Died/Where = 'Paris'
return $x/Name
```

Example document

```
⟨Composer⟩
    ⟨Name⟩ Claude Debussy ⟨/Name⟩
    ⟨Vita⟩
        ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
        ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
        ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
        ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
    ⟨/Vita⟩
    ⟨Piece⟩
        ⟨PTitle⟩ La Mer ⟨/PTitle⟩
        ⟨PYear⟩ 1905 ⟨/PYear⟩
        ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
        ⟨Movements⟩ 3 ⟨/Movements⟩
        ...
    ⟨/Piece⟩
    ...
⟨/Composer⟩
...
```

`XSLT`

XSLT transforms documents by means of templates

Example documen

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

XSLT

XSLT transforms documents by means of templates

Example documen

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestr
      ⟨Movements⟩ 3 ⟨/Movemen
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Example

```
⟨xsl:template match=”Composer[Vita//Where=’Paris’]”⟩
   ⟨ParisComposer⟩
      ⟨xsl:copy-of select=“Name” /⟩
      ⟨xsl:copy-of select=“Vita/Born” /⟩
   ⟨/ParisComposer⟩
⟨/xsl:template⟩
```

Result

⟨ParisComposer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Born⟩
    ⟨When⟩ August 22, 1862 ⟨/When⟩
    ⟨Where⟩ Paris ⟨/Where⟩
  ⟨/Born⟩
⟨/ParisComposer⟩
⟨ParisComposer⟩
  ⟨Name⟩ Frédéric Chopin ⟨/Name⟩
  ⟨Born⟩
    ⟨When⟩ March 1, 1810 ⟨/When⟩
    ⟨Where⟩ Želazowa ⟨/Where⟩
  ⟨/Born⟩
⟨/ParisComposer⟩
⟨ParisComposer⟩
  ⟨Name⟩ Camille Saint-Saëns ⟨/Name⟩
  ⟨Born⟩
    ⟨When⟩ October 9, 1835 ⟨/When⟩
    ⟨Where⟩ Paris ⟨/Where⟩
  ⟨/Born⟩
⟨/ParisComposer⟩

XSLT

XSLT transforms documents by means of templates

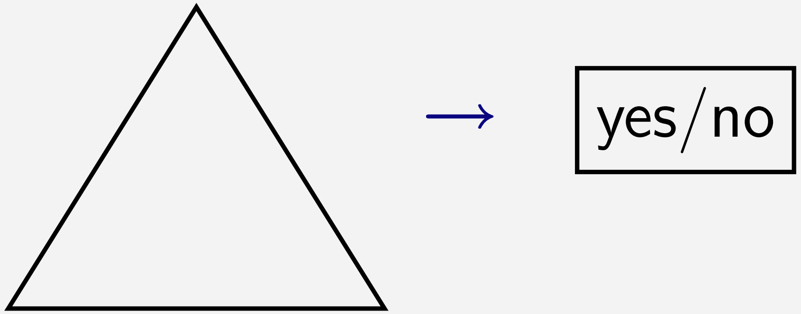
ocumen

hen⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
en⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
en⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
n⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩

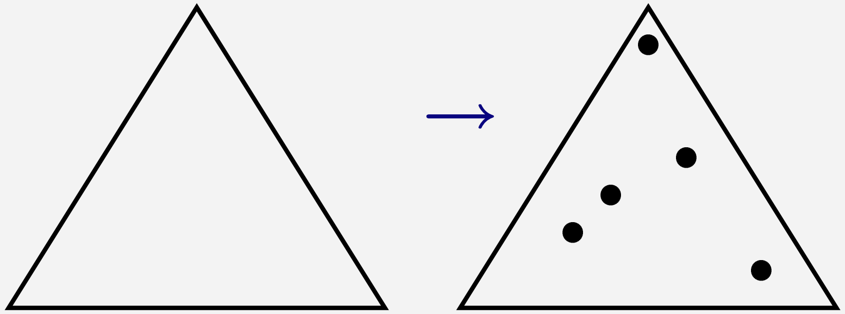
Example

nplate match="Composer[Vita//Where='Paris']"⟩
sComposer⟩
sl:copy-of select="Name" /⟩
sl:copy-of select="Vita/Born" /⟩
risComposer⟩
mplate⟩

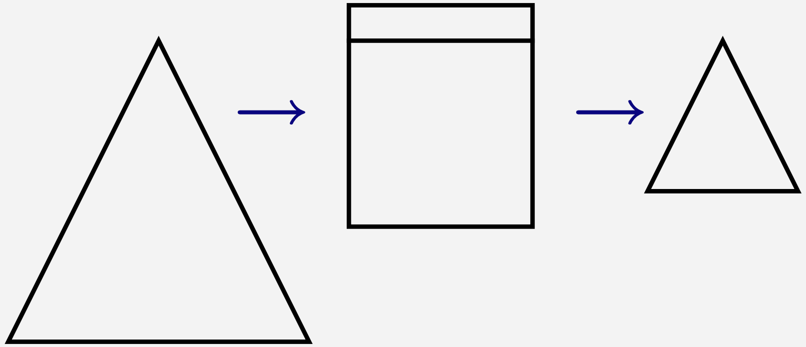
DTD/ XML Schema



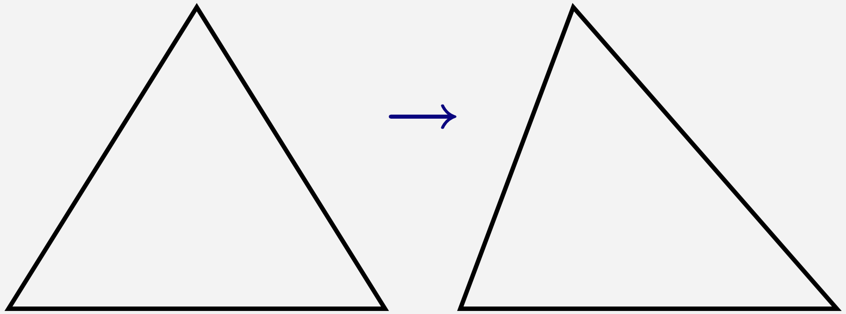
XPath



XQuery



XSLT



Aim

- Introduction
- Basic techniques and models
- Not a survey
- In particular: many important papers are not mentioned

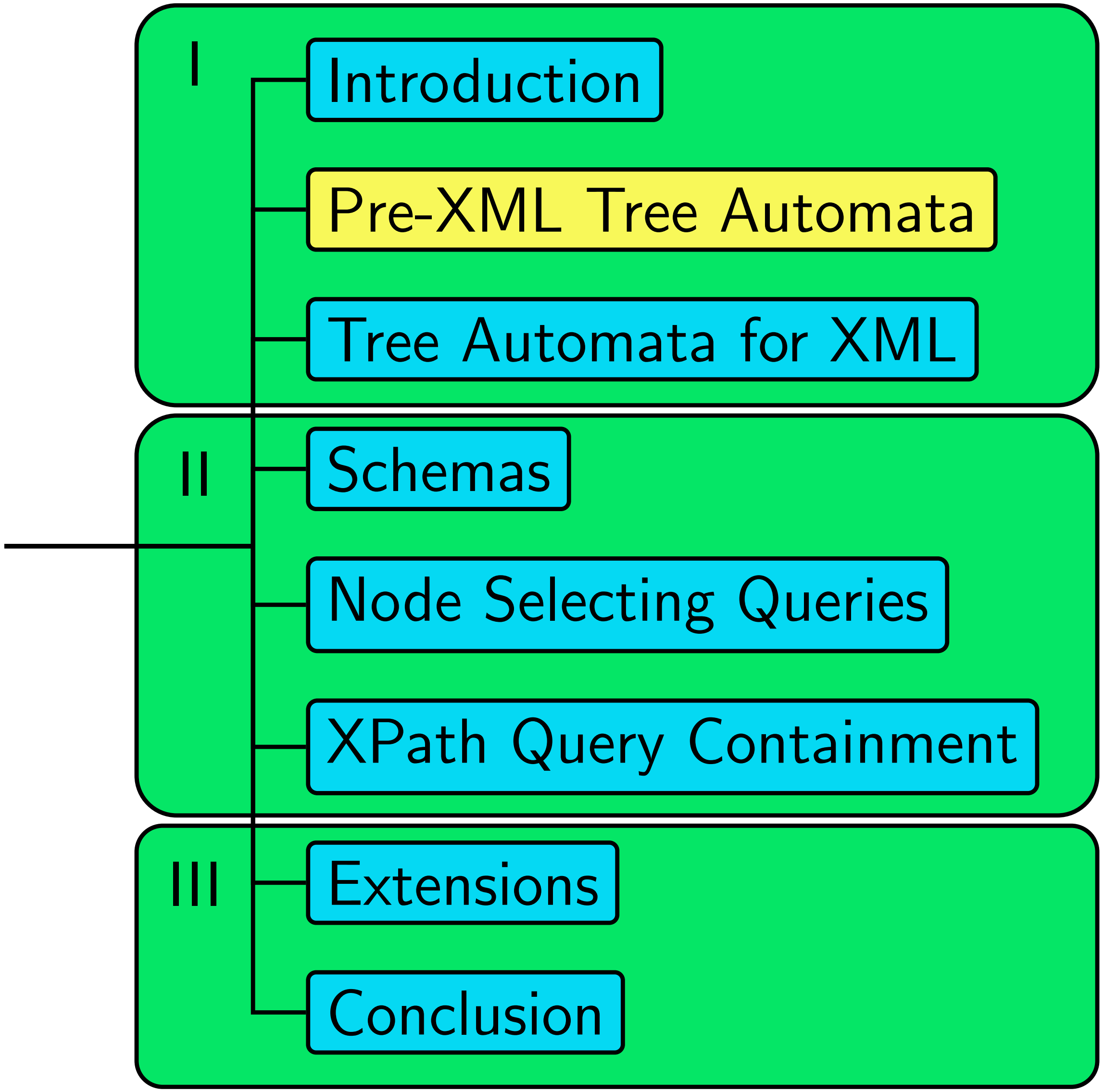
Overall structure

Part 1: Background on tree automata and how they can be adapted for XML purposes

Part 2: Examples for the use of automata for XML

- Two robust classes of schema languages
- A robust class of node-selecting queries
- Automata as an algorithmic tool for checking `XPath` query containment

Part 3: Some words about related results and about extensions and limitations



I

- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II

- Schemas
- Node Selecting Queries
- XPath Query Containment

III

- Extensions
- Conclusion

From Strings to Trees

A String

abcab

From Strings to Trees

A String

abcab

String as Tree

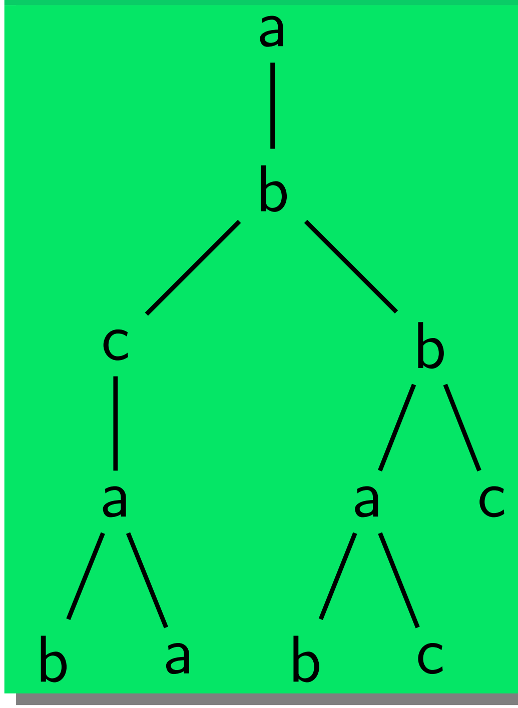
From Strings to Trees

A String

abcab

String as Tree

A Ranked Tree

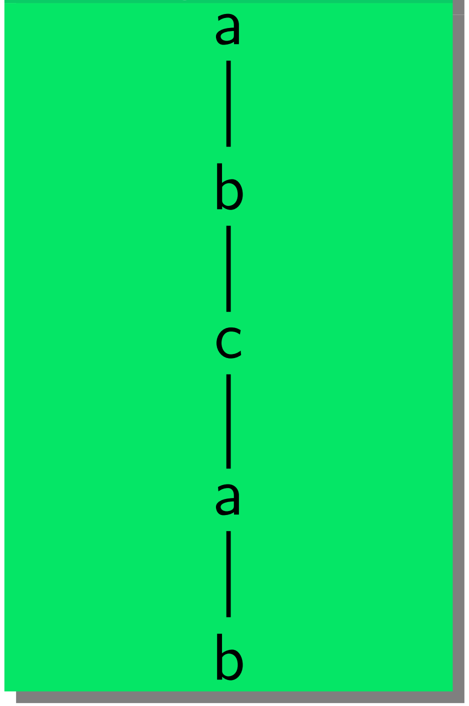


From Strings to Trees

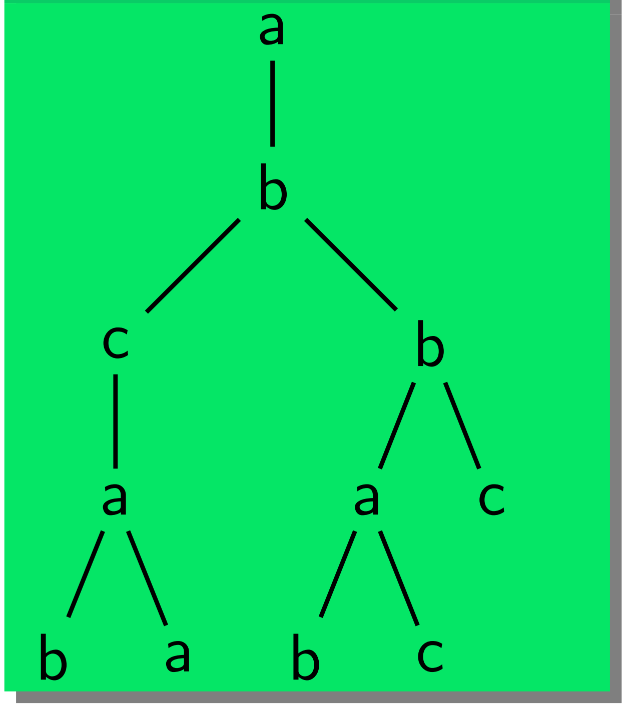
A String

abcab

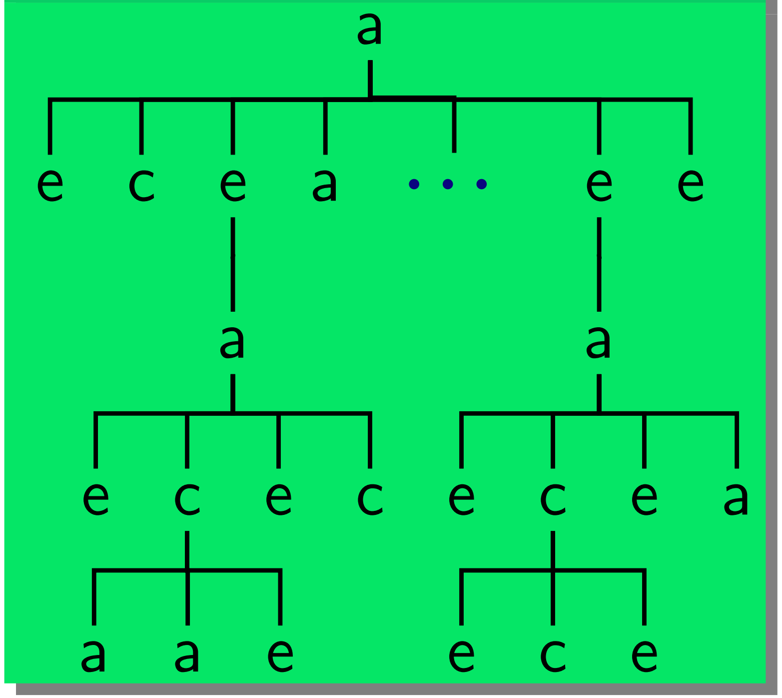
String as Tree



A Ranked Tree



An Unranked Tree



From Strings to Trees (cont.)

XML and Trees

- XML trees are unranked:
  the number of children of a node is not restricted
- Automata have first been considered on ranked trees,
  where each symbol has a fixed number of children (rank)
- Most important ideas were already developed for ranked trees

→ Let us take a look at this first

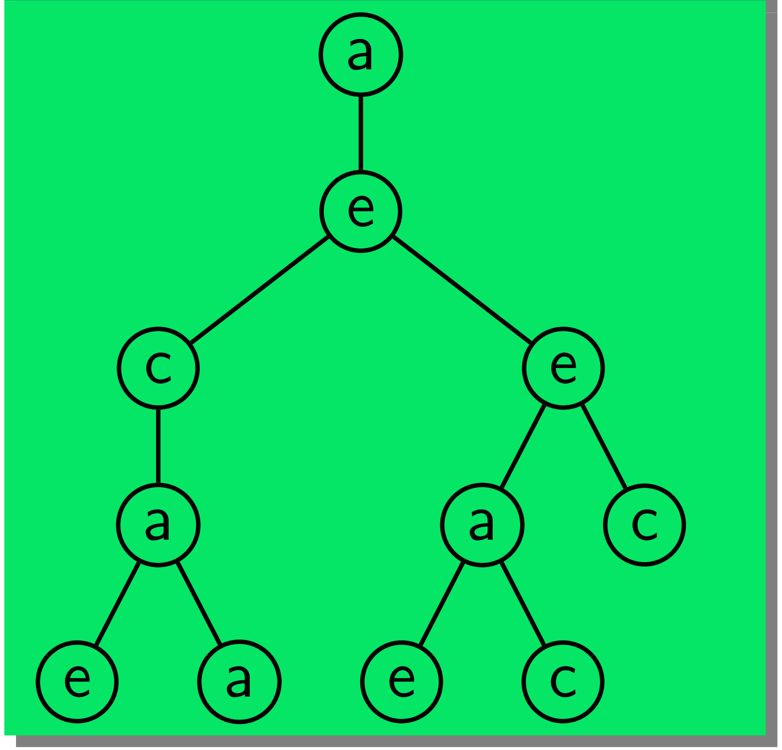
Pre-XML

From String Automata to Tree Automata

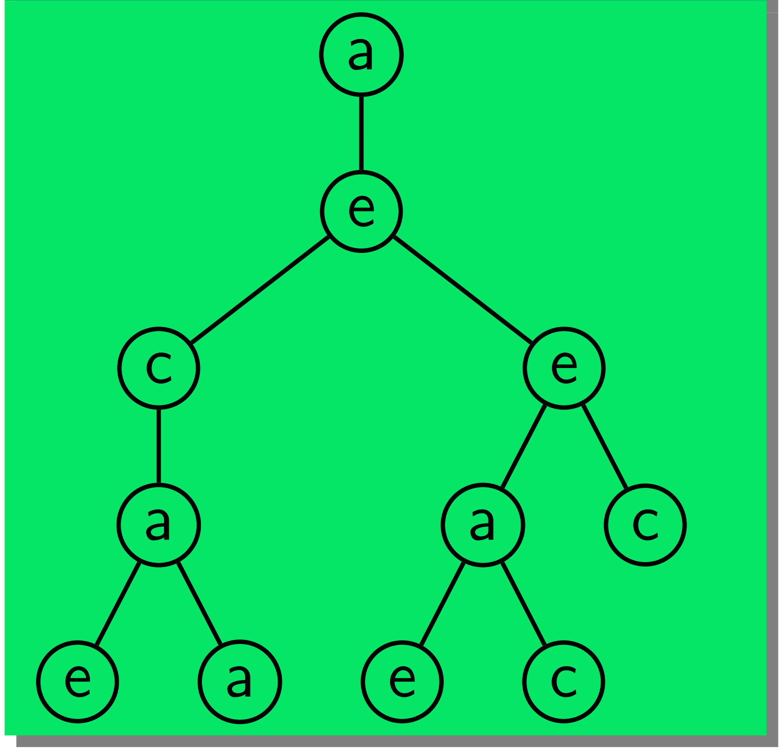
Question

How do automata generalize to trees?

Sequential



Parallel

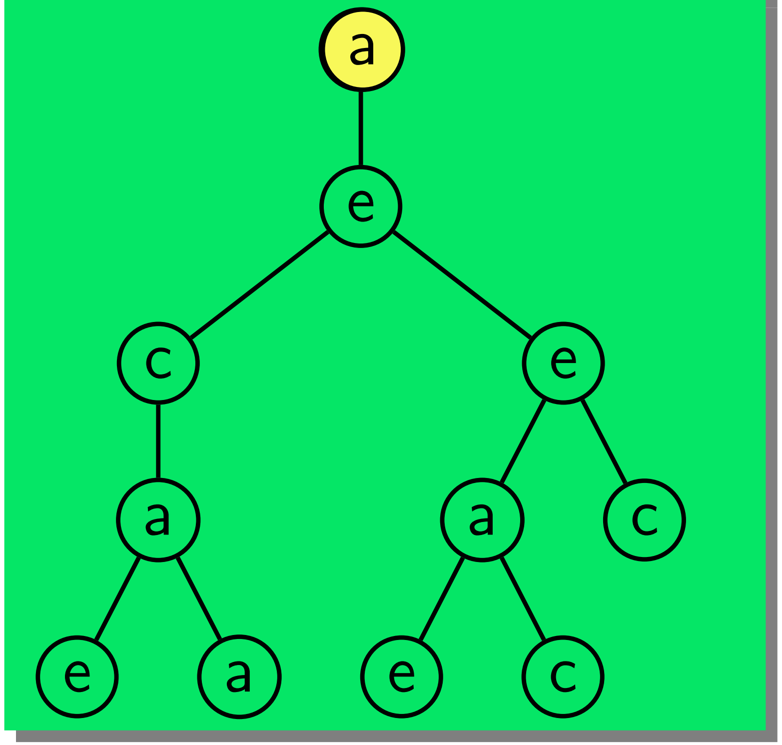


From String Automata to Tree Automata

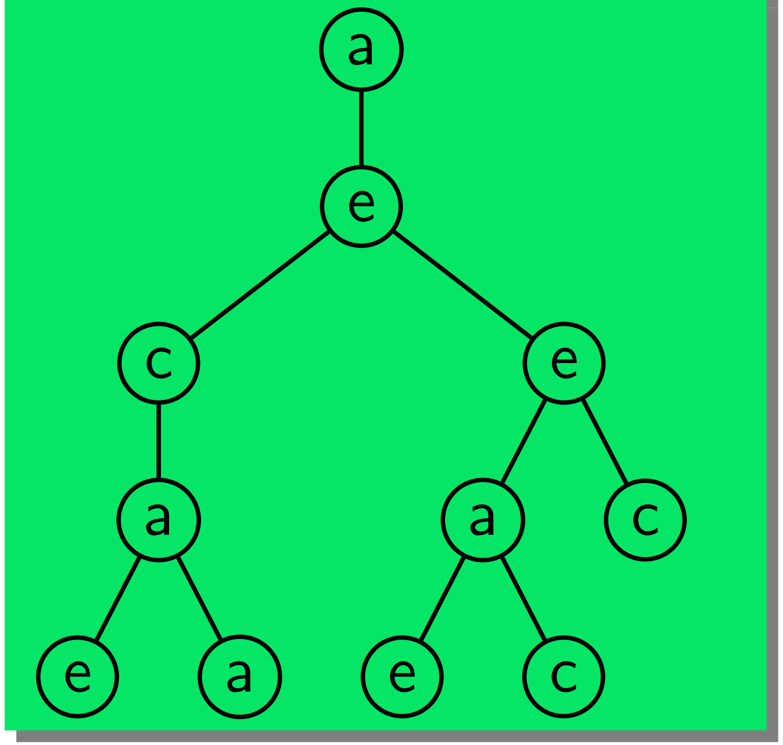
Question

How do automata generalize to trees?

Sequential



Parallel

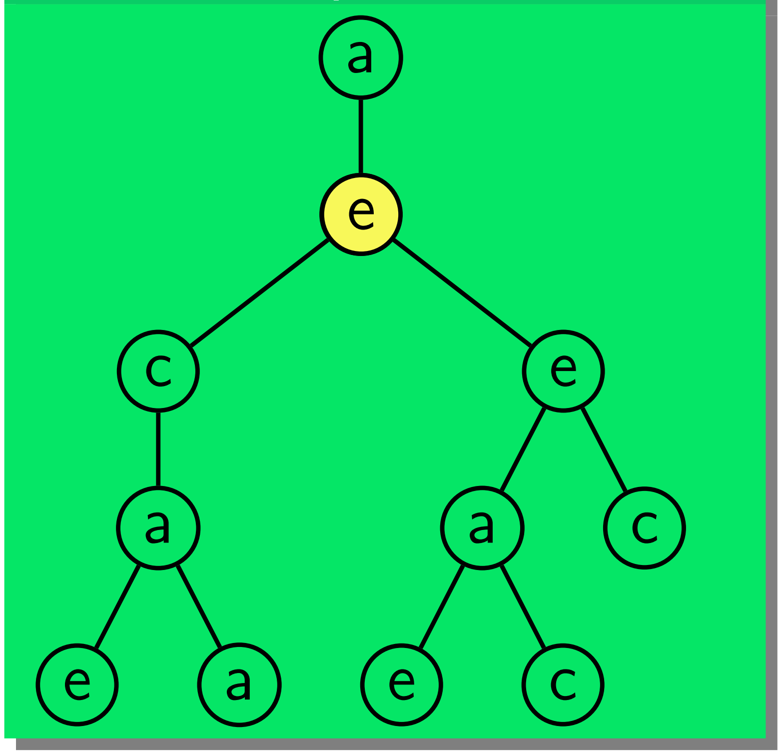


From String Automata to Tree Automata

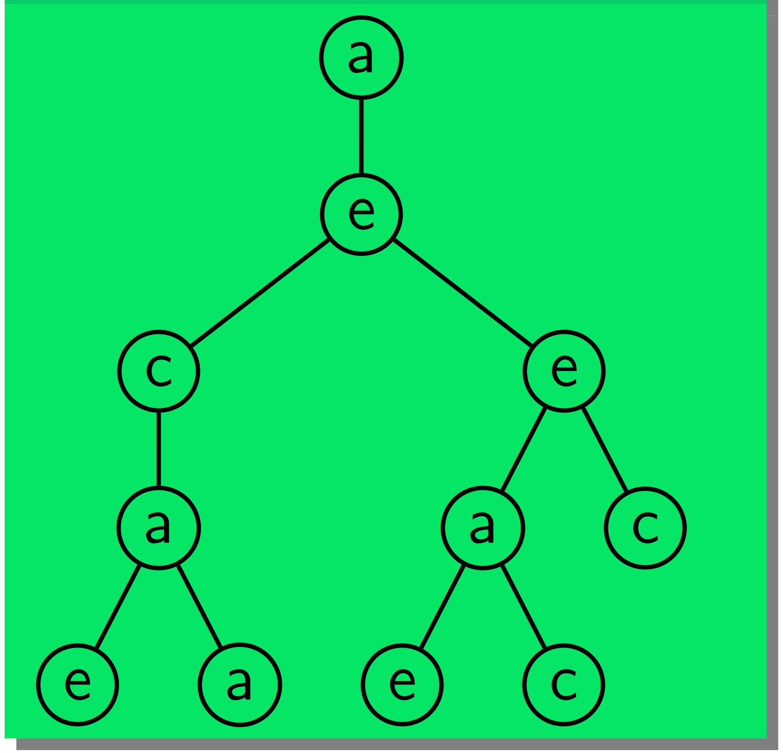
Question

How do automata generalize to trees?

Sequential



Parallel

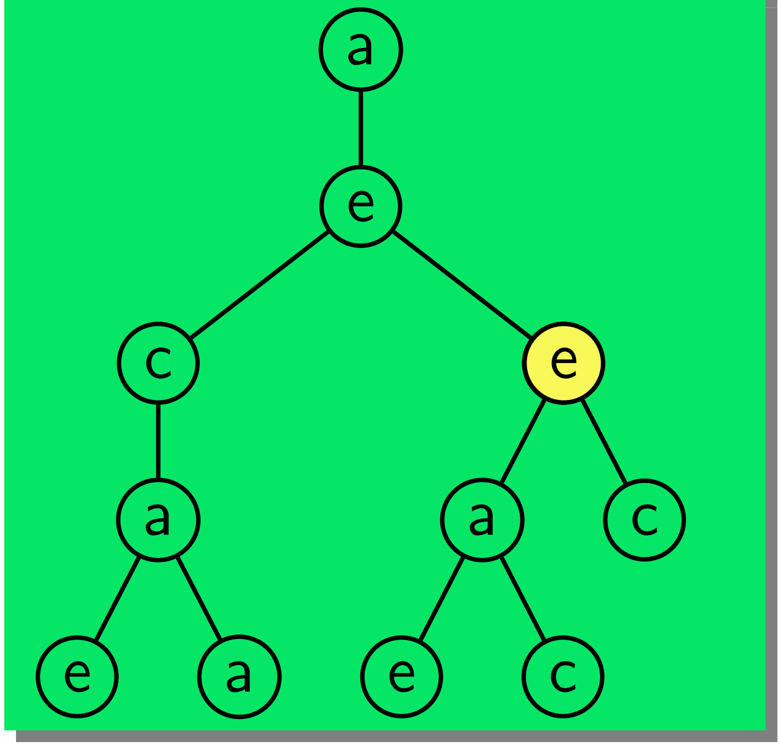


From String Automata to Tree Automata

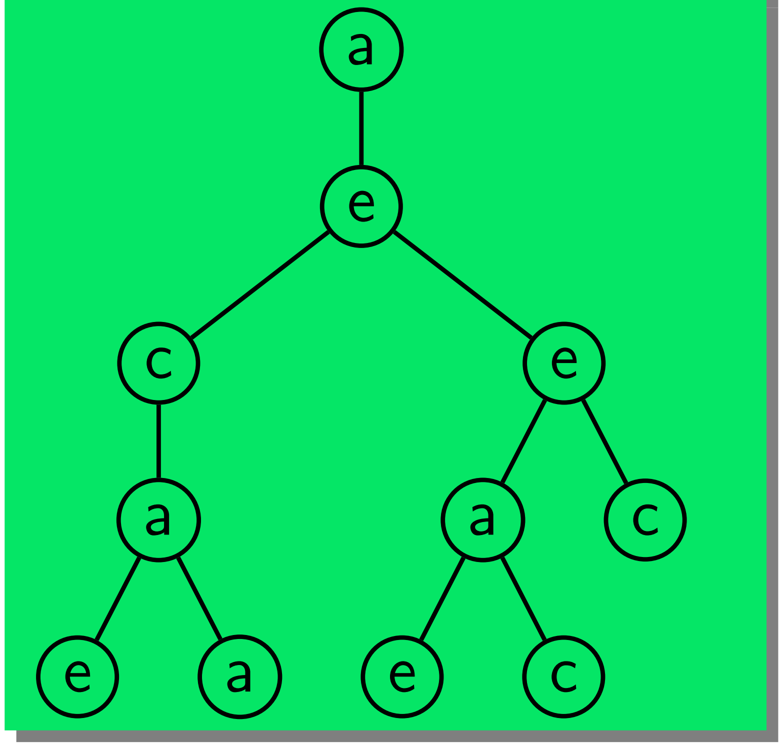
Question

How do automata generalize to trees?

Sequential



Parallel

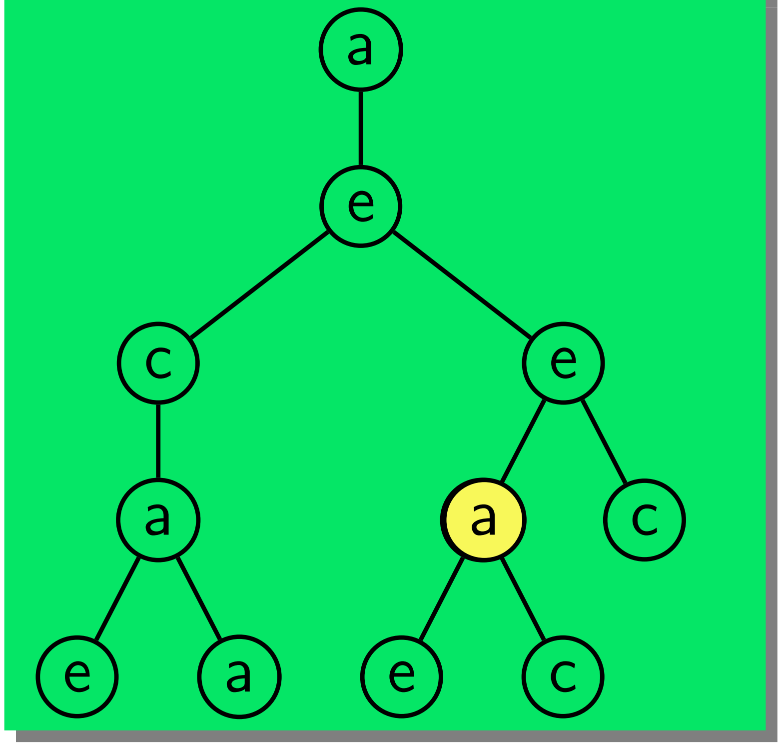


From String Automata to Tree Automata

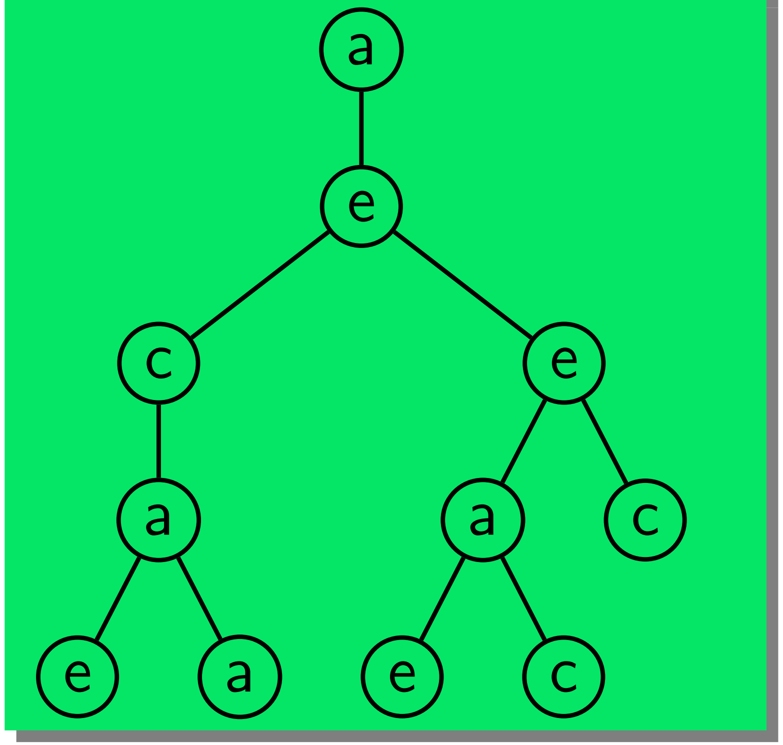
Question

How do automata generalize to trees?

Sequential



Parallel

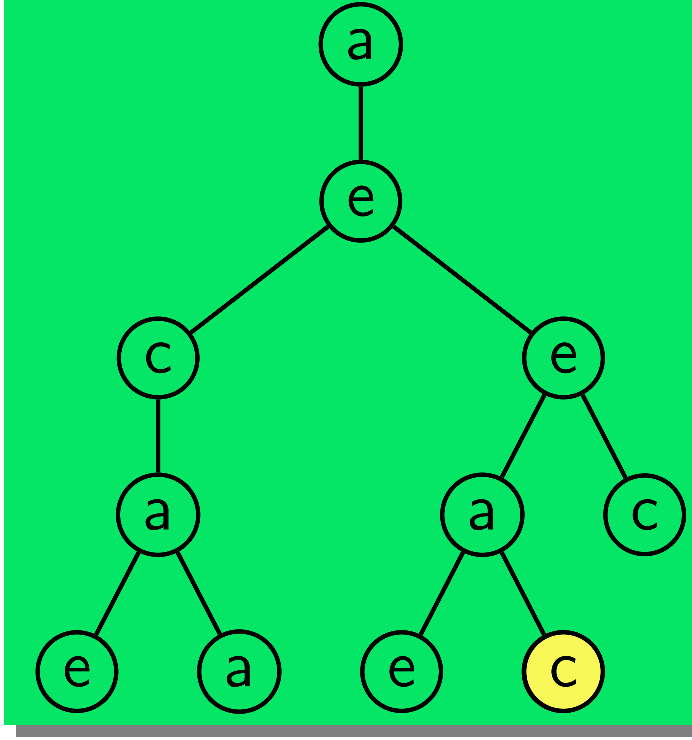


From String Automata to Tree Automata

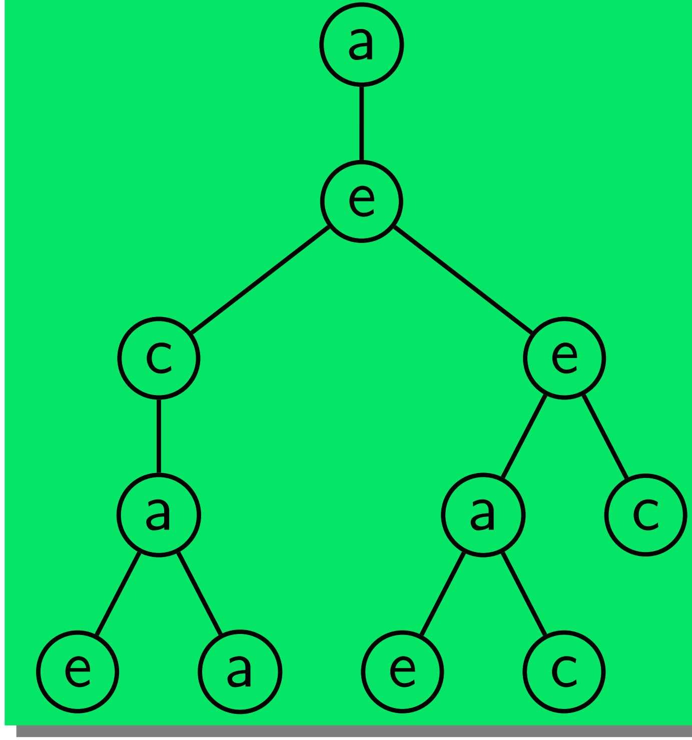
Question

How do automata generalize to trees?

Sequential



Parallel

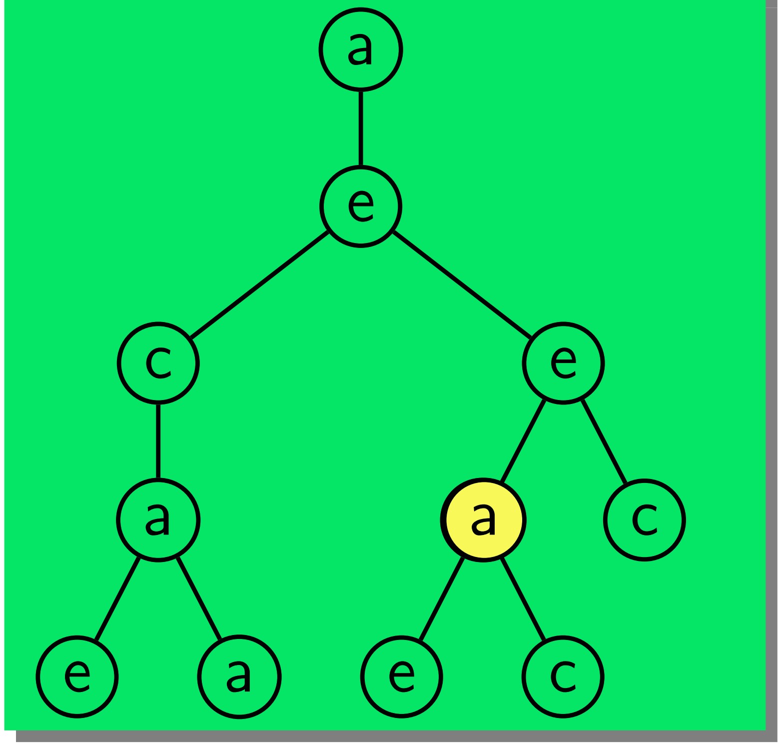


From String Automata to Tree Automata

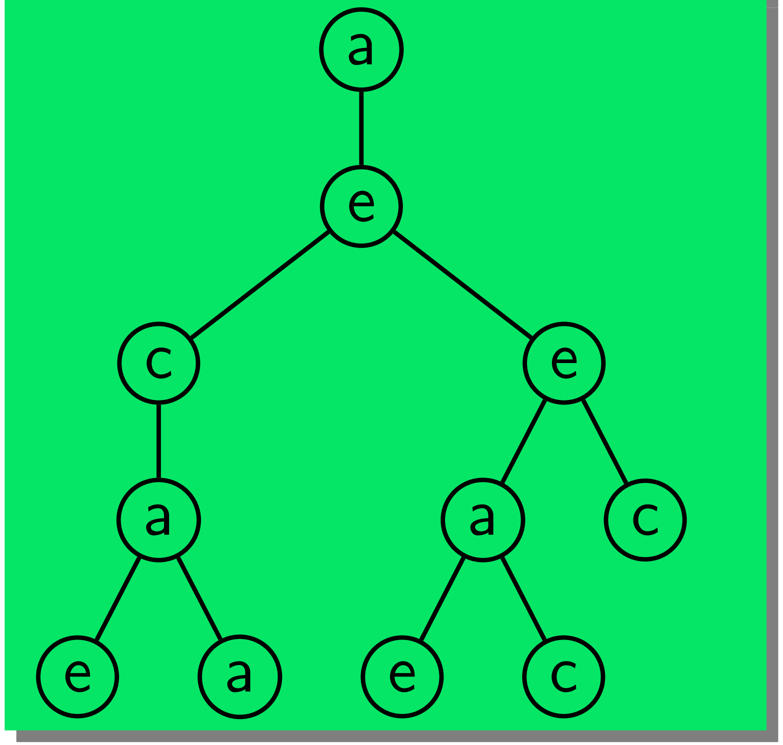
Question

How do automata generalize to trees?

Sequential



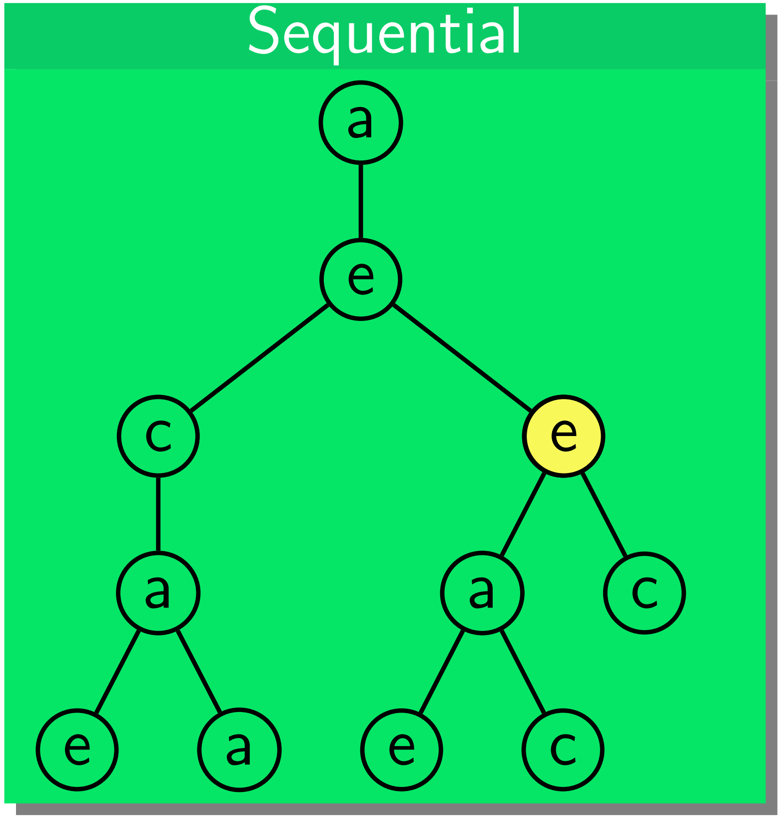
Parallel

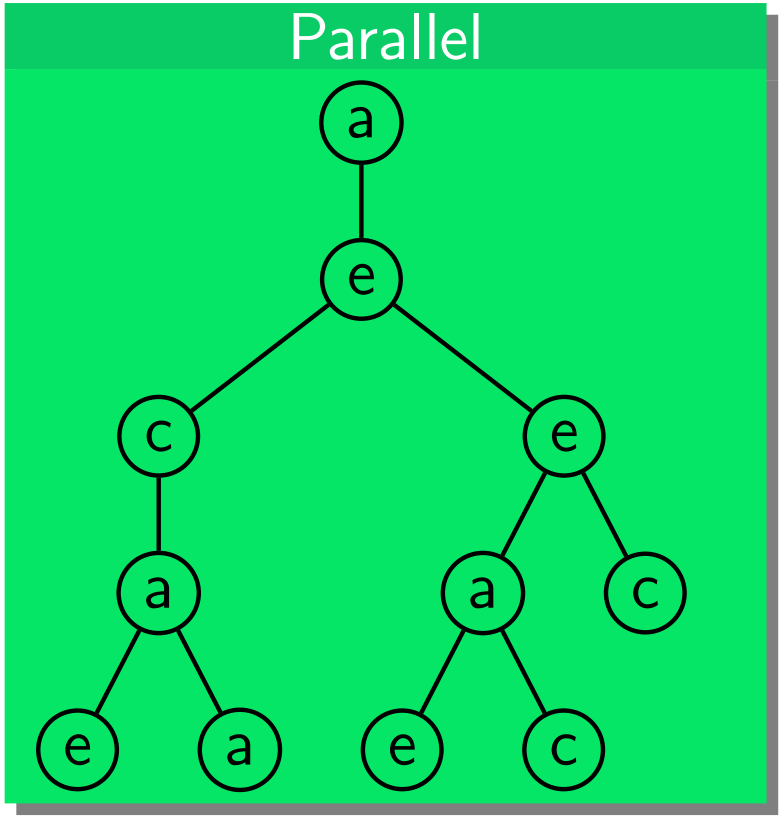


From String Automata to Tree Automata

Question

How do automata generalize to trees?


Sequential

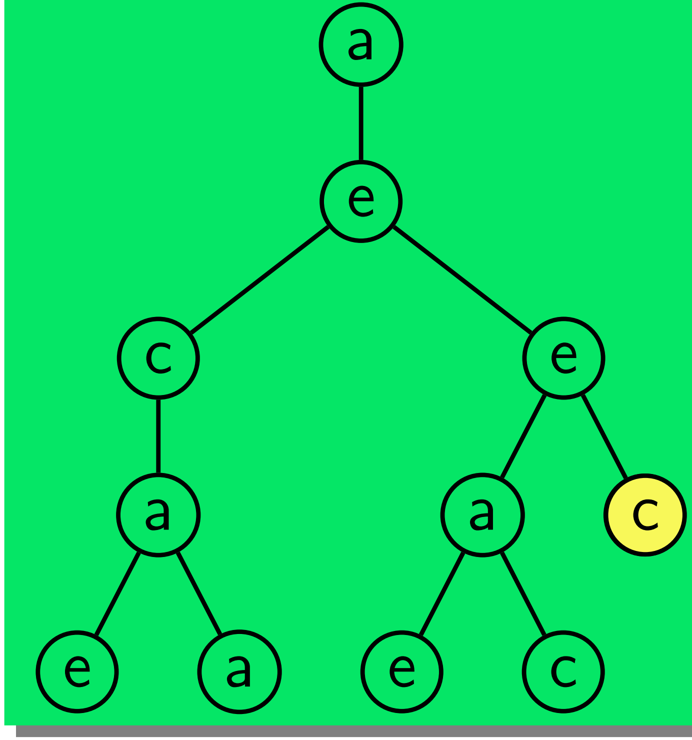

Parallel

From String Automata to Tree Automata

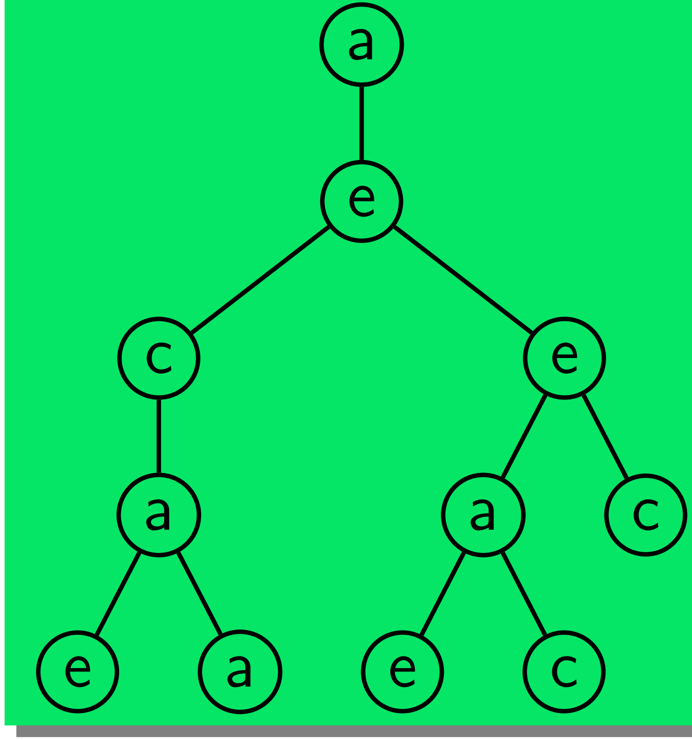
Question

How do automata generalize to trees?

Sequential



Parallel

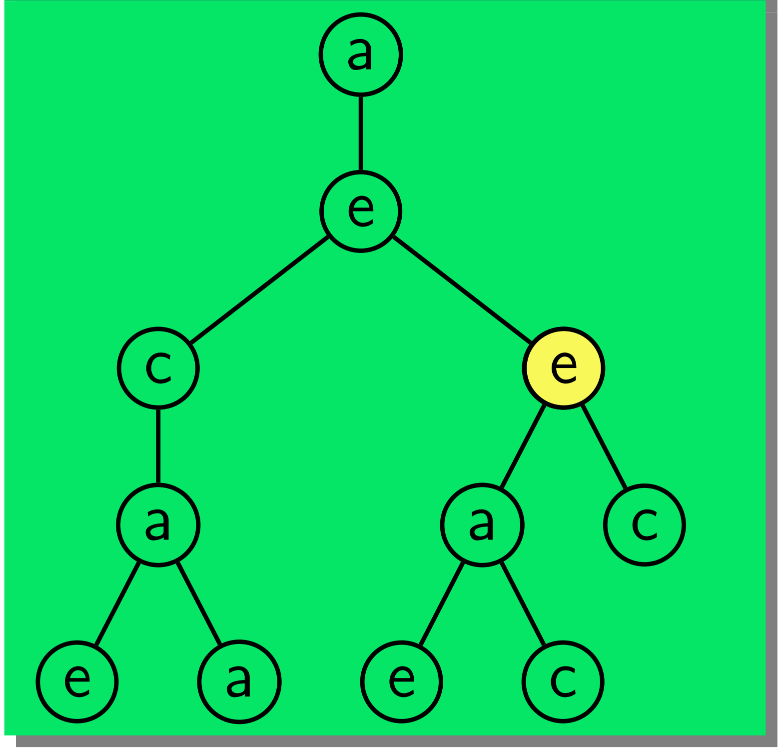


From String Automata to Tree Automata

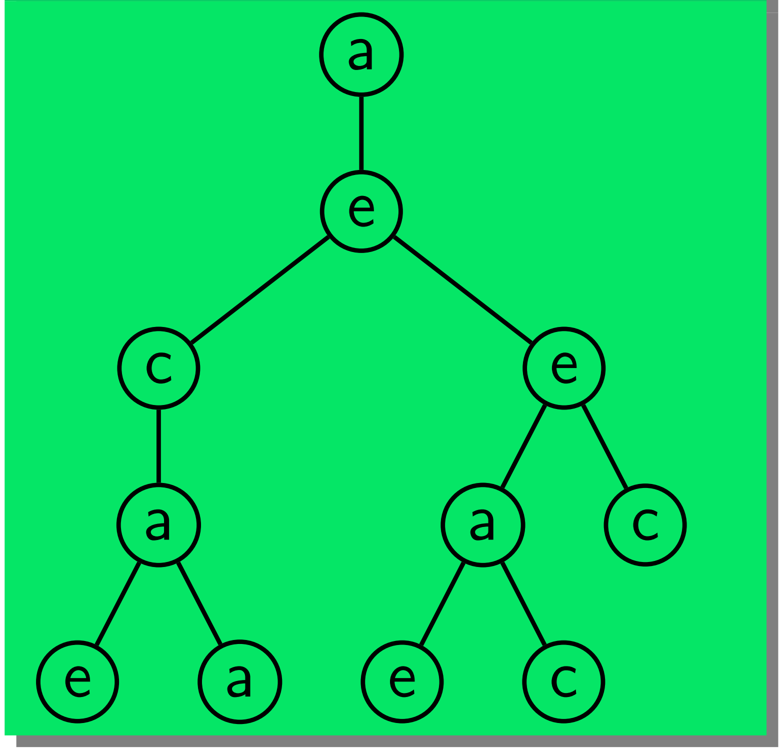
Question

How do automata generalize to trees?

Sequential



Parallel

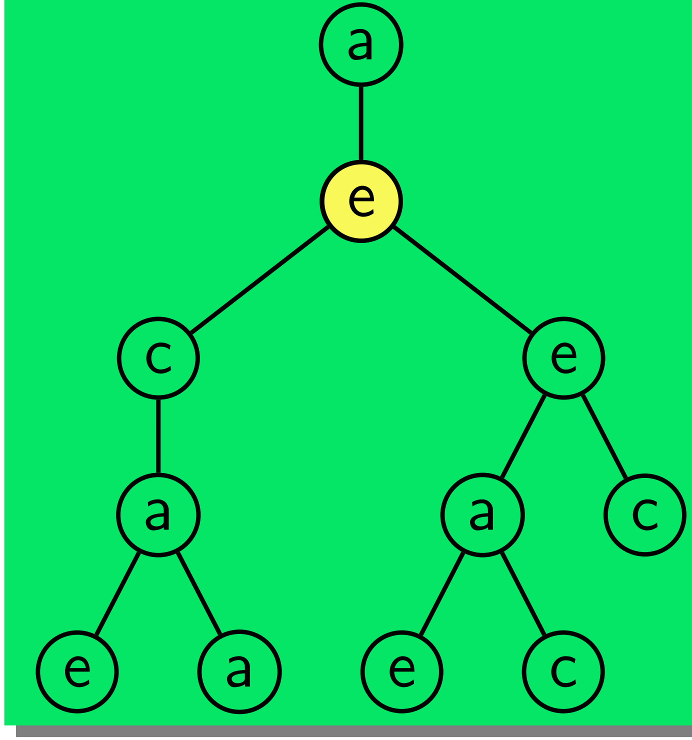


From String Automata to Tree Automata

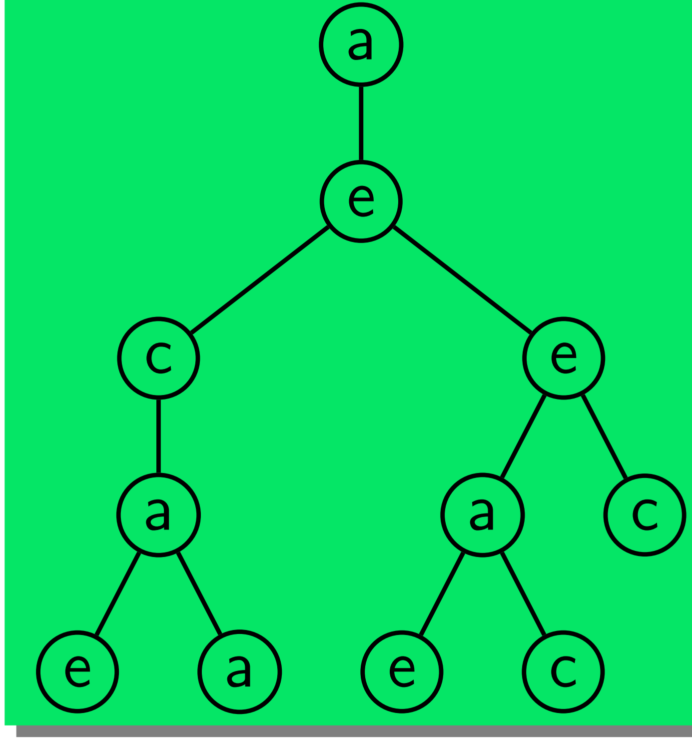
Question

How do automata generalize to trees?

Sequential



Parallel

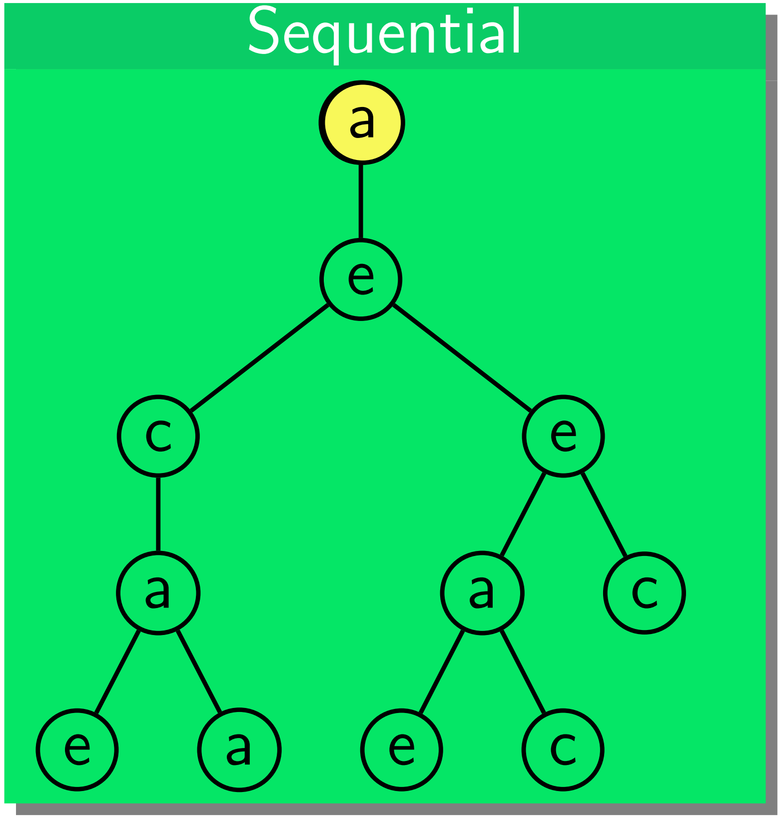


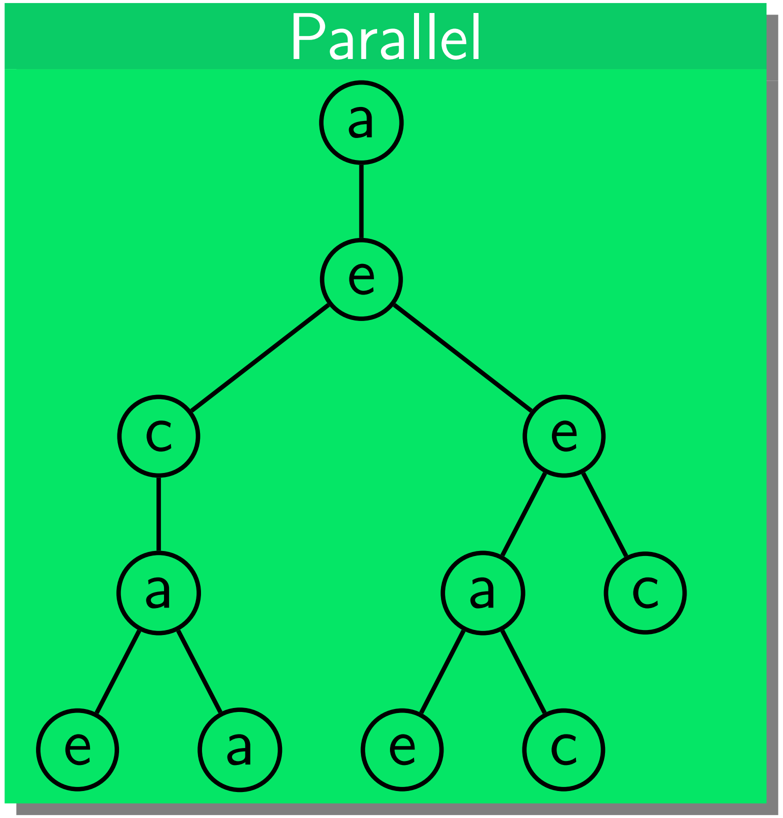
Pre-XML

From String Automata to Tree Automata

Question

How do automata generalize to trees?

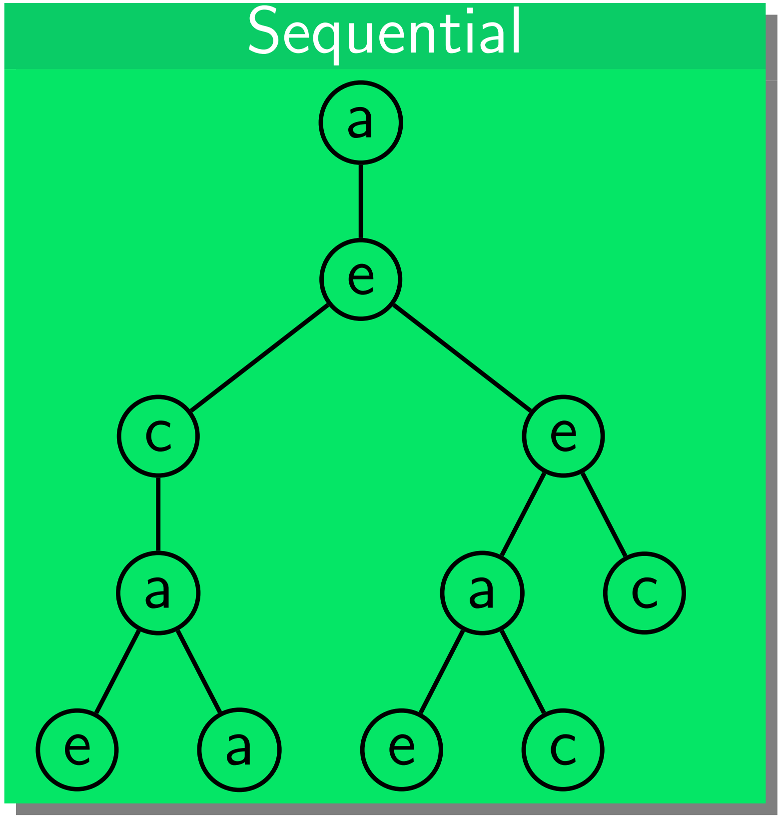


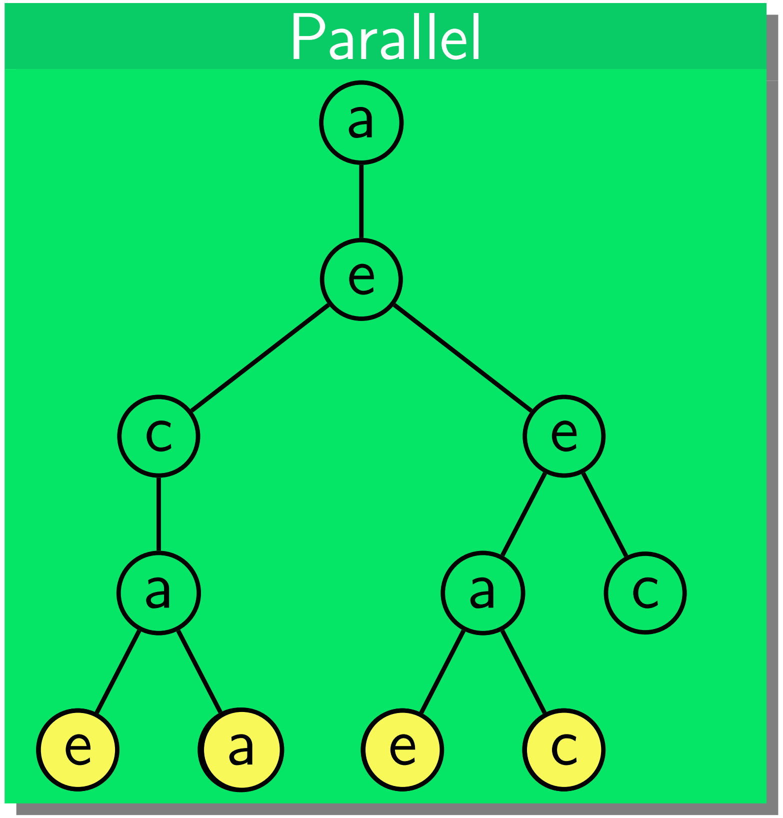


From String Automata to Tree Automata

Question

How do automata generalize to trees?


Sequential

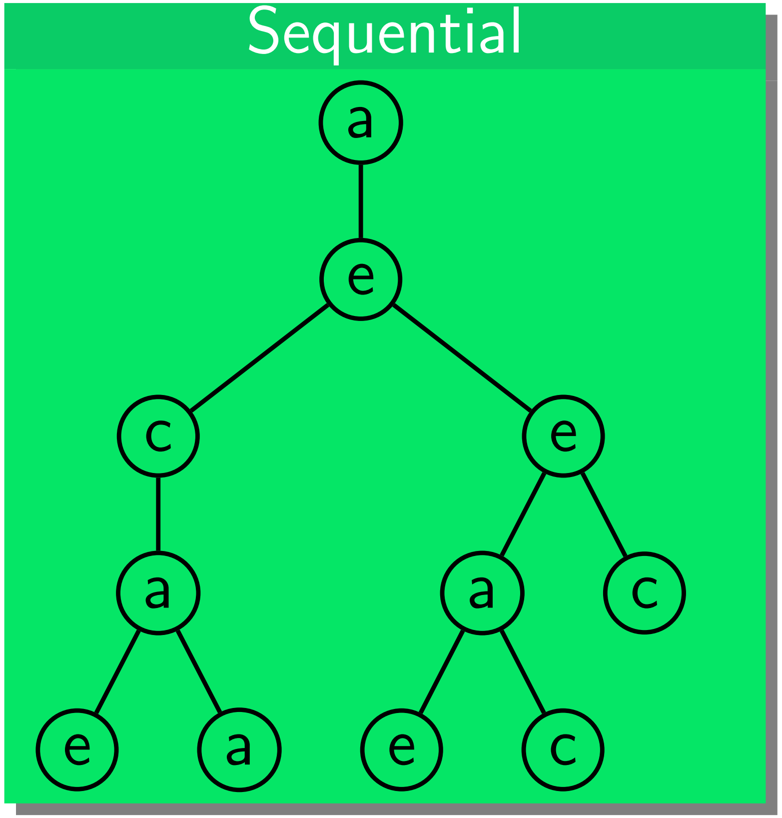

Parallel

From String Automata to Tree Automata

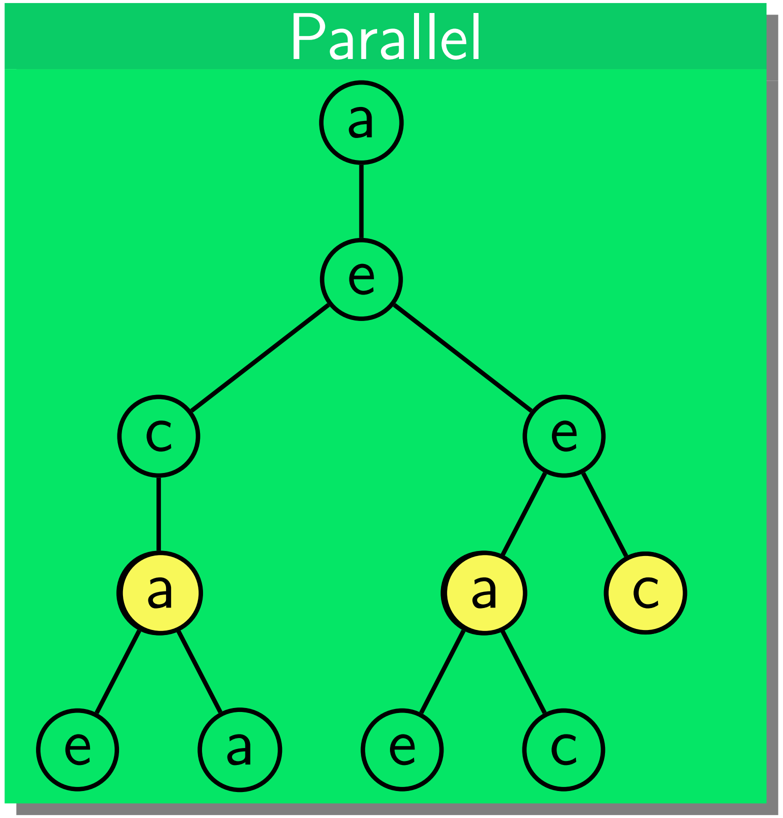
Question

How do automata generalize to trees?

Sequential



Parallel

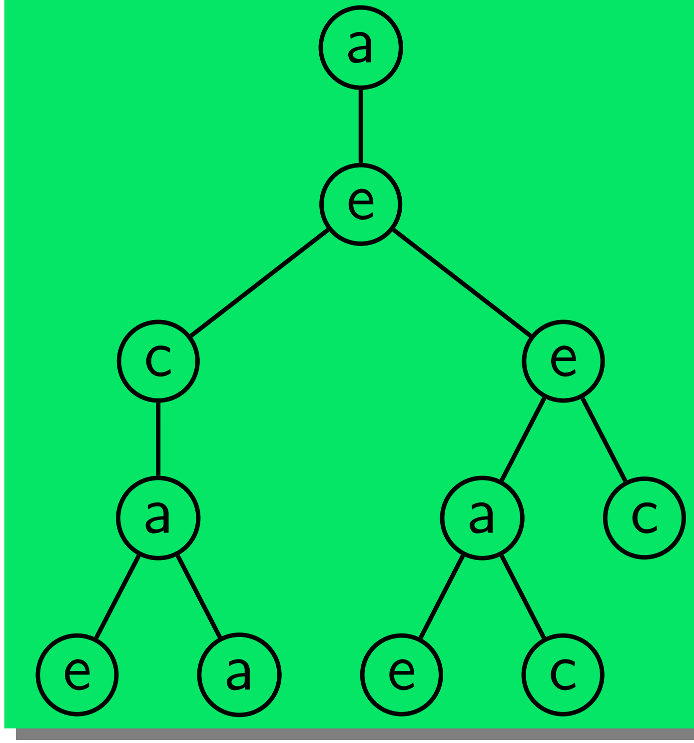


From String Automata to Tree Automata

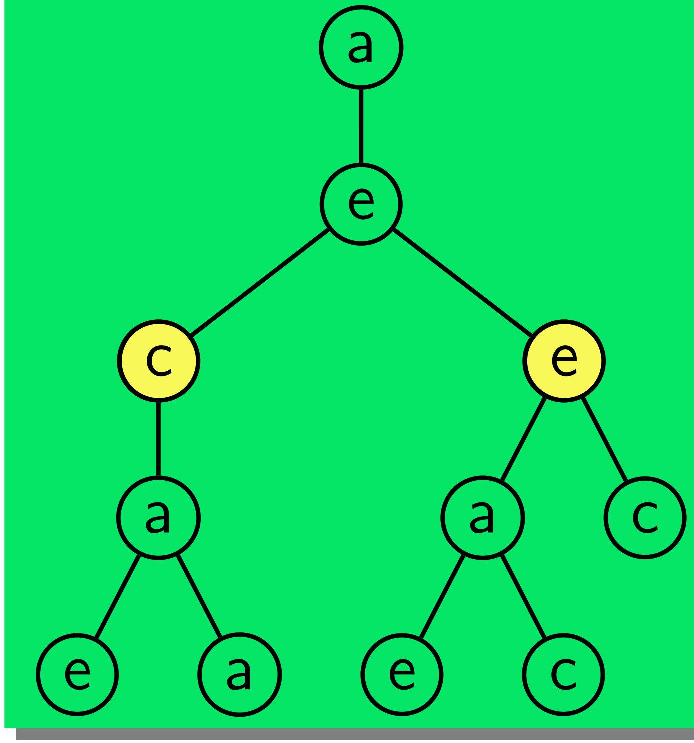
Question

How do automata generalize to trees?

Sequential



Parallel



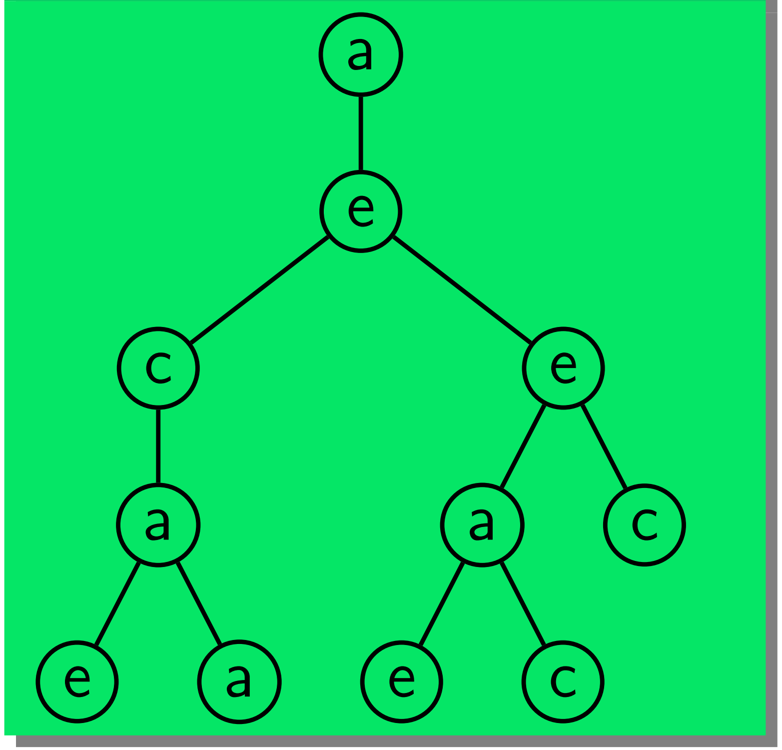
Pre-XML

From String Automata to Tree Automata

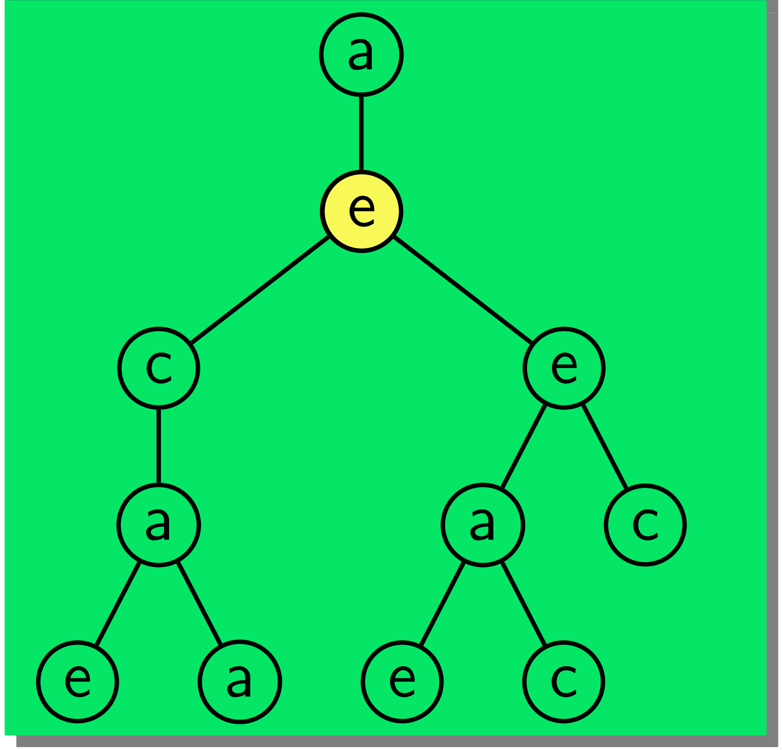
Question

How do automata generalize to trees?

Sequential



Parallel

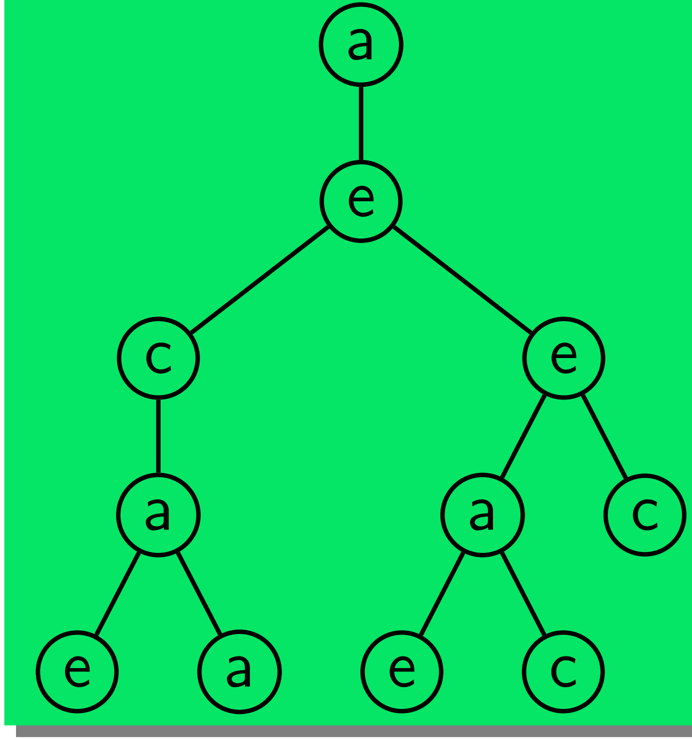


From String Automata to Tree Automata

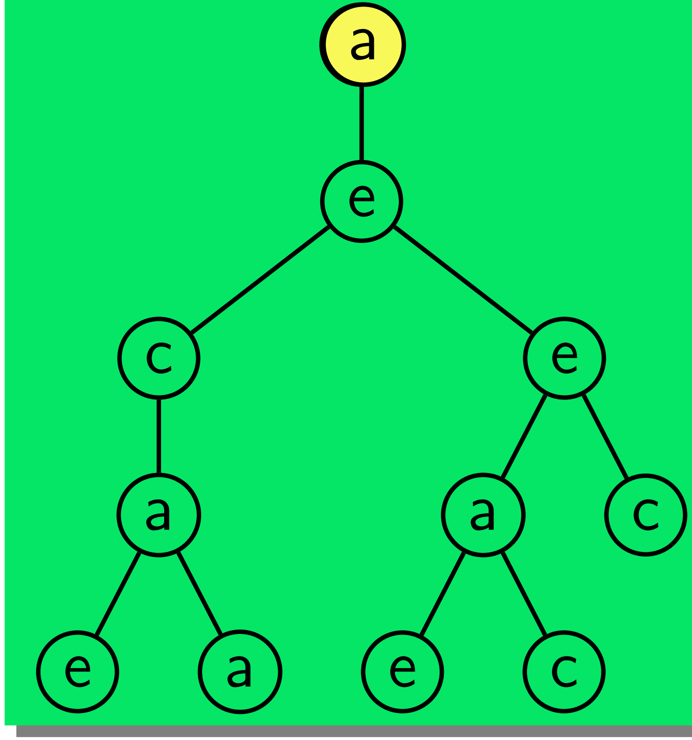
Question

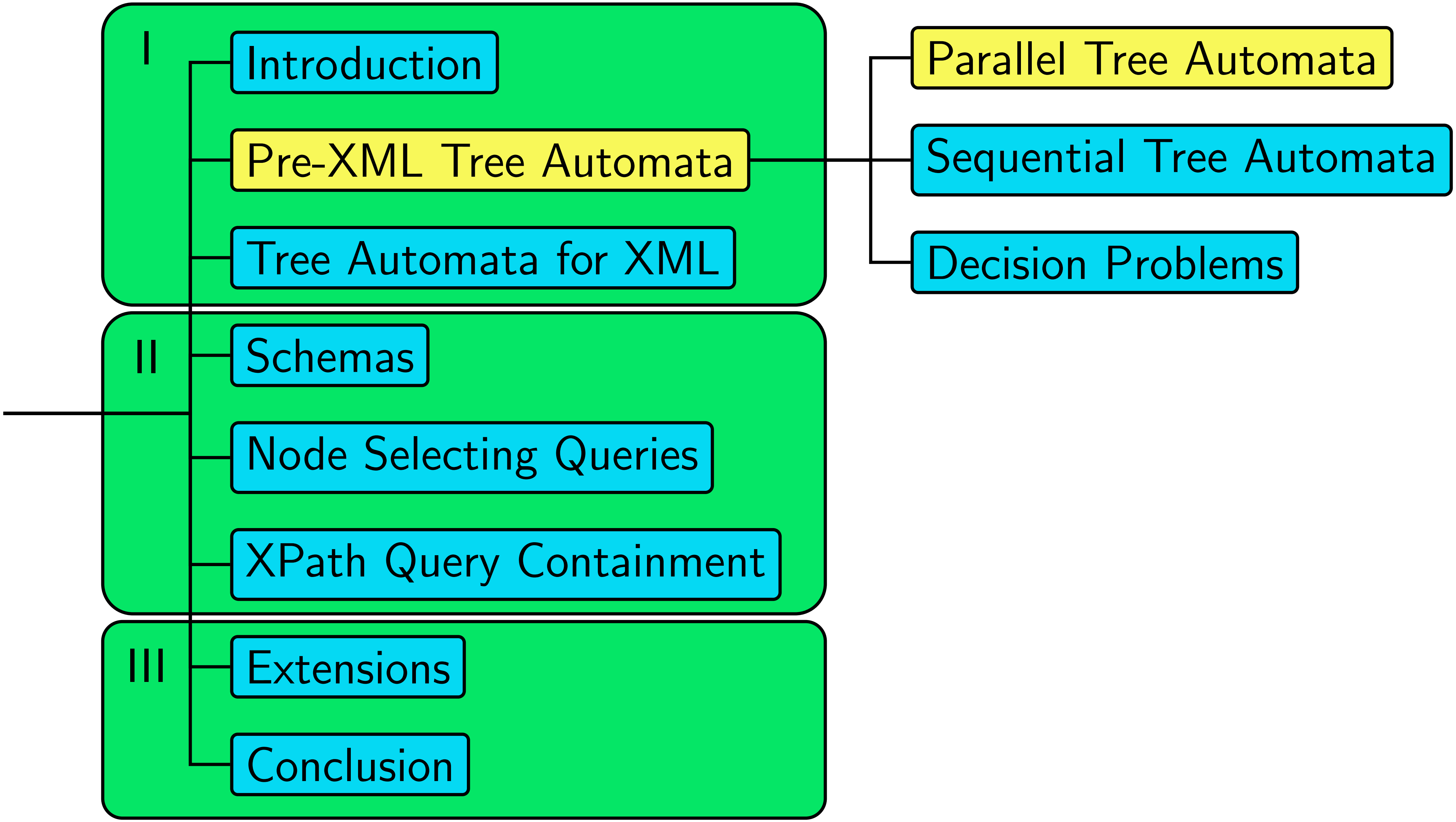
How do automata generalize to trees?

Sequential



Parallel





I

Introduction

Pre-XML Tree Automata

Tree Automata for XML

II

Schemas

Node Selecting Queries

XPath Query Containment

III

Extensions

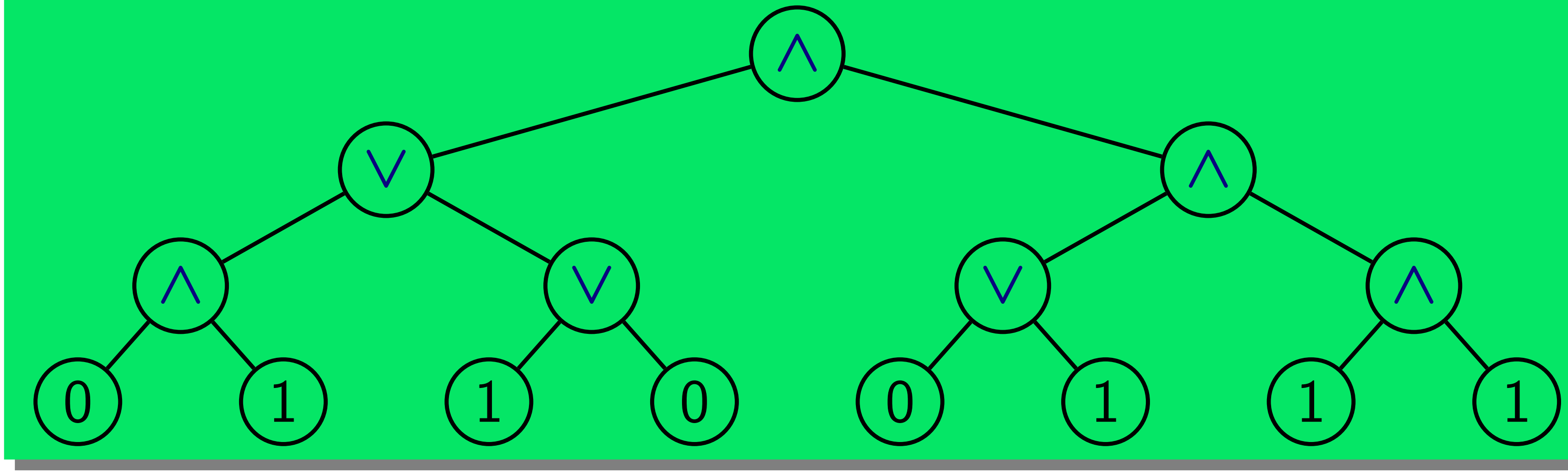
Conclusion

Parallel Tree Automata

Sequential Tree Automata

Decision Problems

Example: Tree-structured Boolean Circuits



Idea

Tree-structured Boolean circuits

Two states: q_0, q_1

Accepting at the root: q_1

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

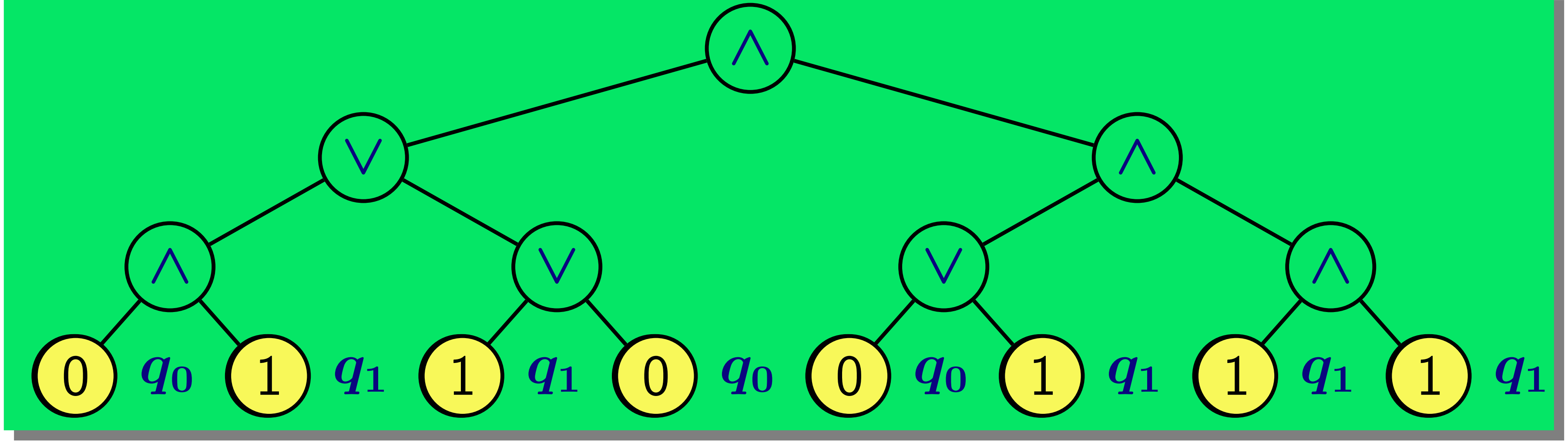
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$

Example: Tree-structured Boolean Circuits



Idea

Tree-structured Boolean circuits

Two states: q_0, q_1

Accepting at the root: q_1

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

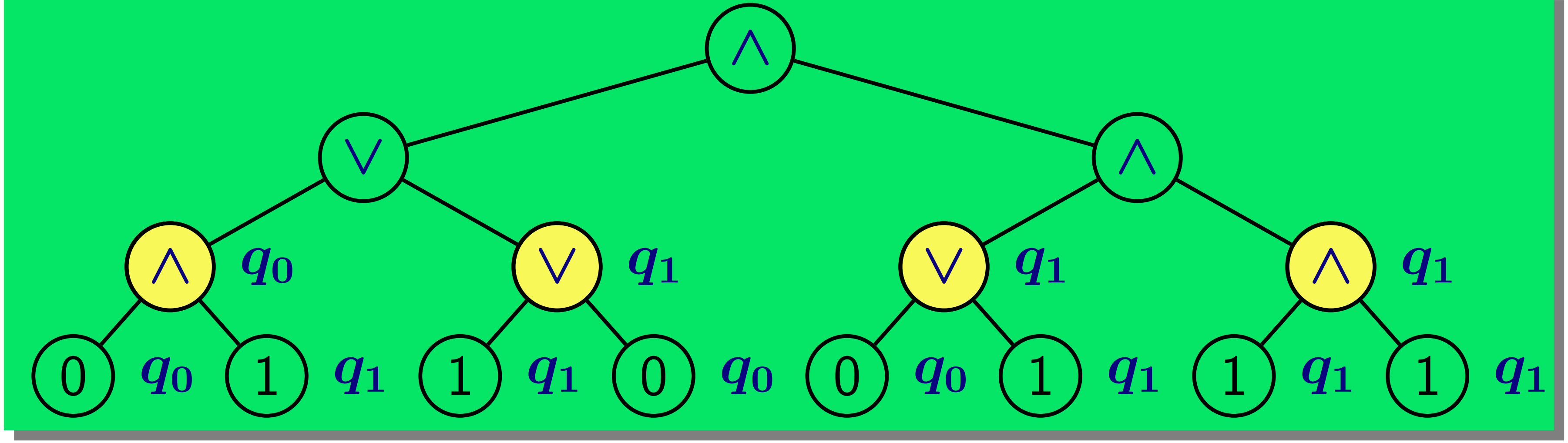
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$

Example: Tree-structured Boolean Circuits



Idea

Tree-structured Boolean circuits

Two states: q_0, q_1

Accepting at the root: q_1

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

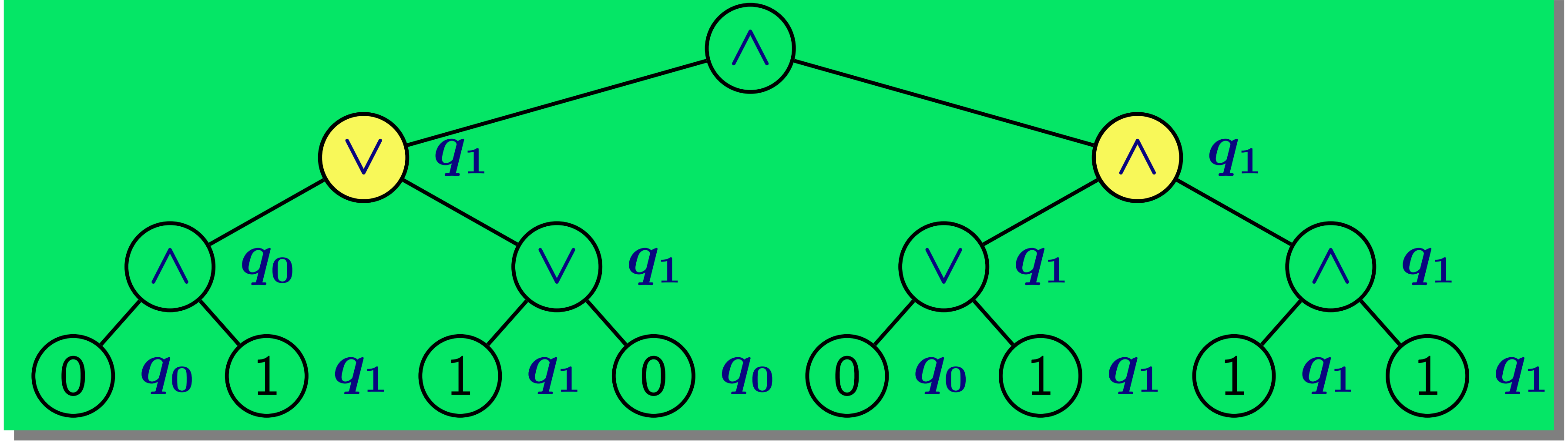
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$

Example: Tree-structured Boolean Circuits



Idea

Tree-structured Boolean circuits

Two states: q_0, q_1

Accepting at the root: q_1

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

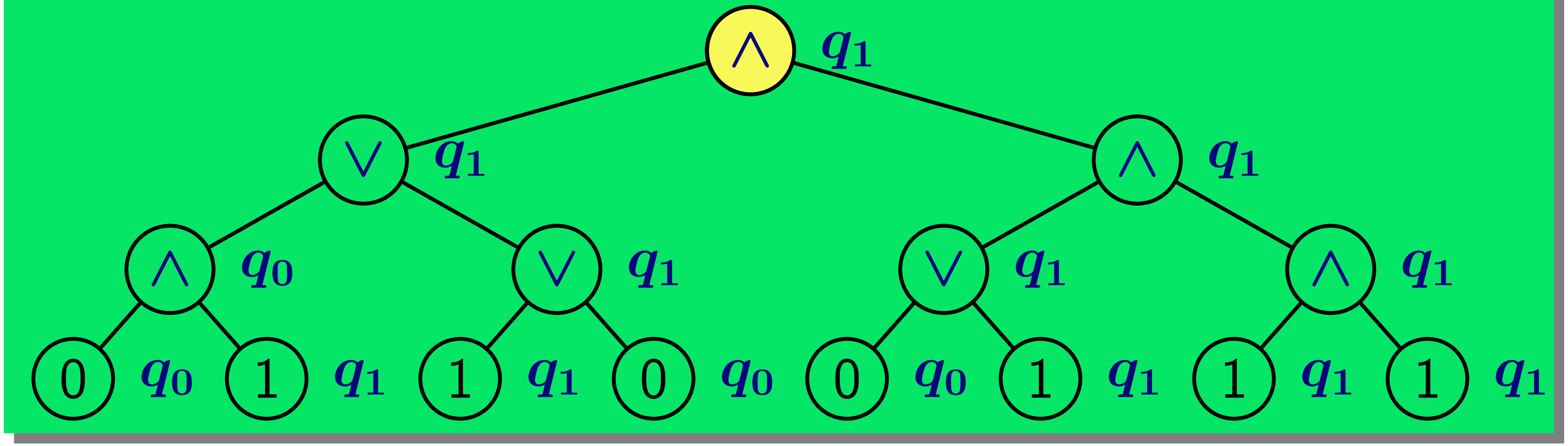
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$

Example: Tree-structured Boolean Circuits



Idea

Tree-structured Boolean circuits

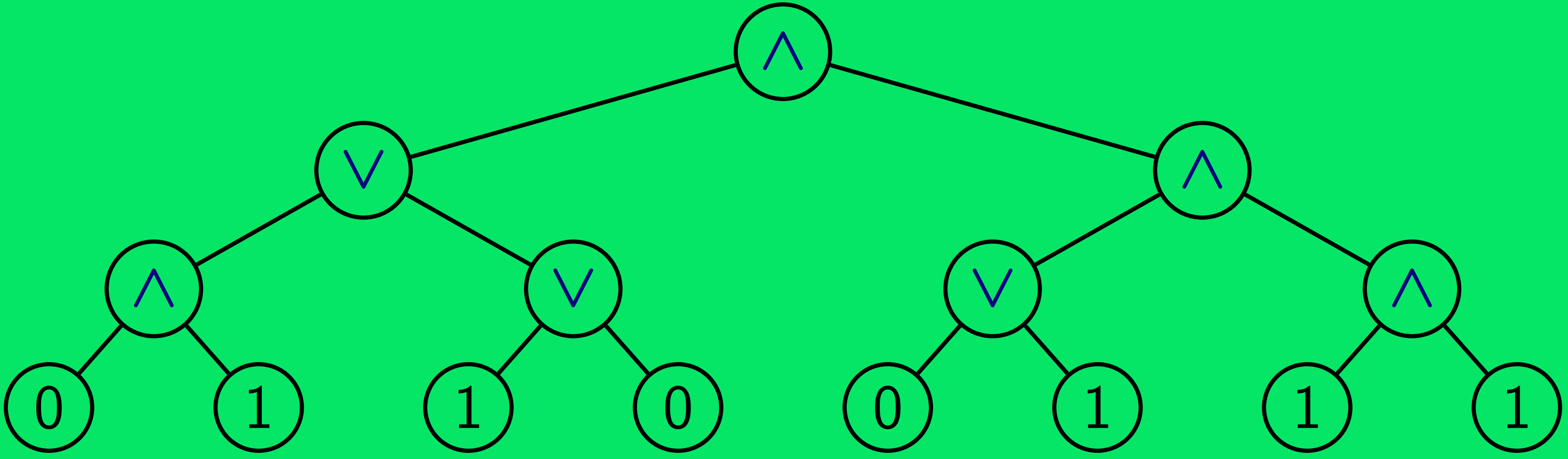
Two states: q_0, q_1

Accepting at the root: q_1

Transitions

$$\delta(\wedge, q_1) = \{(q_1, q_1)\}$$
$$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$$
$$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$$
$$\delta(\vee, q_0) = \{(q_0, q_0)\}$$
$$\delta(0, q_0) = \{\epsilon\}; \delta(0, q_1) = \emptyset$$
$$\delta(1, q_1) = \{\epsilon\}; \delta(1, q_0) = \emptyset$$

Example



Idea

Guess the correct values starting from the root

Check at the leaves

Three states: $q_0, q_1,$ acc

Initial state q_1 at the root

Accepting if all leaves end in acc

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

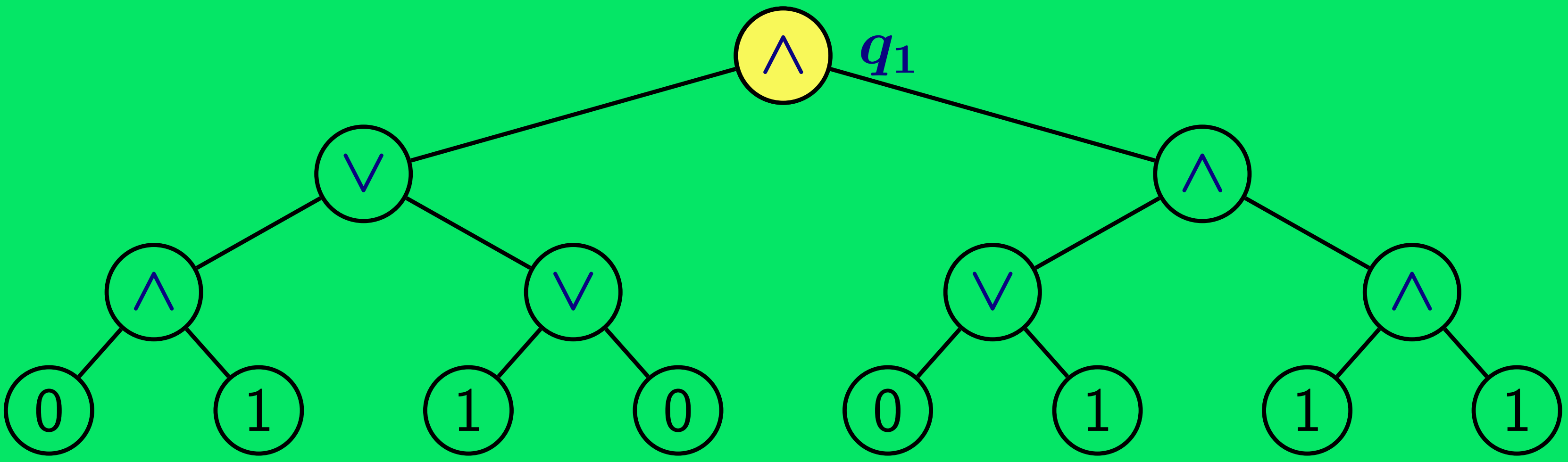
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$

Example



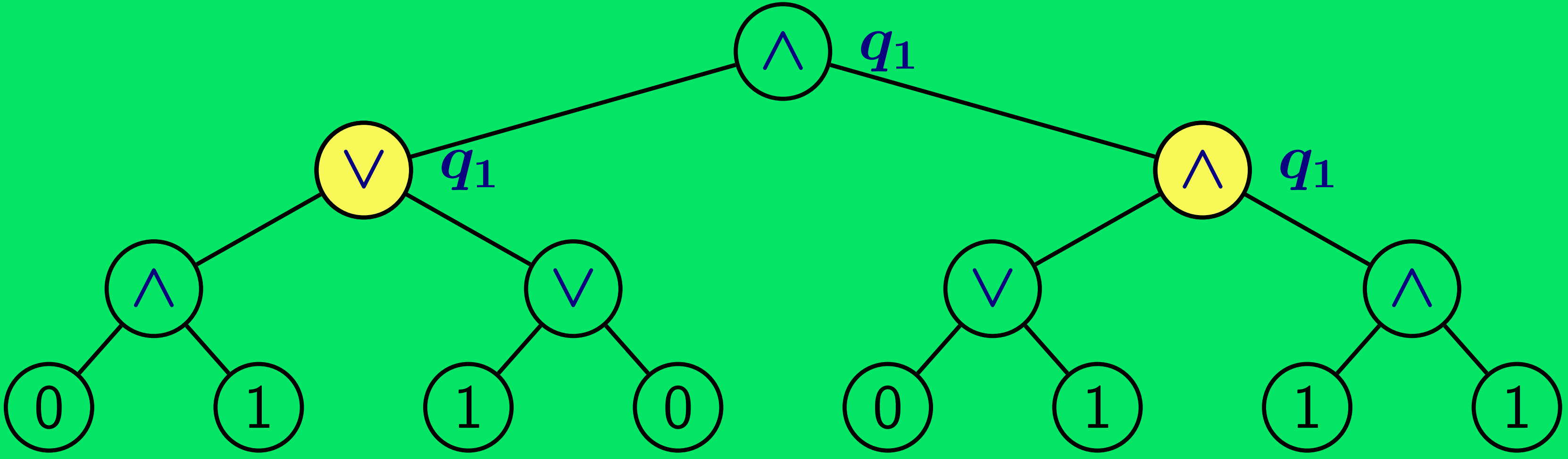
Idea

- Guess the correct values starting from the root
- Check at the leaves
- Three states: q_0, q_1, acc
- Initial state q_1 at the root
- Accepting if all leaves end in acc

Transitions

$$\delta(\wedge, q_1) = \{(q_1, q_1)\}$$
$$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$$
$$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$$
$$\delta(\vee, q_0) = \{(q_0, q_0)\}$$
$$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$$
$$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$$

Example



Idea

Guess the correct values starting from the root

Check at the leaves

Three states: q_0, q_1, acc

Initial state q_1 at the root

Accepting if all leaves end in acc

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

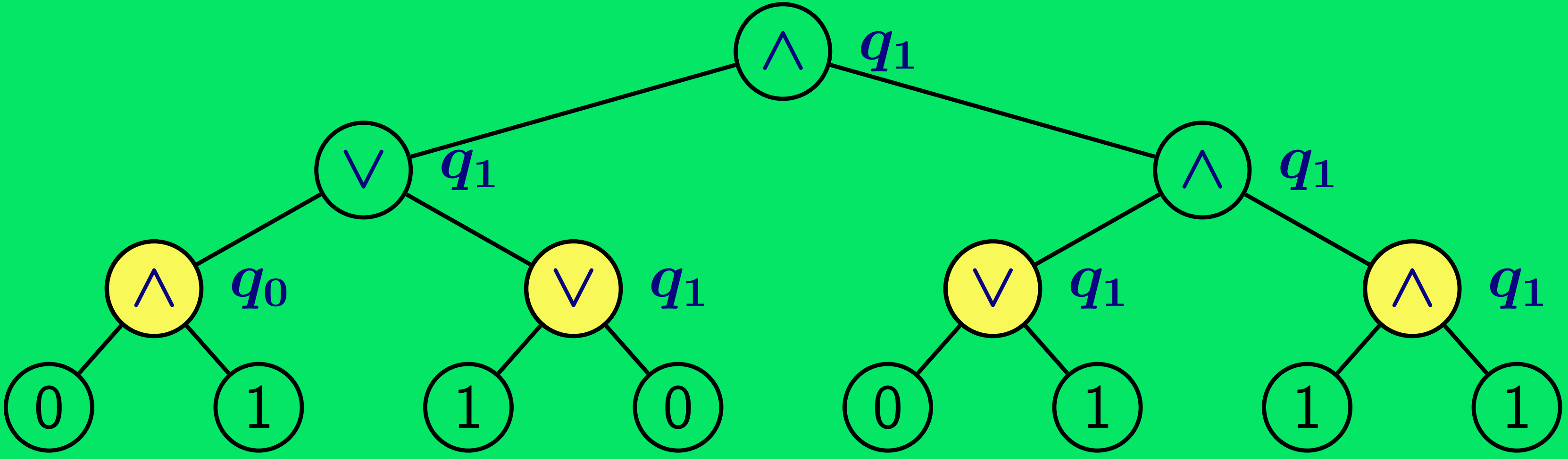
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$

Example



Idea

Guess the correct values starting from the root

Check at the leaves

Three states: q_0, q_1, acc

Initial state q_1 at the root

Accepting if all leaves end in acc

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

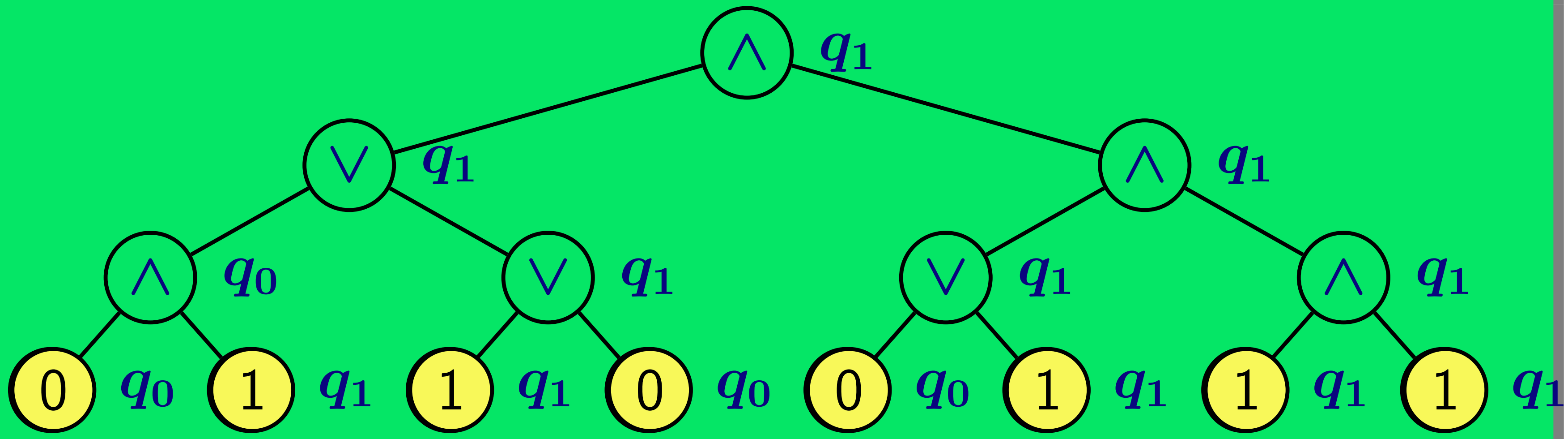
$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$

Example



Idea

Guess the correct values starting from the root

Check at the leaves

Three states: $q_0, q_1,$ acc

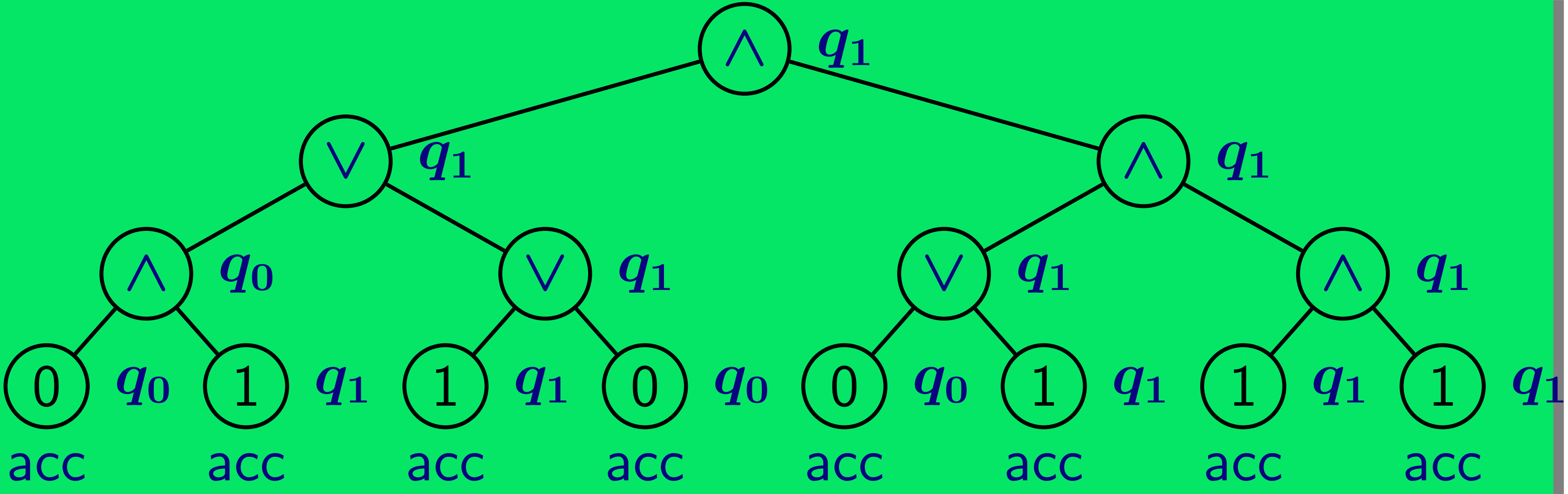
Initial state q_1 at the root

Accepting if all leaves end in acc

Transitions

$$\delta(\wedge, q_1) = \{(q_1, q_1)\}$$
$$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$$
$$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$$
$$\delta(\vee, q_0) = \{(q_0, q_0)\}$$
$$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$$
$$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$$

Example



Idea

Guess the correct values starting from the root

Check at the leaves

Three states: $q_0, q_1,$ acc

Initial state q_1 at the root

Accepting if all leaves end in acc

Transitions

$\delta(\wedge, q_1) = \{(q_1, q_1)\}$

$\delta(\wedge, q_0) = \{(q_0, q_1), (q_1, q_0), (q_0, q_0)\}$

$\delta(\vee, q_1) = \{(q_0, q_1), (q_1, q_0), (q_1, q_1)\}$

$\delta(\vee, q_0) = \{(q_0, q_0)\}$

$\delta(0, q_0) = \{\text{acc}\}; \delta(0, q_1) = \emptyset$

$\delta(1, q_1) = \{\text{acc}\}; \delta(1, q_0) = \emptyset$

Definition

A bottom-up automaton is **deterministic** if
for each a and $p \neq q$: $\delta(a,p) \cap \delta(a,q) = \emptyset$

Theorem

The following are equivalent for a tree language L:

(a) L is accepted by a nondeterministic bottom-up automaton

(b) L is accepted by a deterministic bottom-up automaton

(c) L is accepted by a nondeterministic top-down automaton

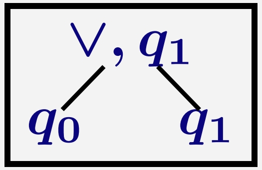
Proof Idea

(a) $\Longrightarrow$ (b): Powerset construction

(a) $\Longleftrightarrow$ (c): Just the same thing, viewed in two different ways

Observation

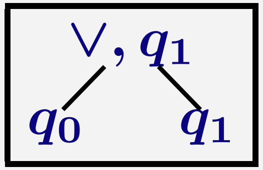
- $(q_0, q_1) \in \delta(\vee, q_1)$ can be interpreted as an allowed pattern:



- A tree is accepted, iff there is a labelling with states such that
  - all local patterns are allowed
  - the root is labelled with q_1

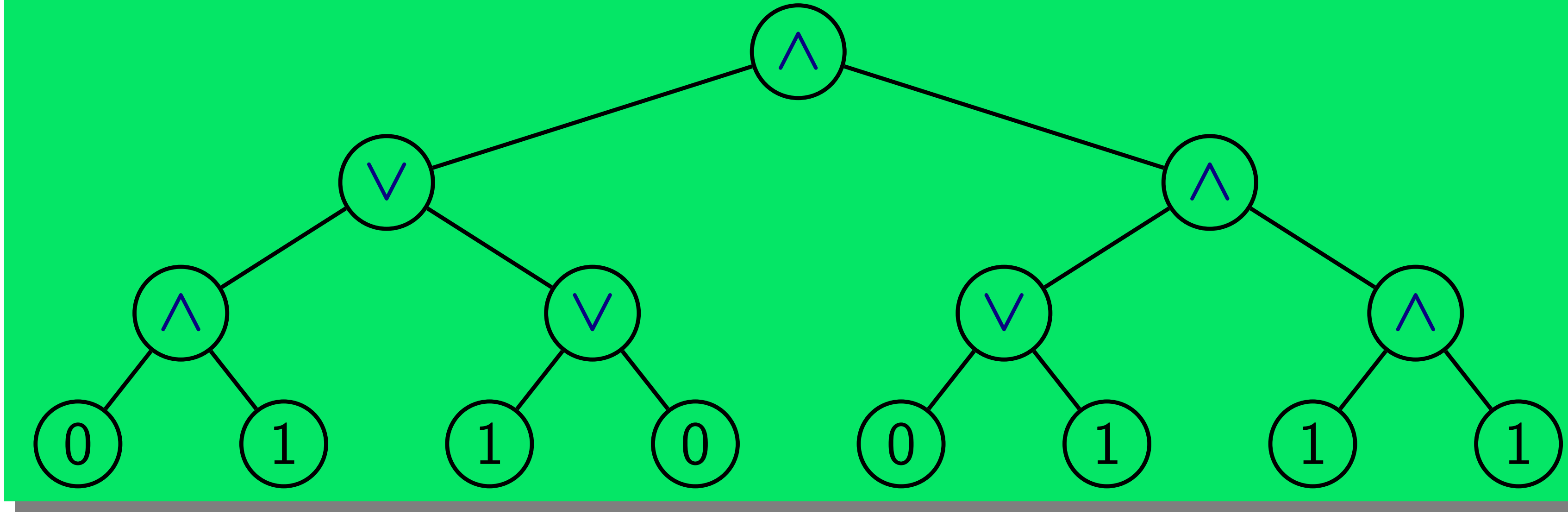
Observation

- $(q_0, q_1) \in \delta(\vee, q_1)$ can be interpreted as an allowed pattern:

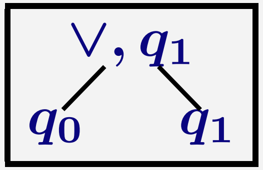


- A tree is accepted, iff there is a labelling with states such that
  - all local patterns are allowed
  - the root is labelled with q_1

Example

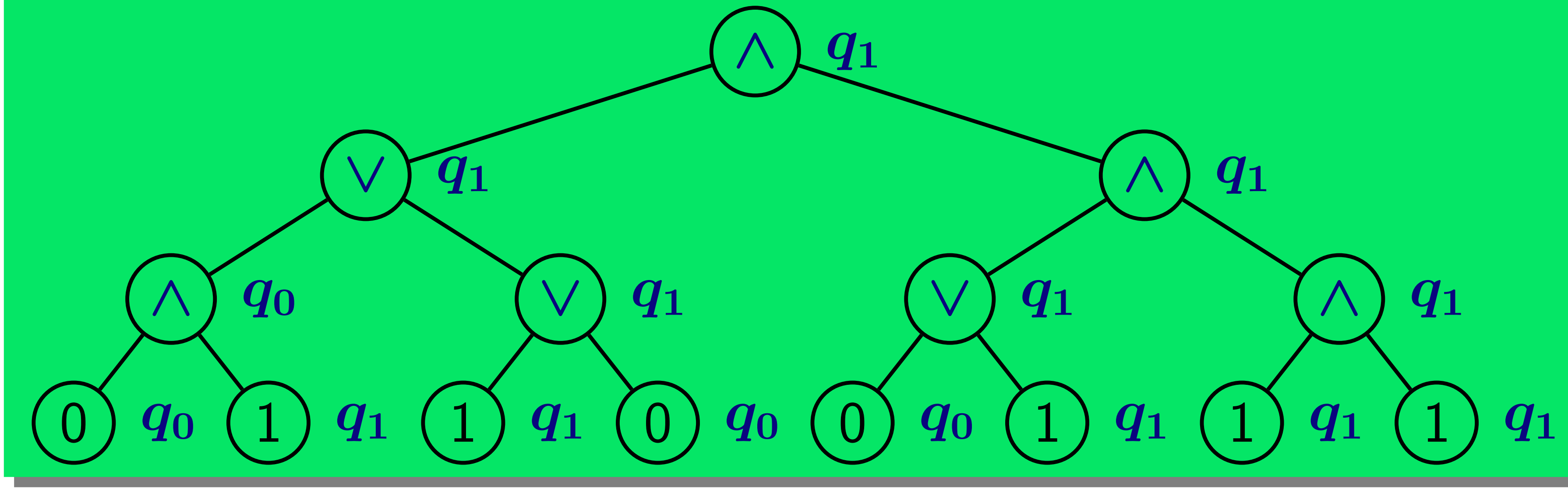


Observation



- $(q_0, q_1) \in \delta(\vee, q_1)$ can be interpreted as an allowed pattern:
- A tree is accepted, iff there is a labelling with states such that
  - all local patterns are allowed
  - the root is labelled with q_1

Example



Definition (MSO logic)

- **Formulas** talk about
  - edges of the tree (E)
  - node labels ($Q_0, Q_1, Q_\wedge, Q_\vee$)
  - the root of the tree (root)
- **First-order-variables** represent nodes
- **Monadic second-order** (MSO) variables represent sets of nodes

Example: Boolean Circuits

$$\begin{aligned} \text{Boolean circuit true} \quad \equiv \quad & \exists X\; X(\text{root}) \wedge \;\forall x \\ & (Q_0(x) \rightarrow \neg X(x)) \wedge \\ & ((Q_\wedge(x) \wedge X(x)) \rightarrow (\forall y[E(x,y) \rightarrow X(y)])) \wedge \\ & ((Q_\vee(x) \wedge X(x)) \rightarrow (\exists y[E(x,y) \wedge X(y)])) \end{aligned}$$

Theorem (Doner 1970; Thatcher, Wright 1968)

MSO $\equiv$ Regular Tree Languages

Theorem

MSO $\equiv$ Regular Tree Languages

Proof Idea

Automata $\Rightarrow$ **MSO**:

Formula expresses that there exists a correct tiling

MSO $\Rightarrow$ **Automata**: more involved

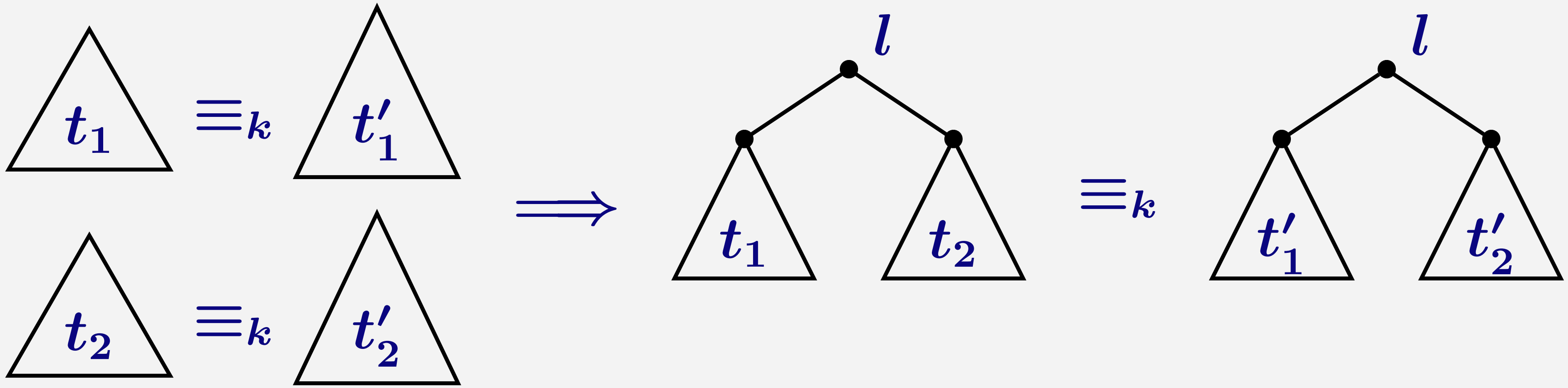
Basic idea:

Automaton computes for each node v the set of formulas which hold in the subtree rooted at v

Formula $\Rightarrow$ automaton

- Let φ be an MSO-formula, $k :=$ quantifier-depth of φ
- **k-type** of a tree $t :=$ (essentially)
  set of MSO-formulas ψ of quantifier-depth $\leq k$ which hold in t
- **$t_1 \equiv_k t_2$**: $k\text{-type}(t_1) = k\text{-type}(t_2)$
- Automaton computes k-type of tree and concludes whether φ holds

Crucial fact



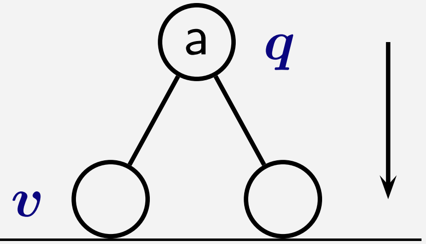
Question

What is the right notion for deterministic top-down automata?

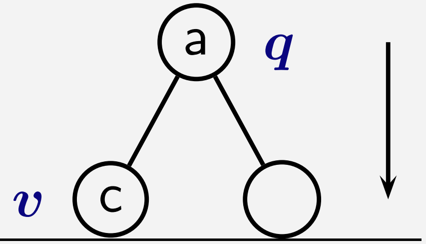
3 Possibilities

State at a node v might depend on

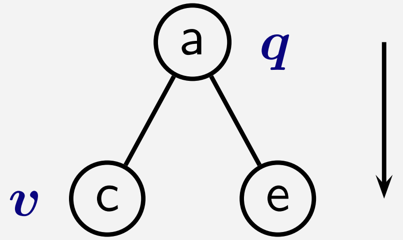
state and symbol of parent



state and symbol of parent and symbol of v



state and symbol of parent and symbols at v and its sibling



Question

What is a good acceptance mechanism for deterministic top-down automata?

Several possibilitites

(1) At all leaves states have to be accepting

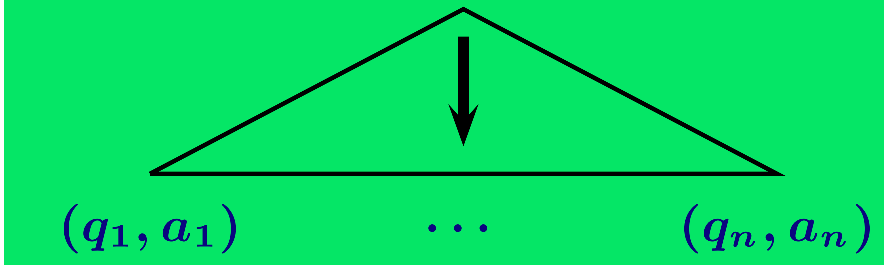
(2) There is a leave with an accepting state

(2) is problematic for complement and intersection
(1) is problematic for complement and union

Definition (Root-to-frontier automata with regular acceptance condition)

- Tree automata $\mathcal{A}$ are equipped with an additional regular string language L over $Q \times \Sigma$
- $\mathcal{A}$ accepts t if the (state,symbol)-string at the leaves (from left to right) is in L

Illustration

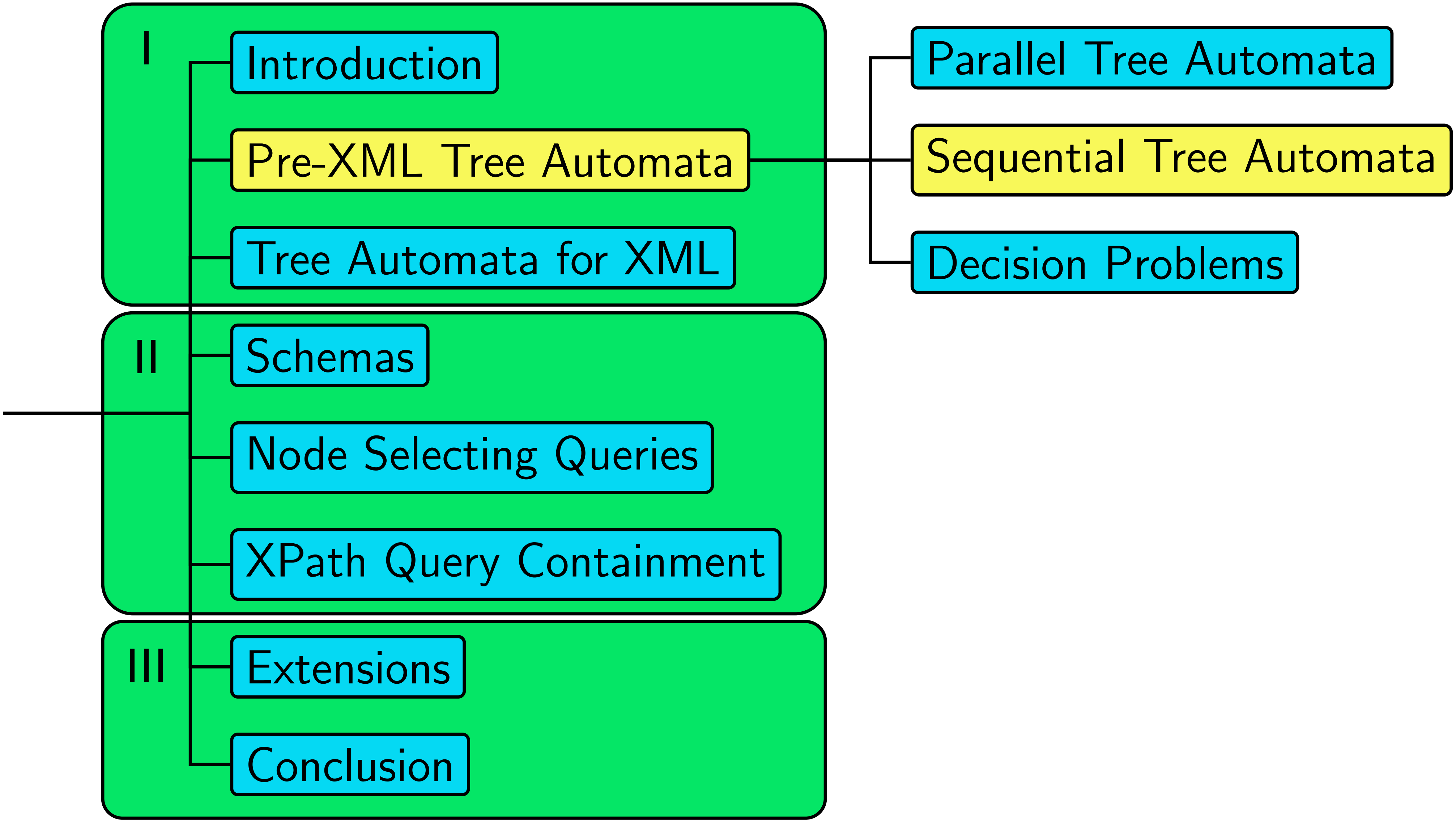


A robust class

- The resulting class is closed under Boolean operations
- Good algorithmic properties
- Does not capture all regular tree languages

Regular tree languages

- Regular tree languages are a robust class
- Characterized by
  - parallel tree automata
  - MSO logic
  - several other models
- They are the natural analog of regular string languages
- Deterministic top-down automata with regular acceptance conditions define a weaker but also robust class


I

Introduction

Pre-XML Tree Automata

Tree Automata for XML

II

Schemas

Node Selecting Queries

XPath Query Containment

III

Extensions

Conclusion

Parallel Tree Automata

Sequential Tree Automata

Decision Problems

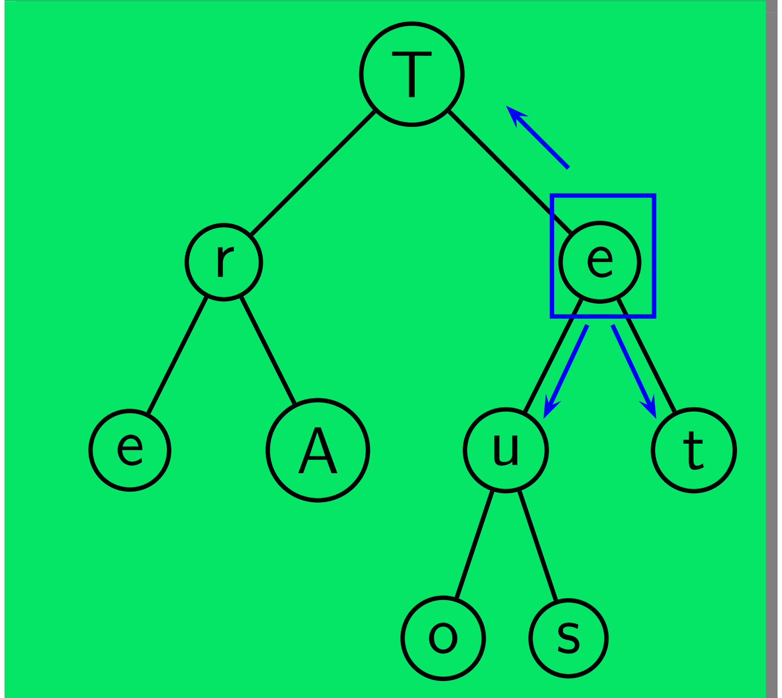
Definition (Tree-walk automata)

Depending on

- current state
- symbol of current node
- position of current node wrt its siblings

the automaton moves to a neighbor and takes a new state

TWA



Question

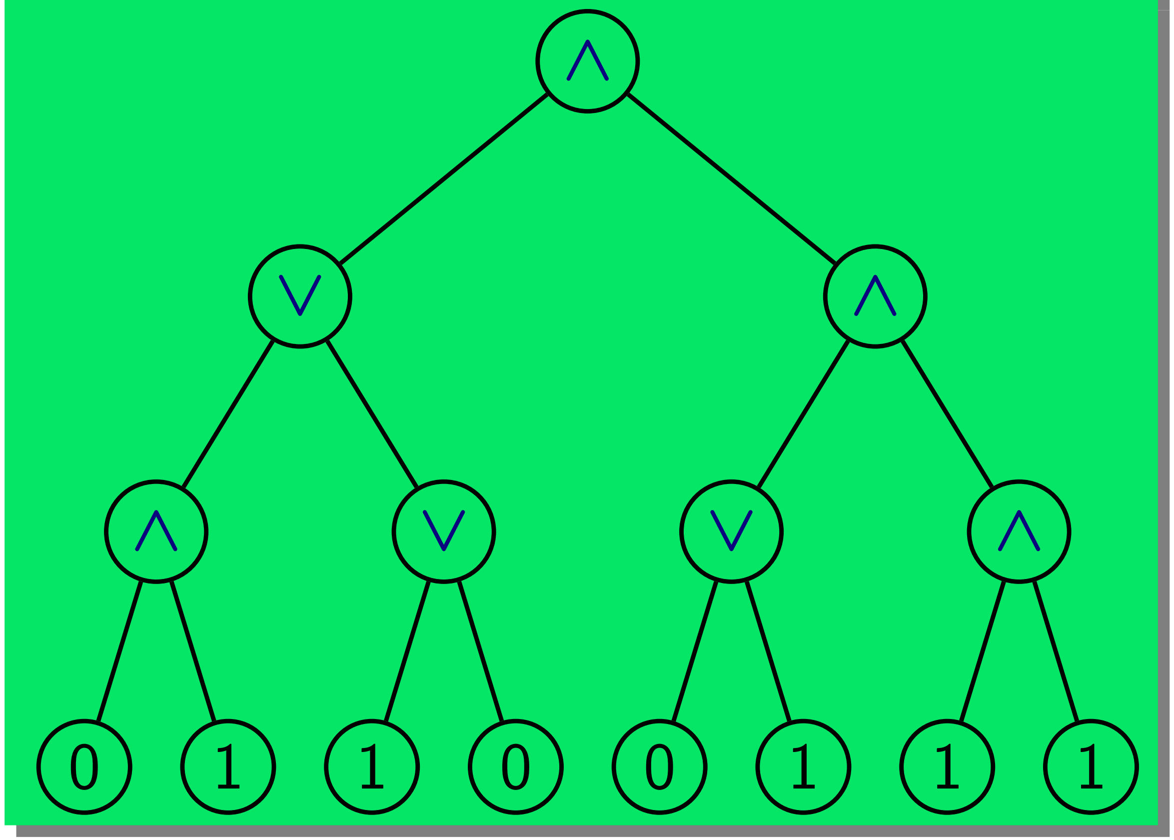
What is the expressive power of tree-walk automata?

Tree-Walk Automata (cont.)

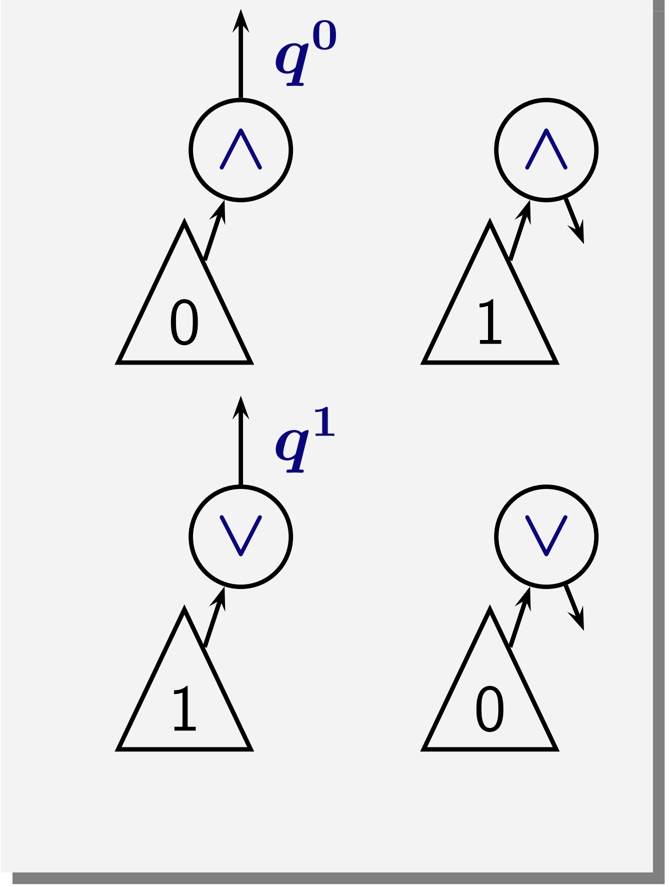
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



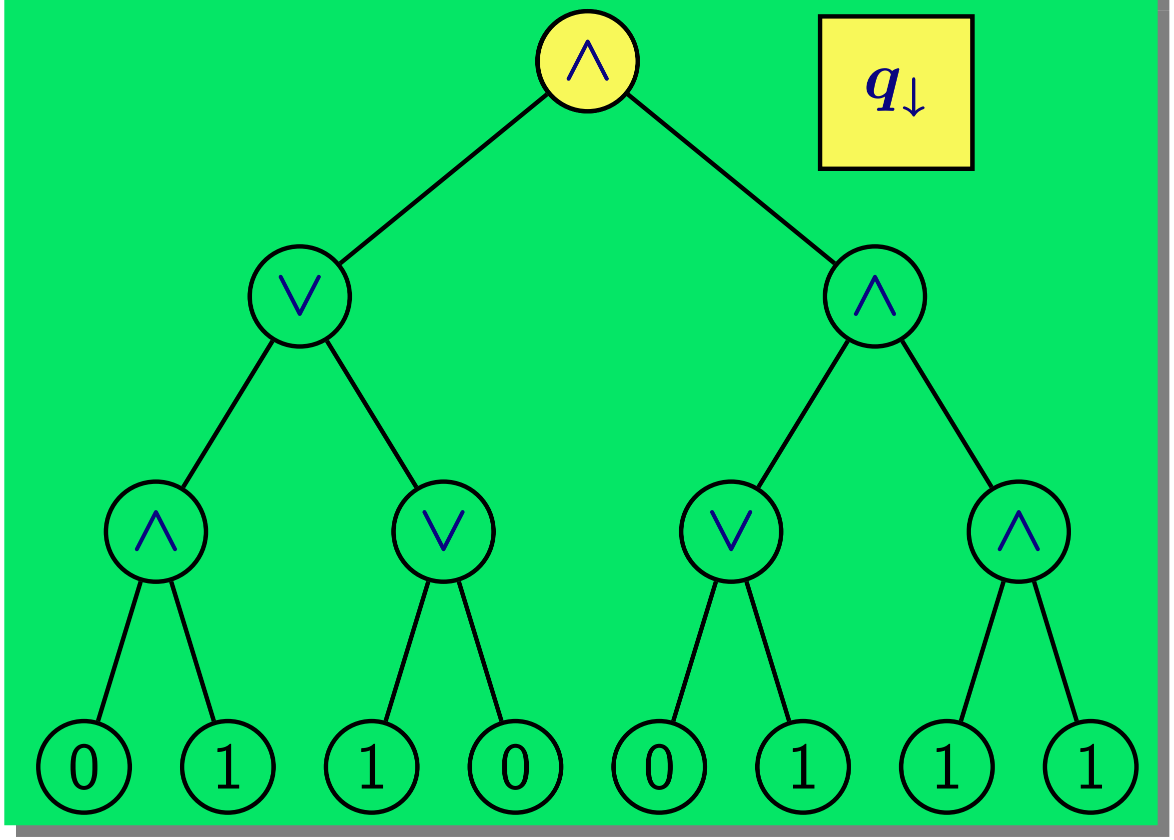
Idea



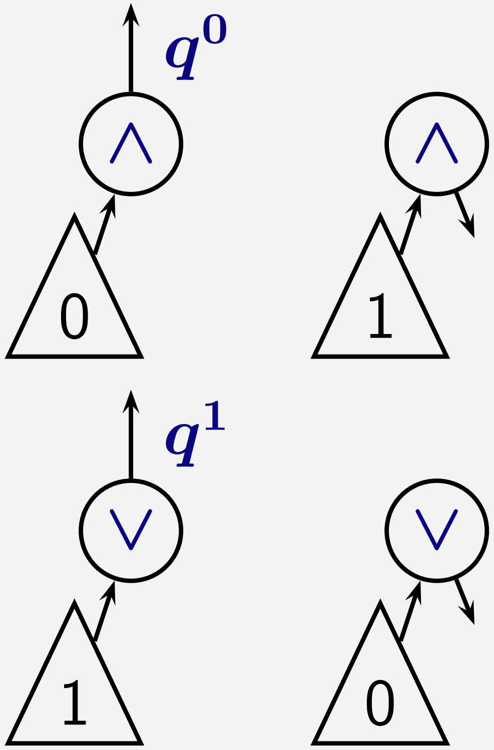
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



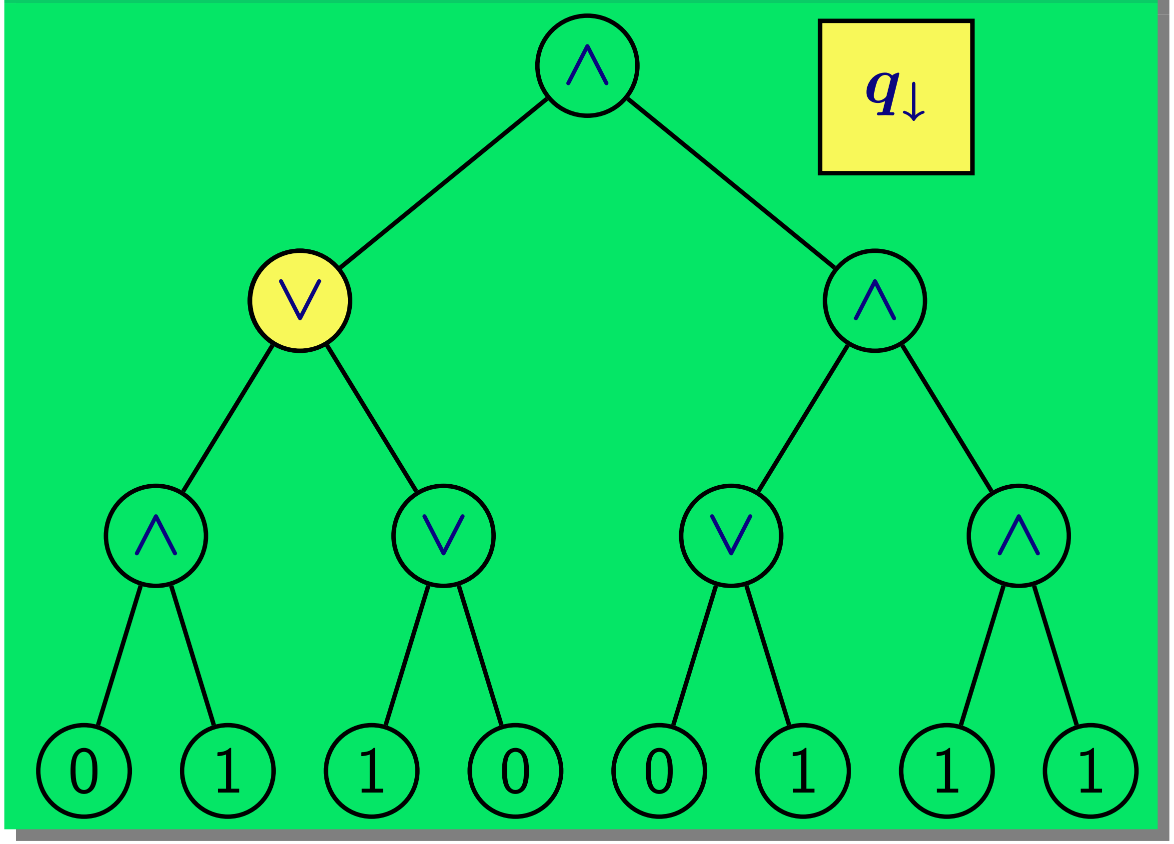
Idea



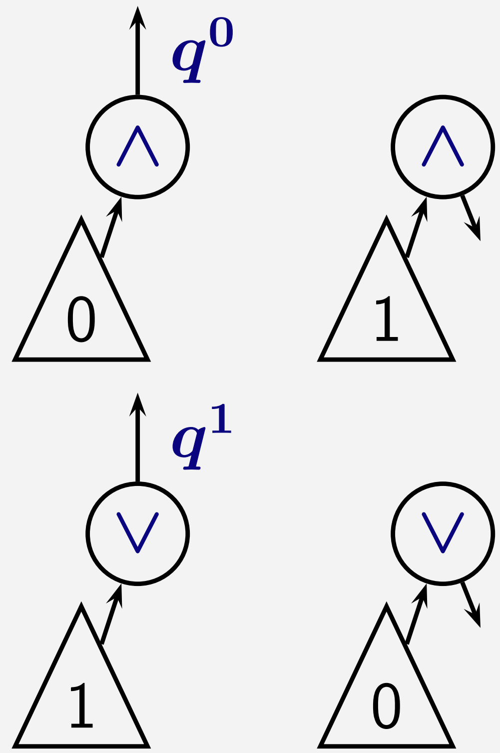
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



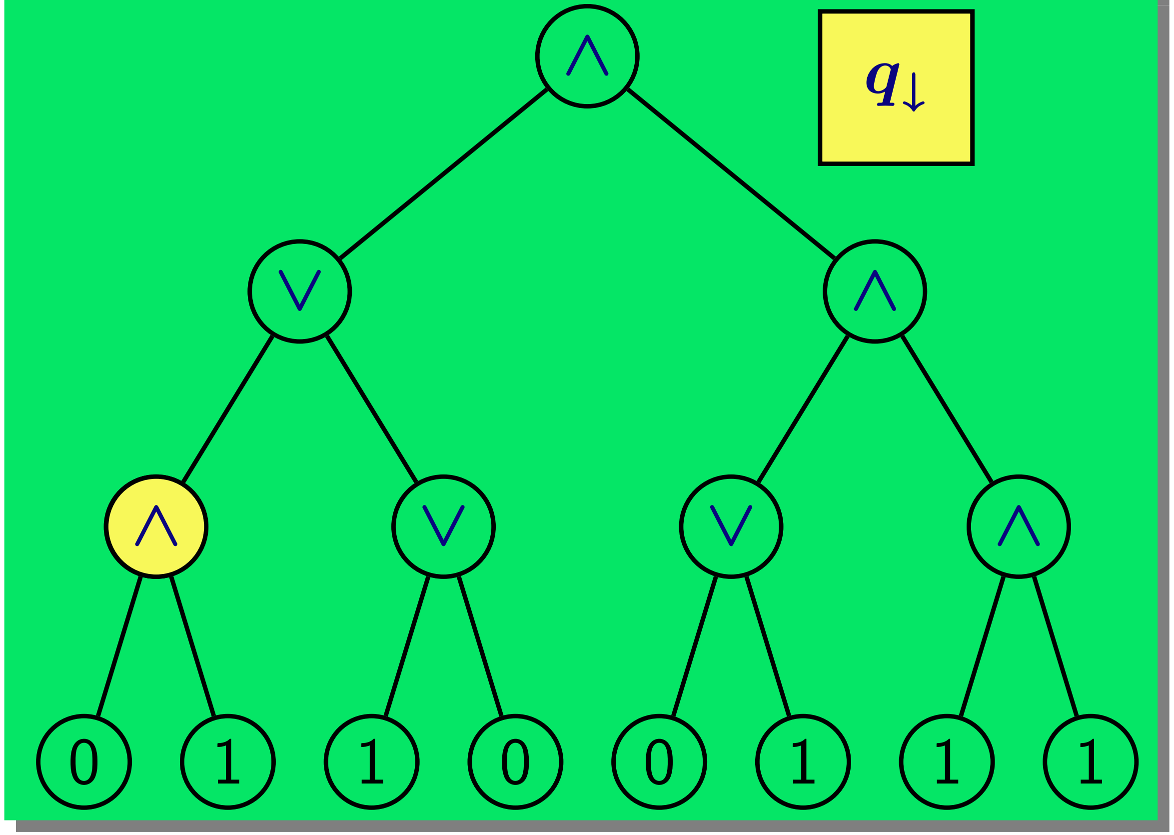
Idea



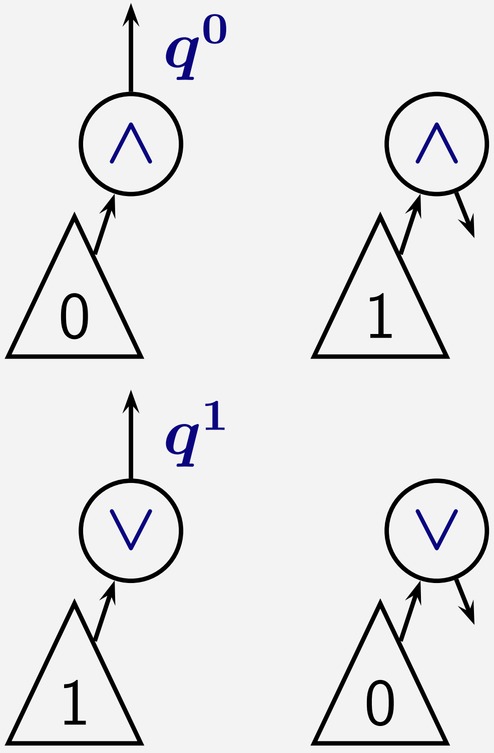
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



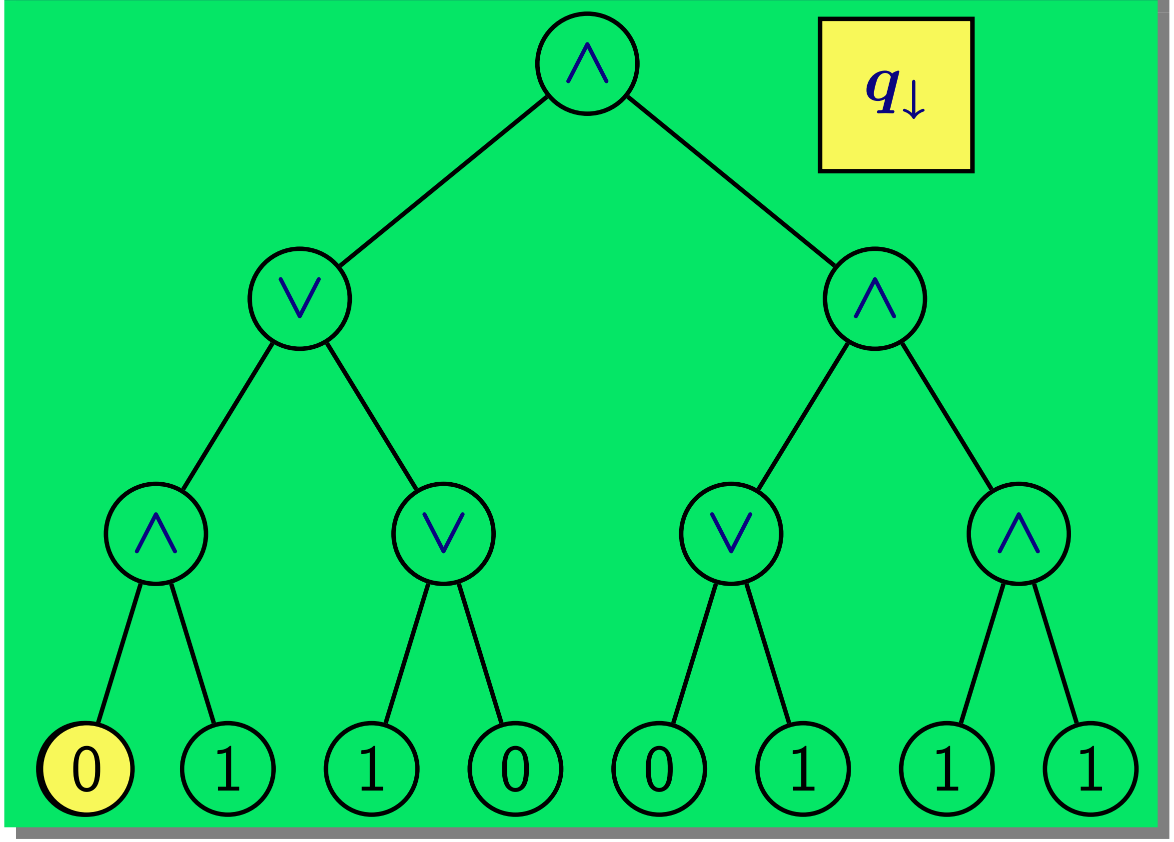
Idea



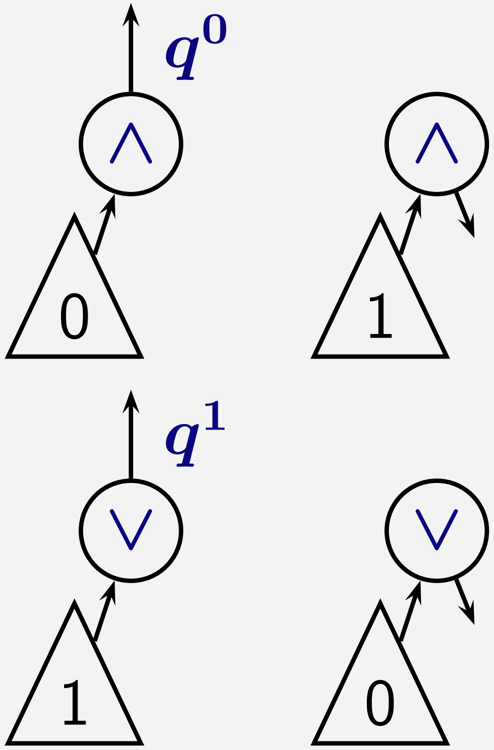
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



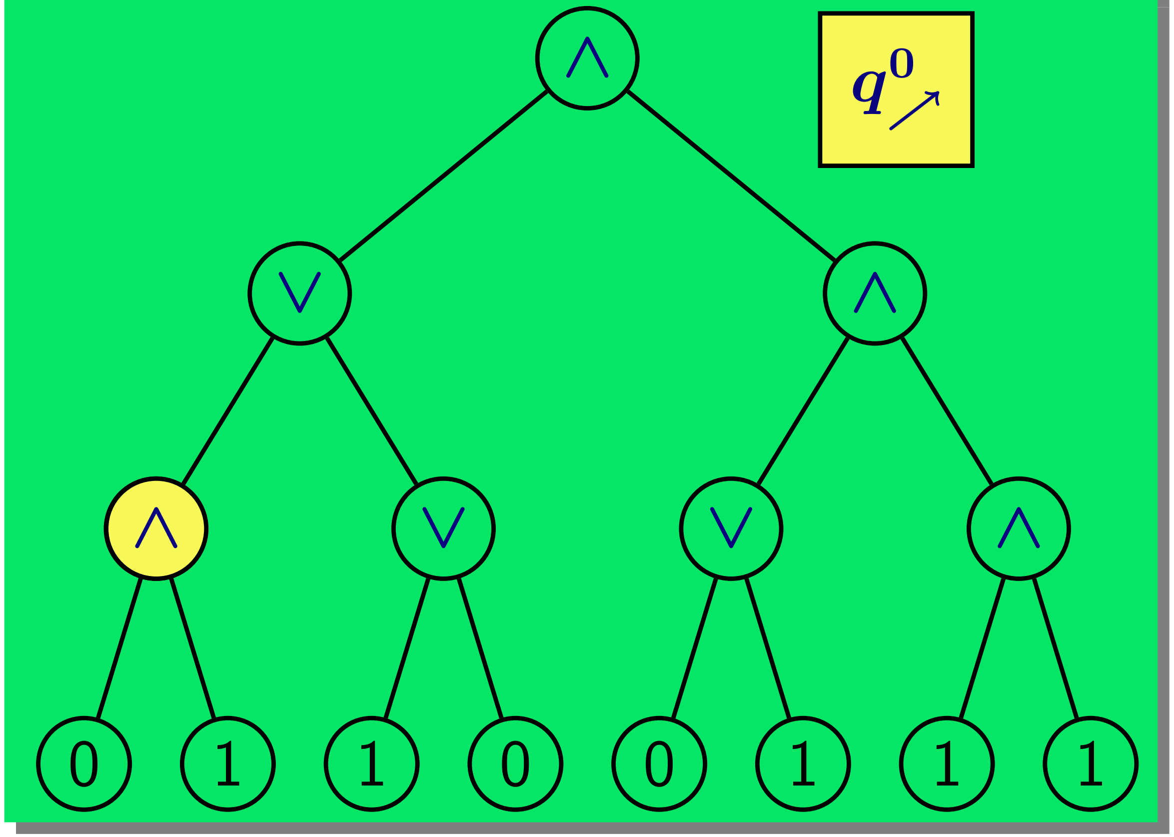
Idea



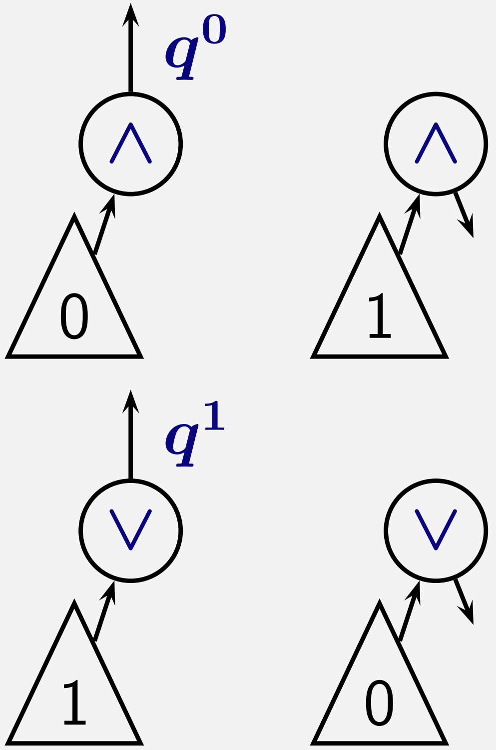
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



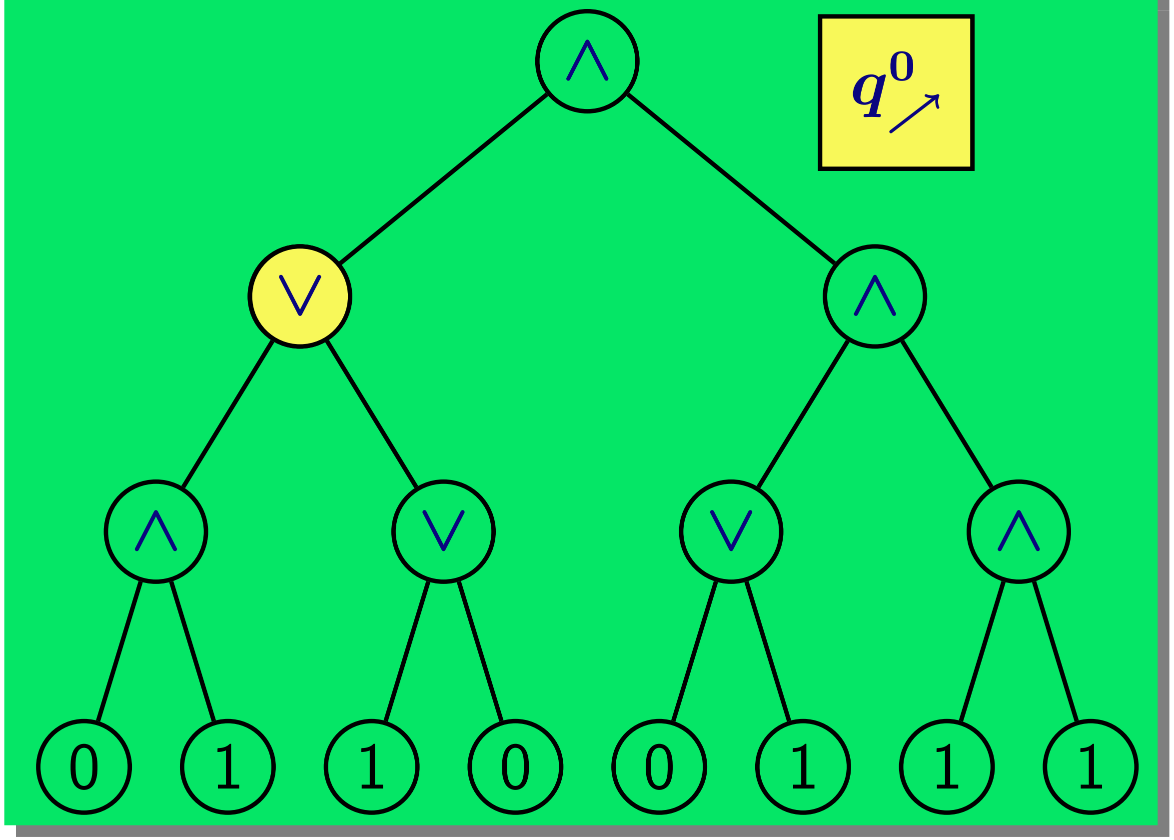
Idea



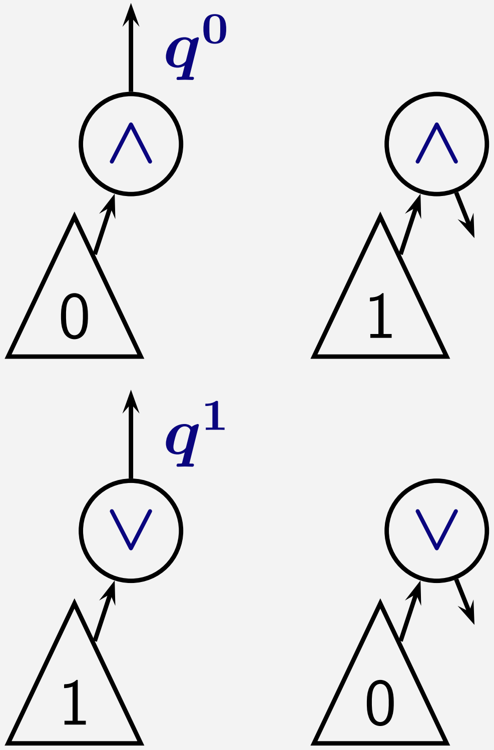
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



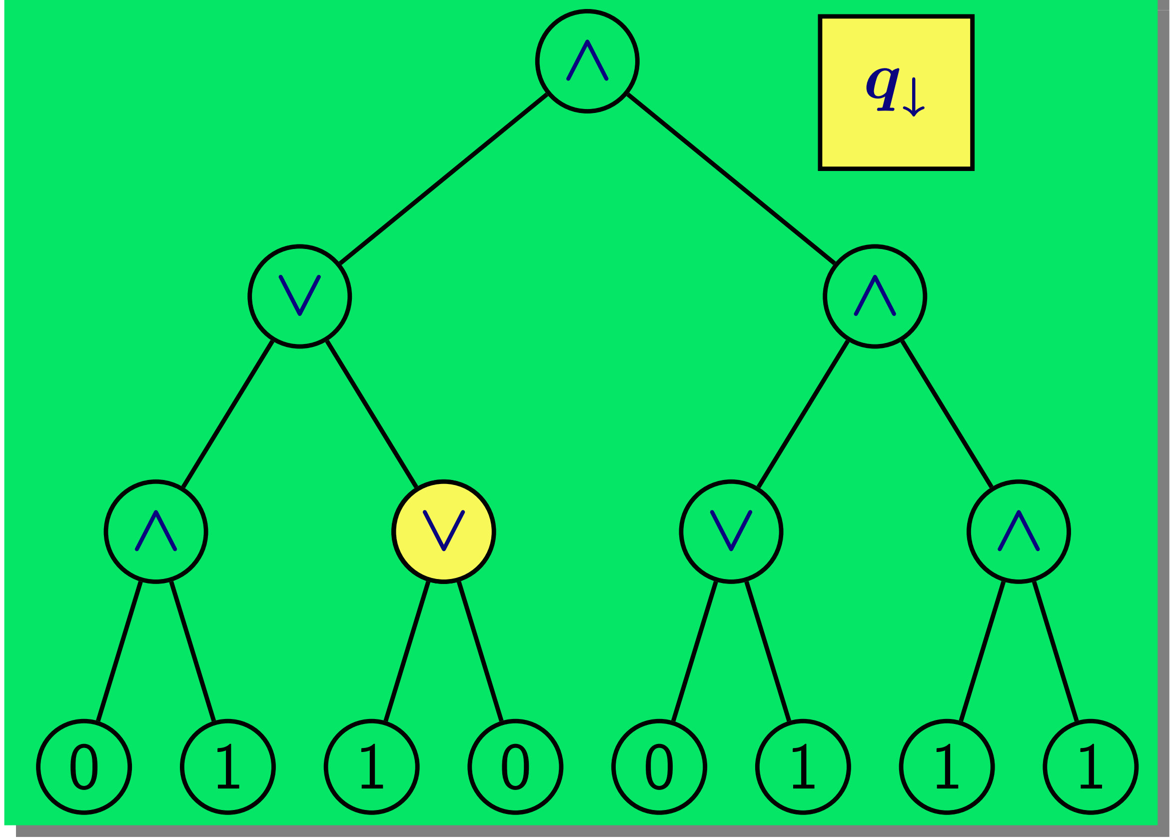
Idea



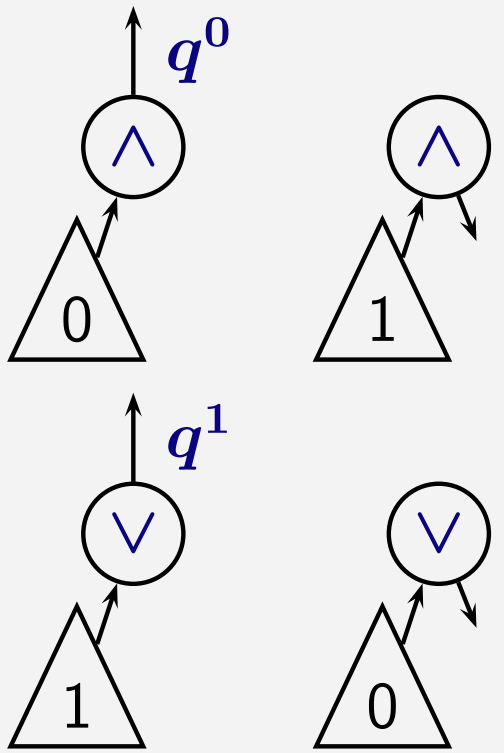
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



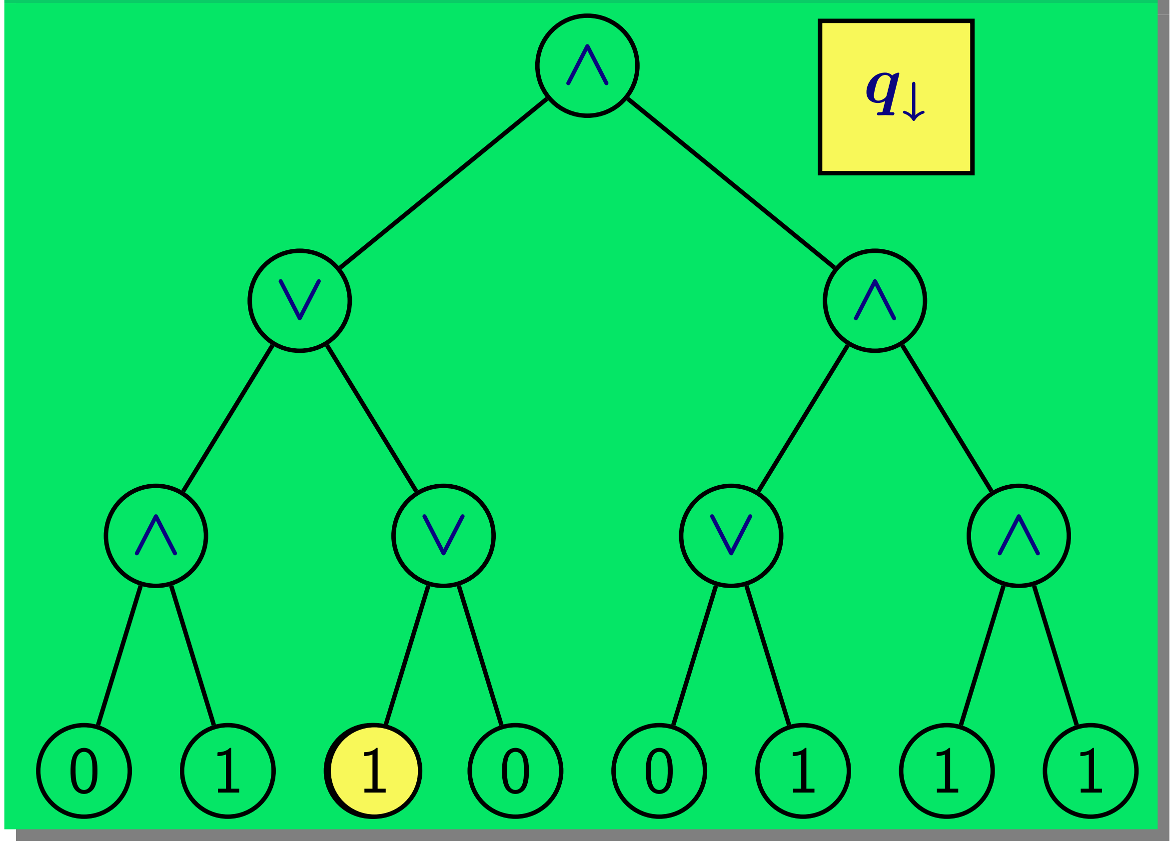
Idea



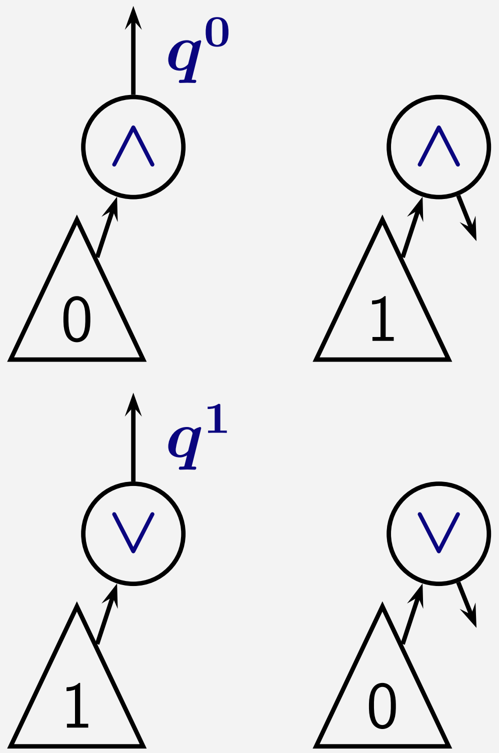
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



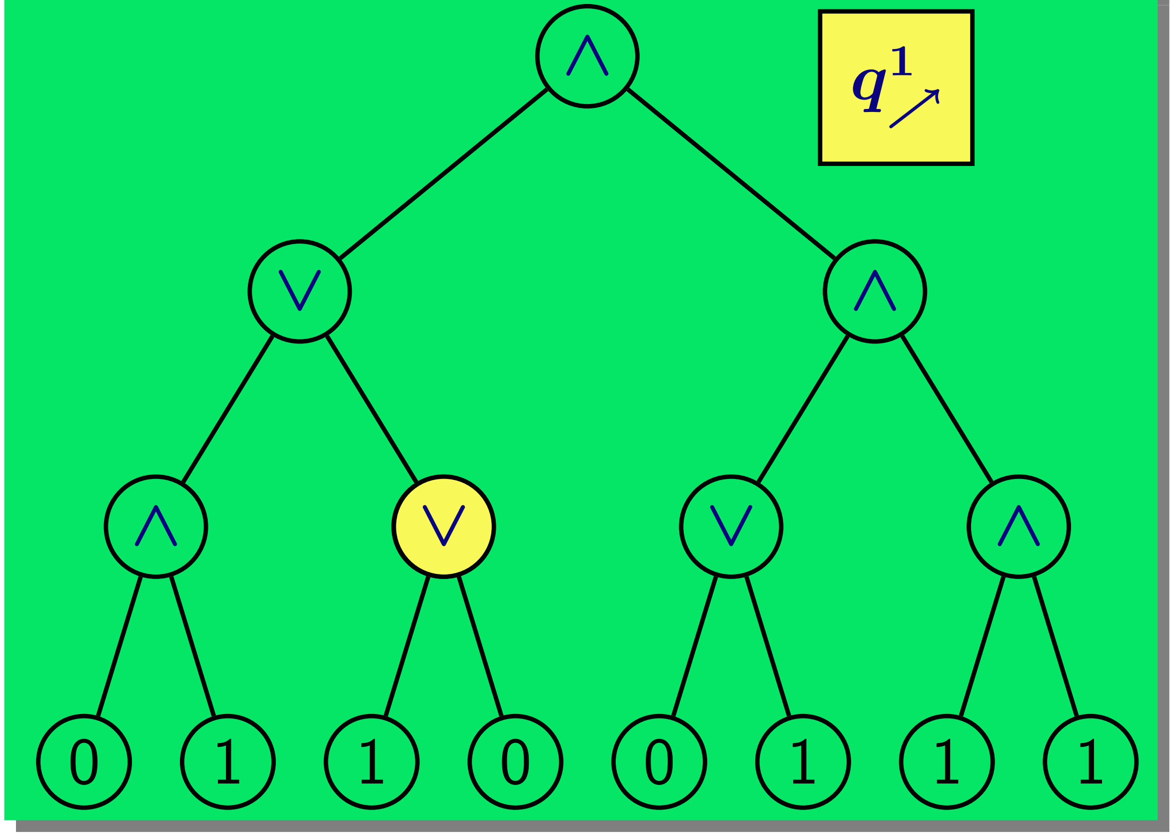
Idea



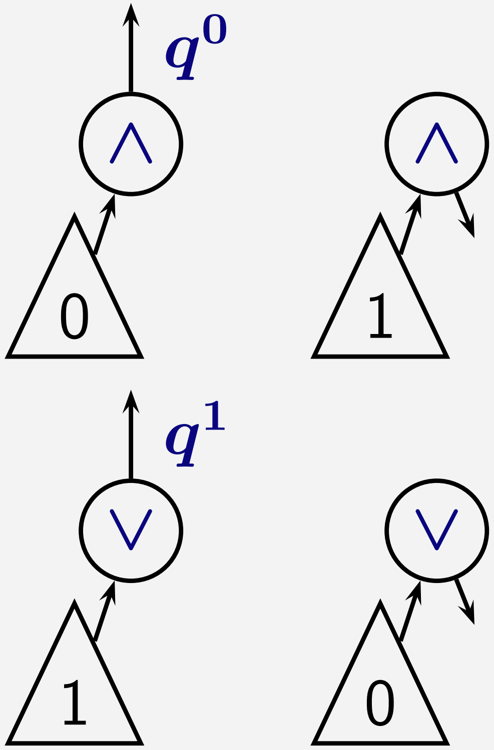
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea

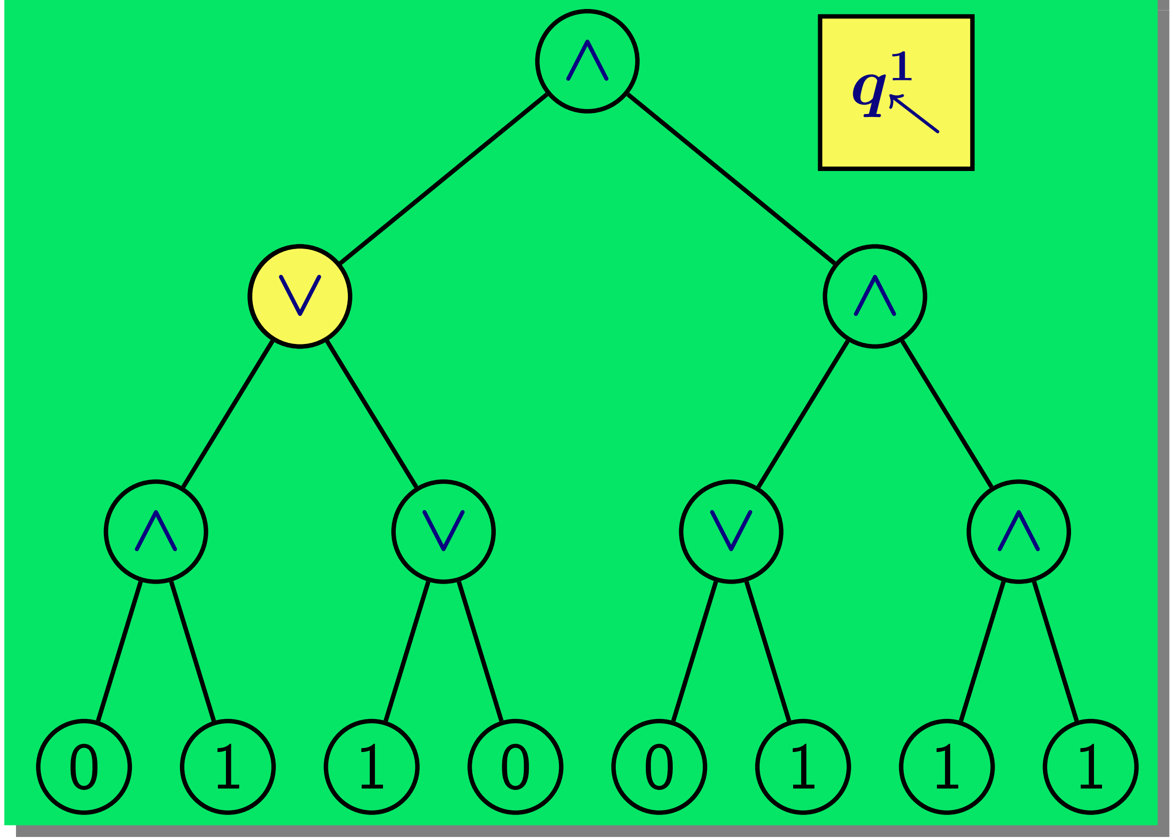


Tree-Walk Automata (cont.)

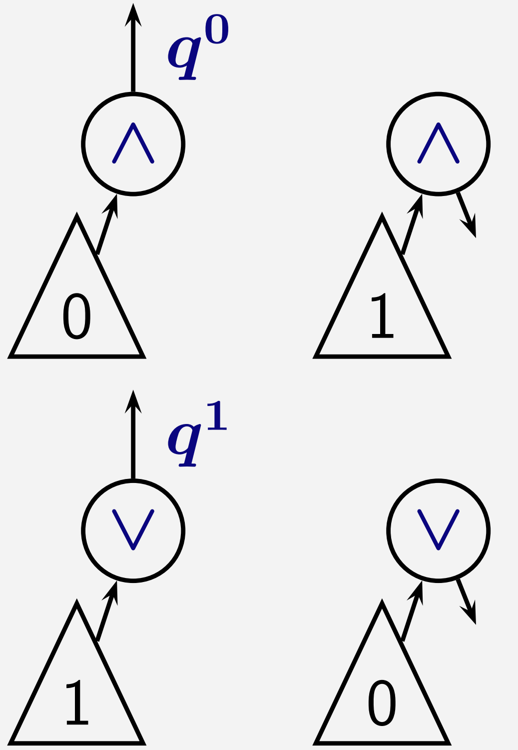
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea

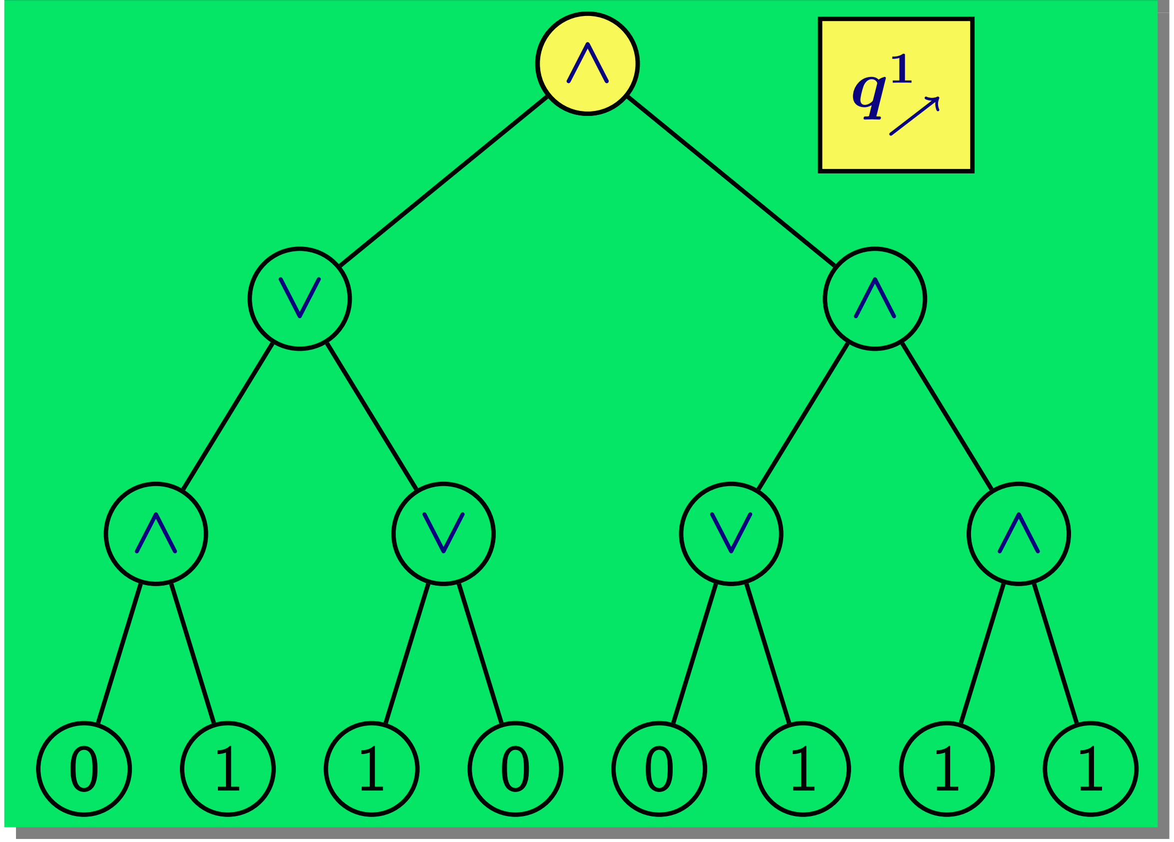


Tree-Walk Automata (cont.)

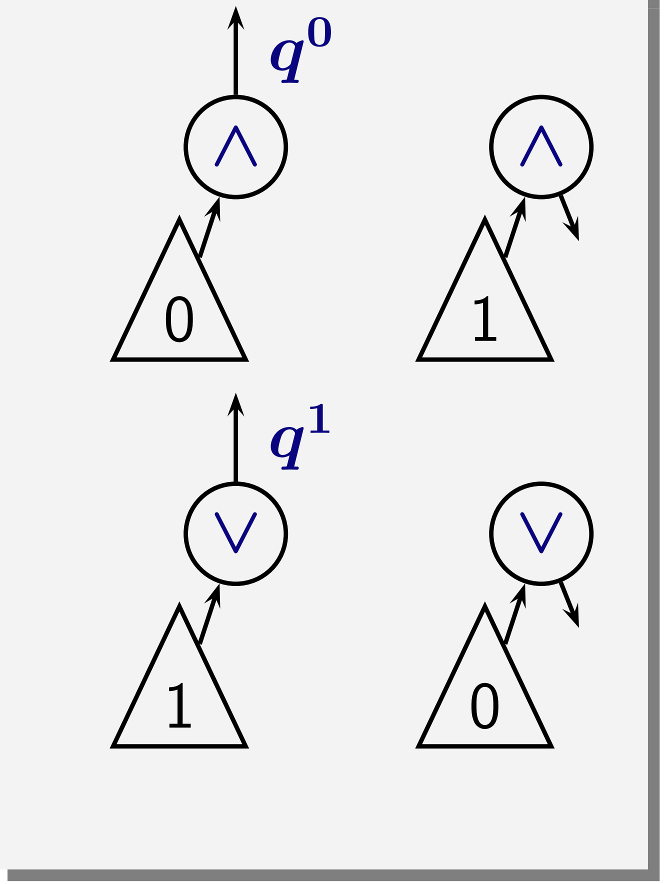
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



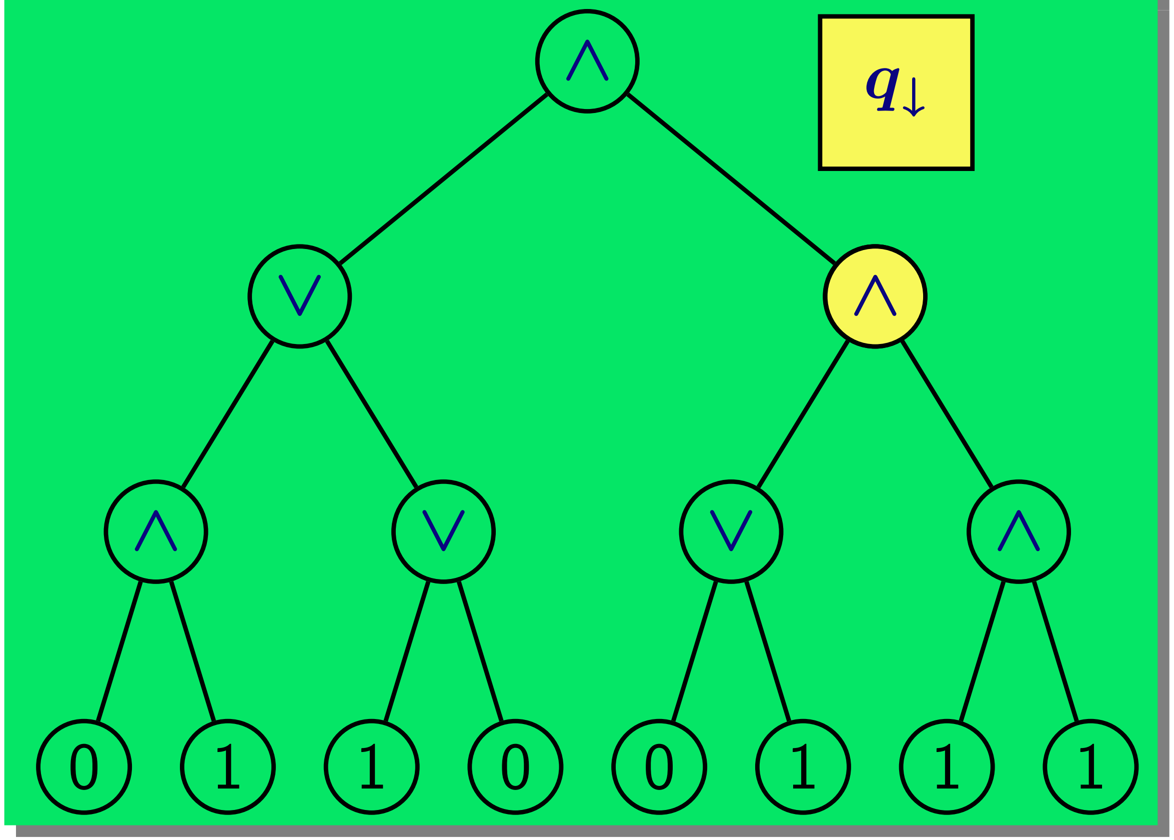
Idea



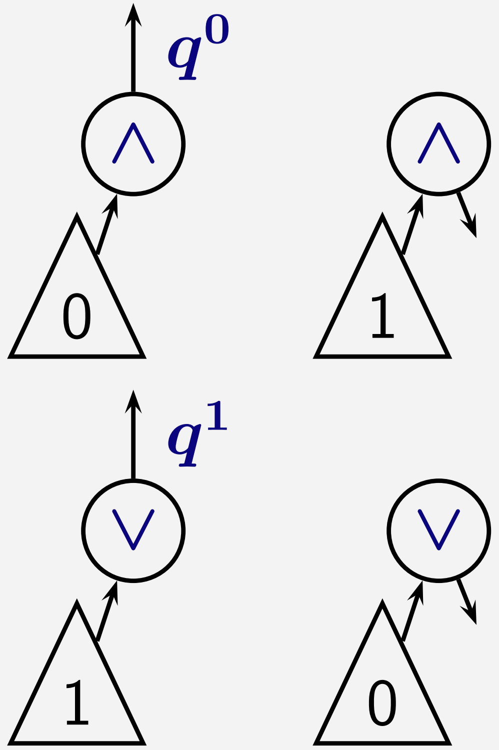
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea

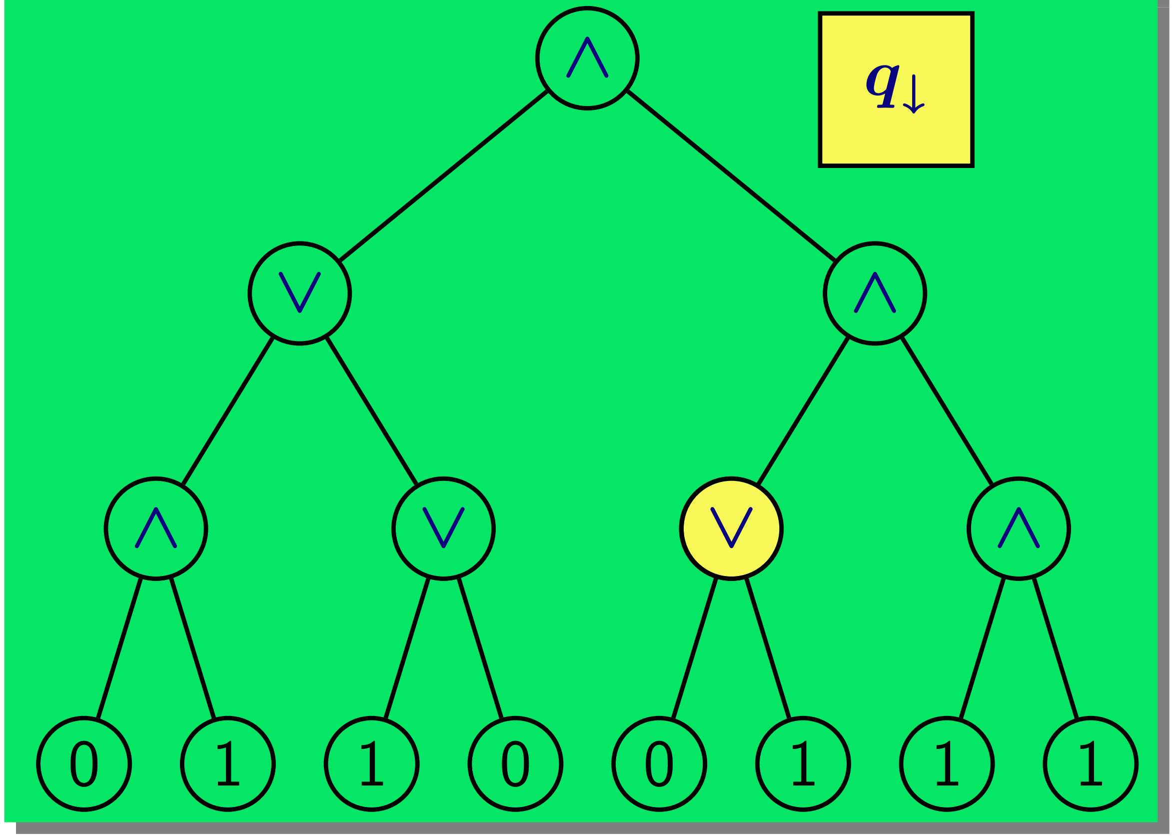


Tree-Walk Automata (cont.)

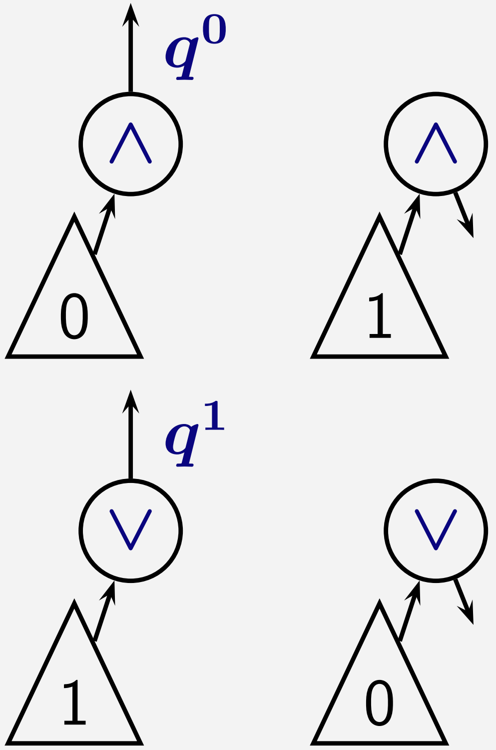
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



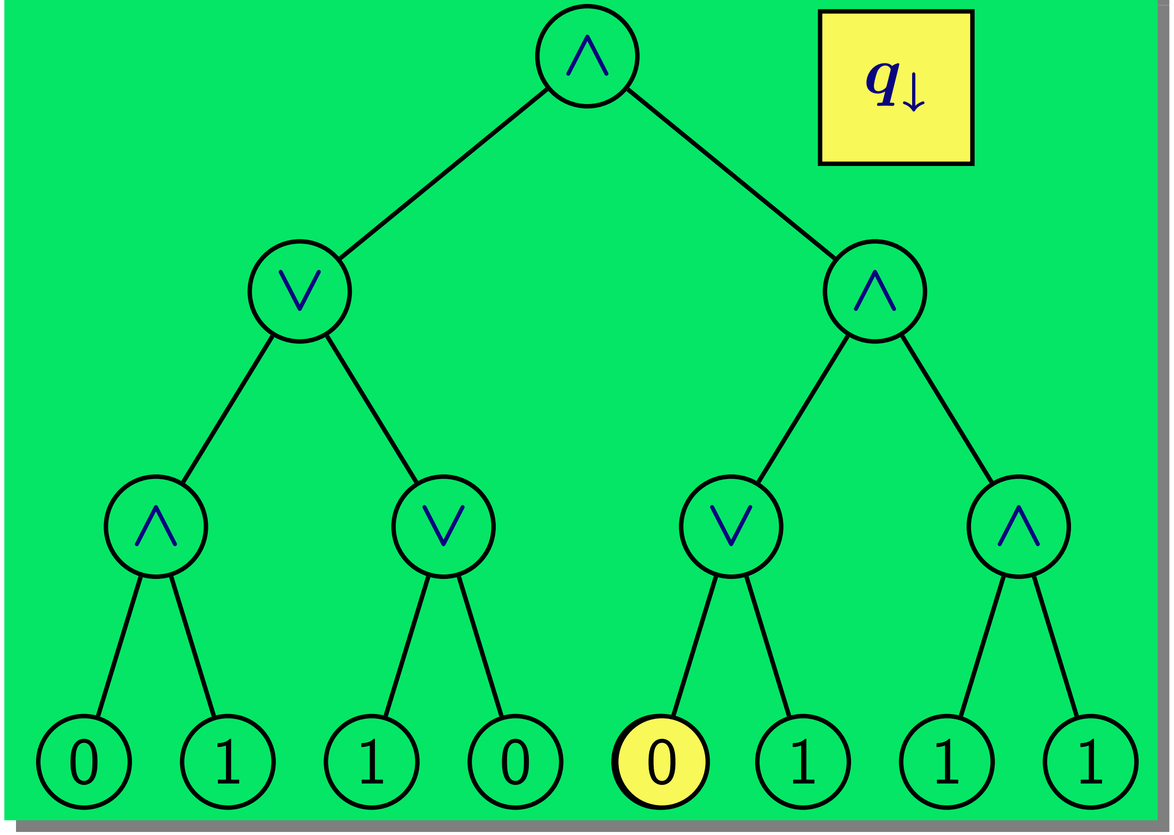
Idea



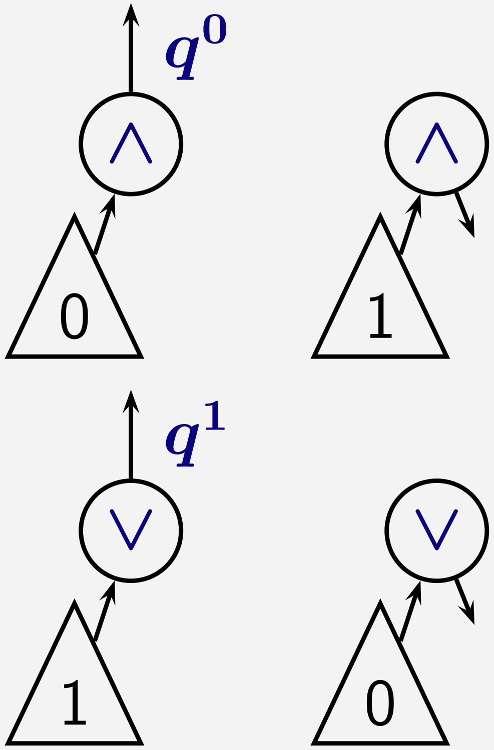
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea

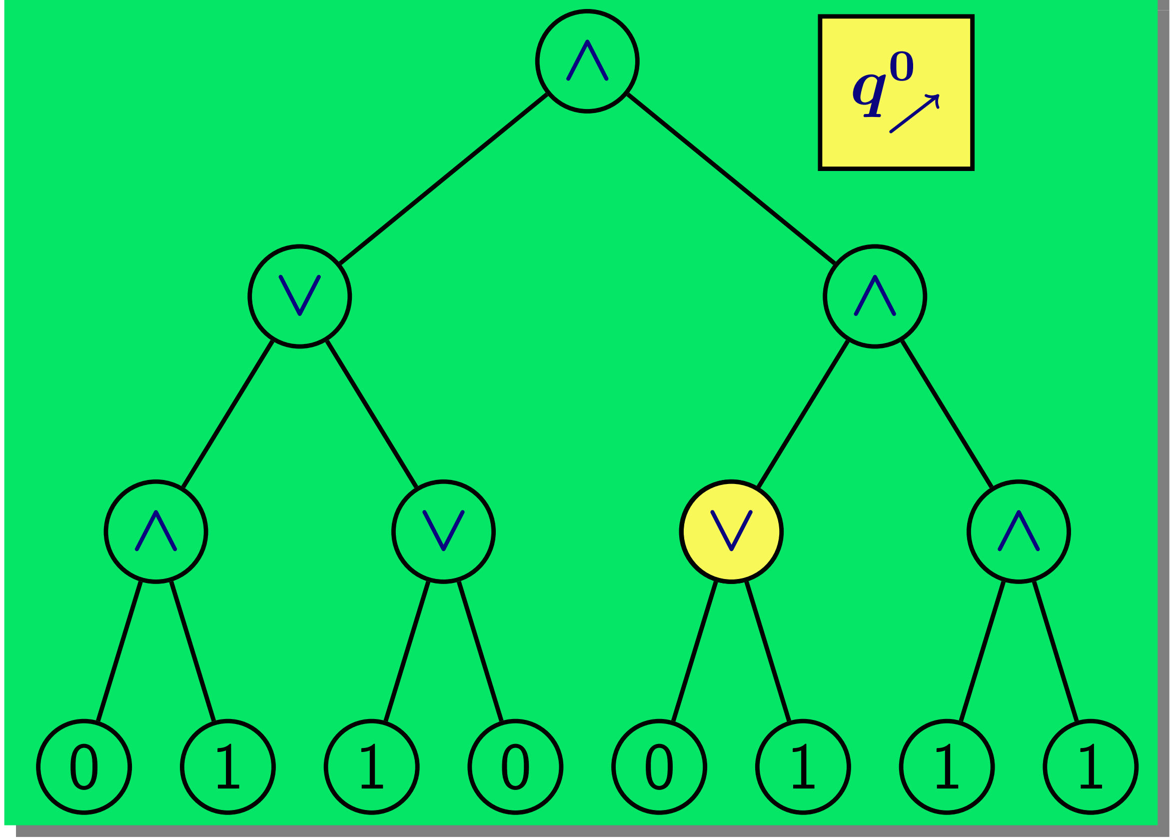


Tree-Walk Automata (cont.)

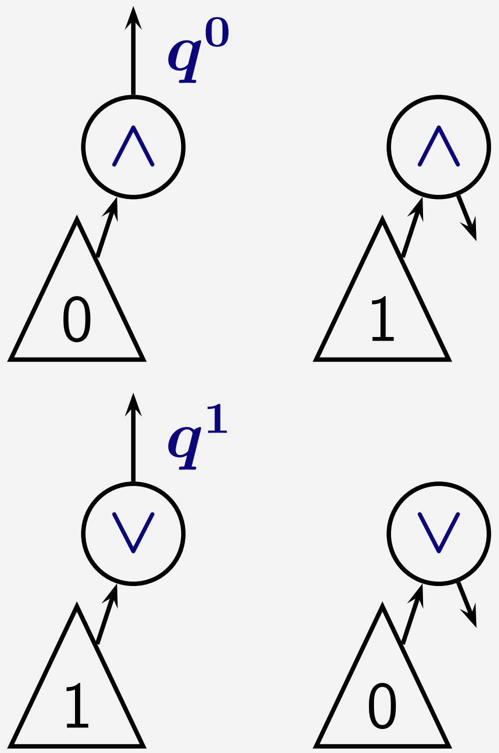
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



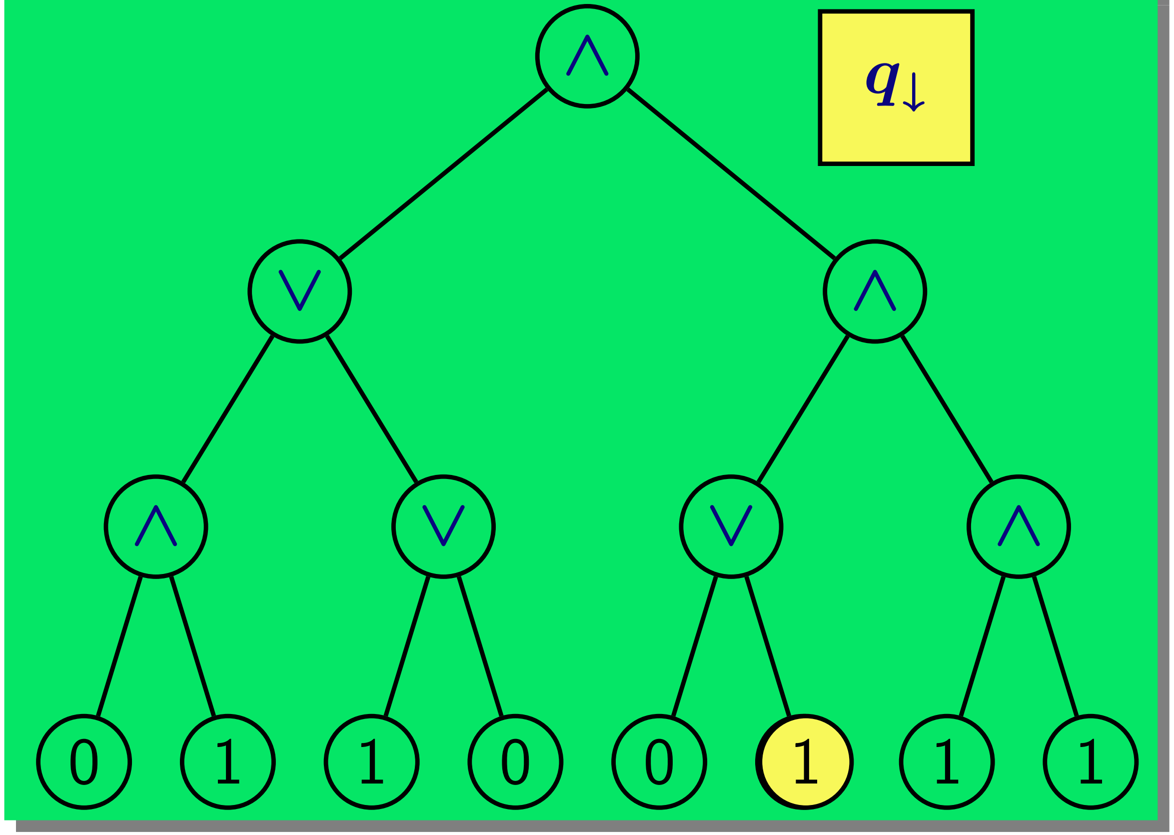
Idea



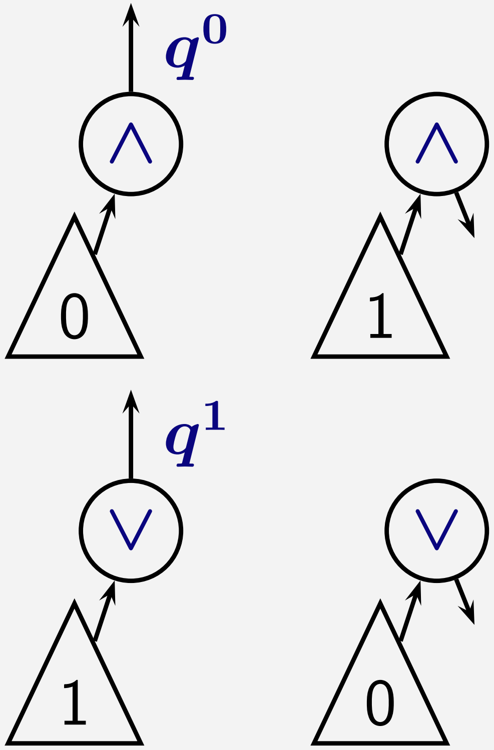
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



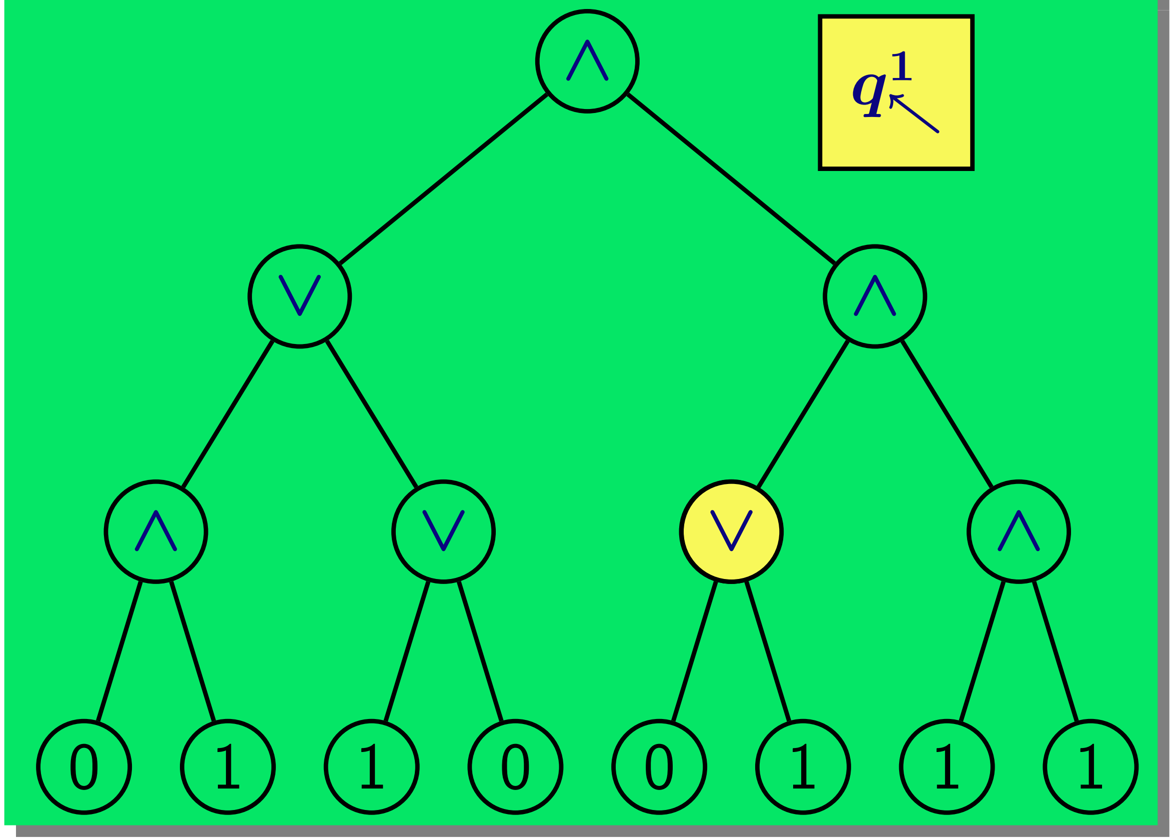
Idea



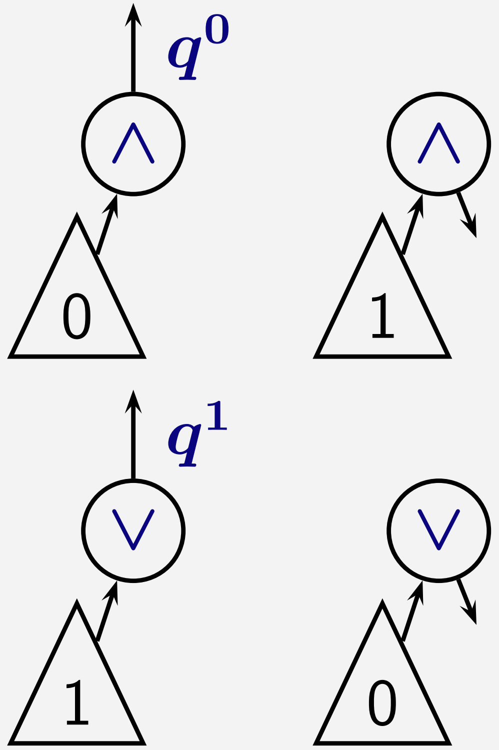
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



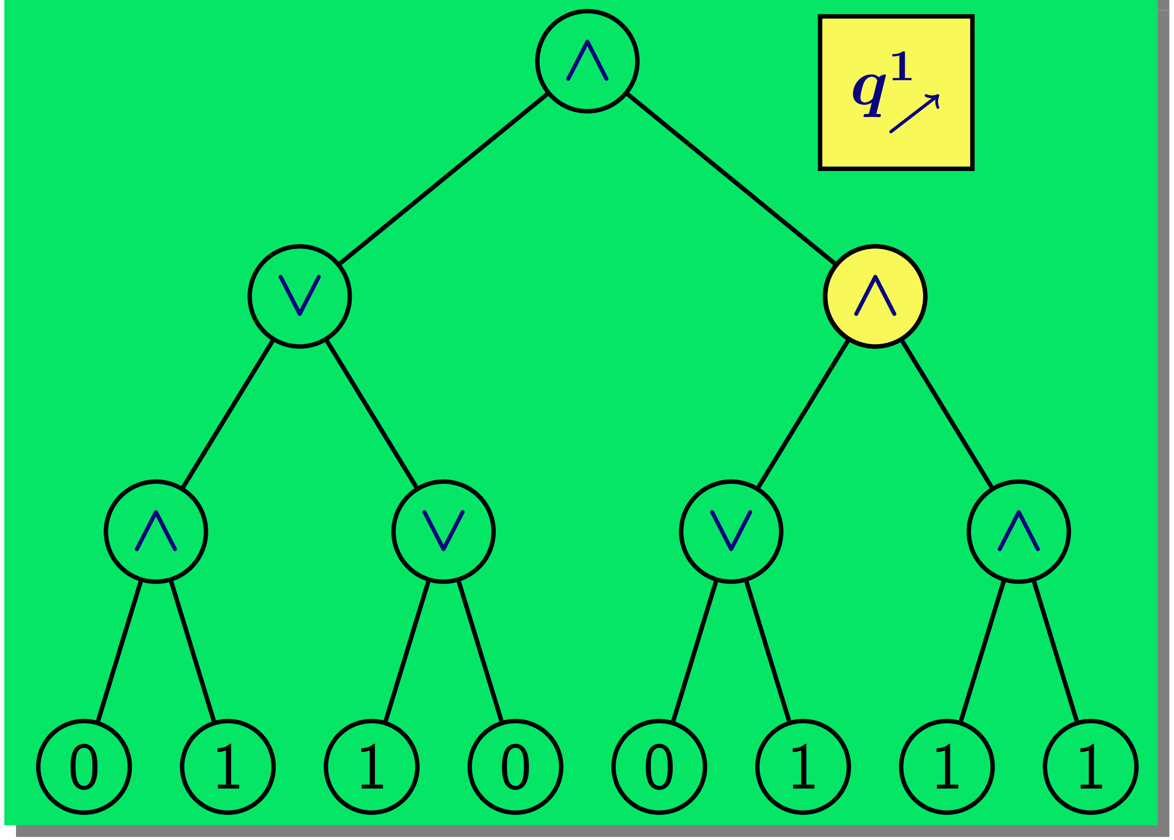
Idea



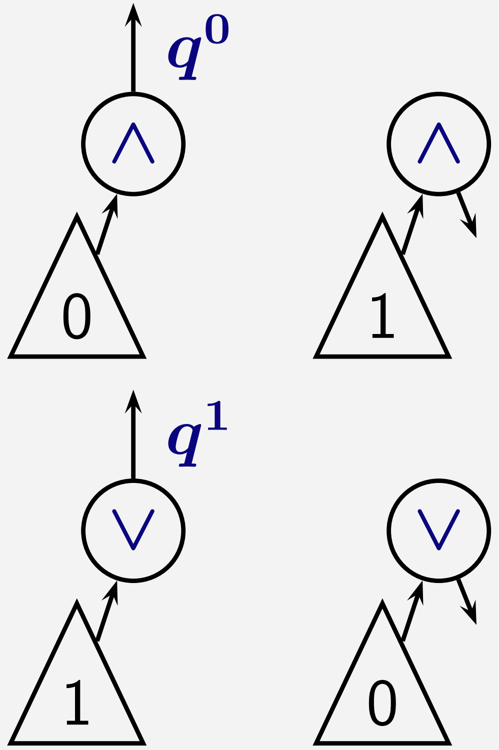
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea

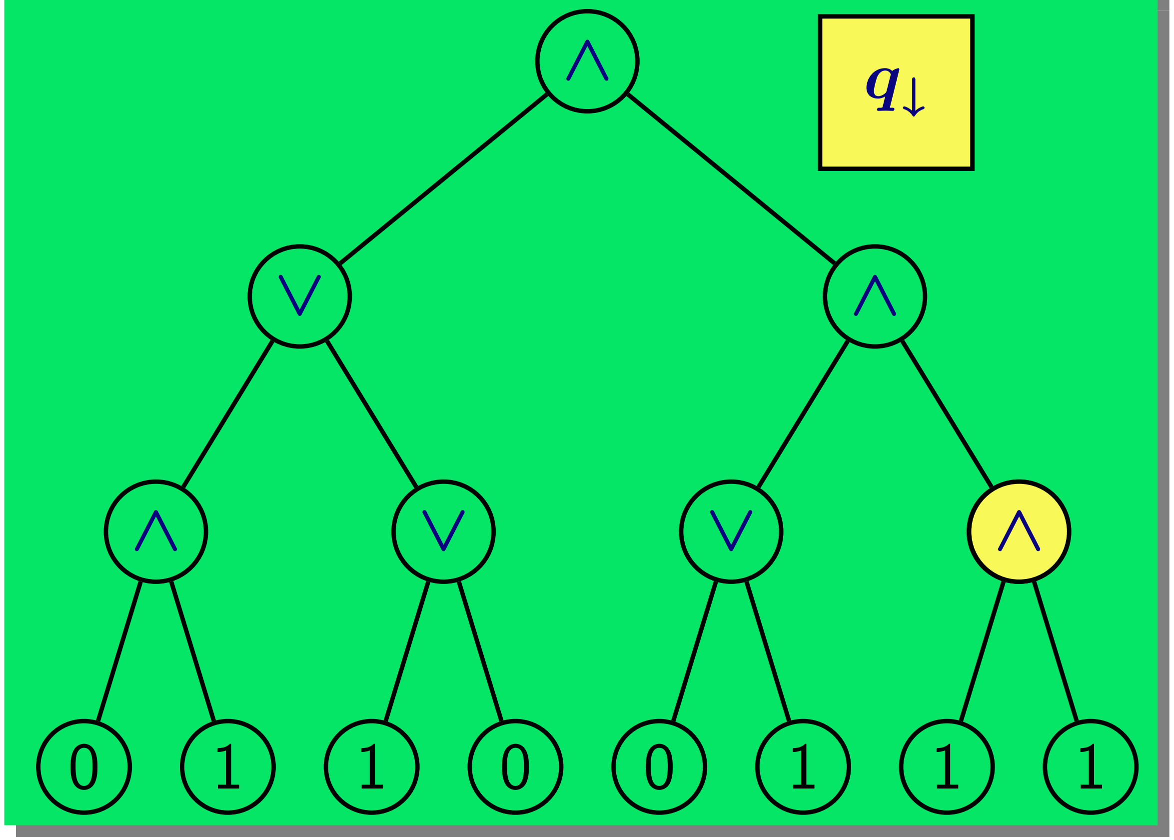


Tree-Walk Automata (cont.)

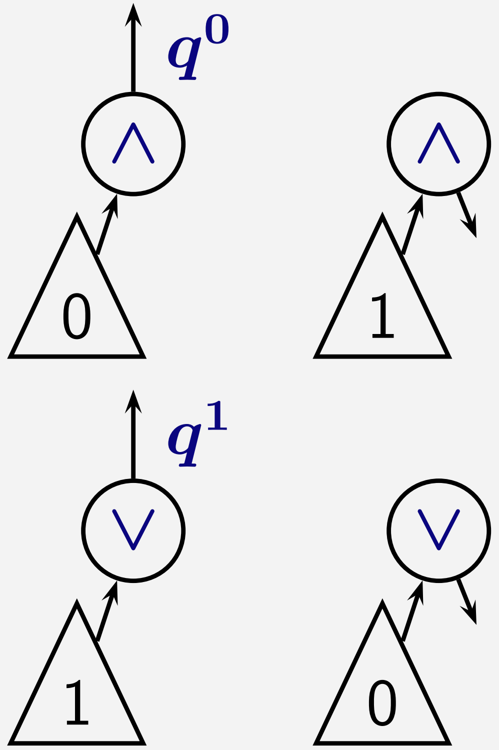
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea

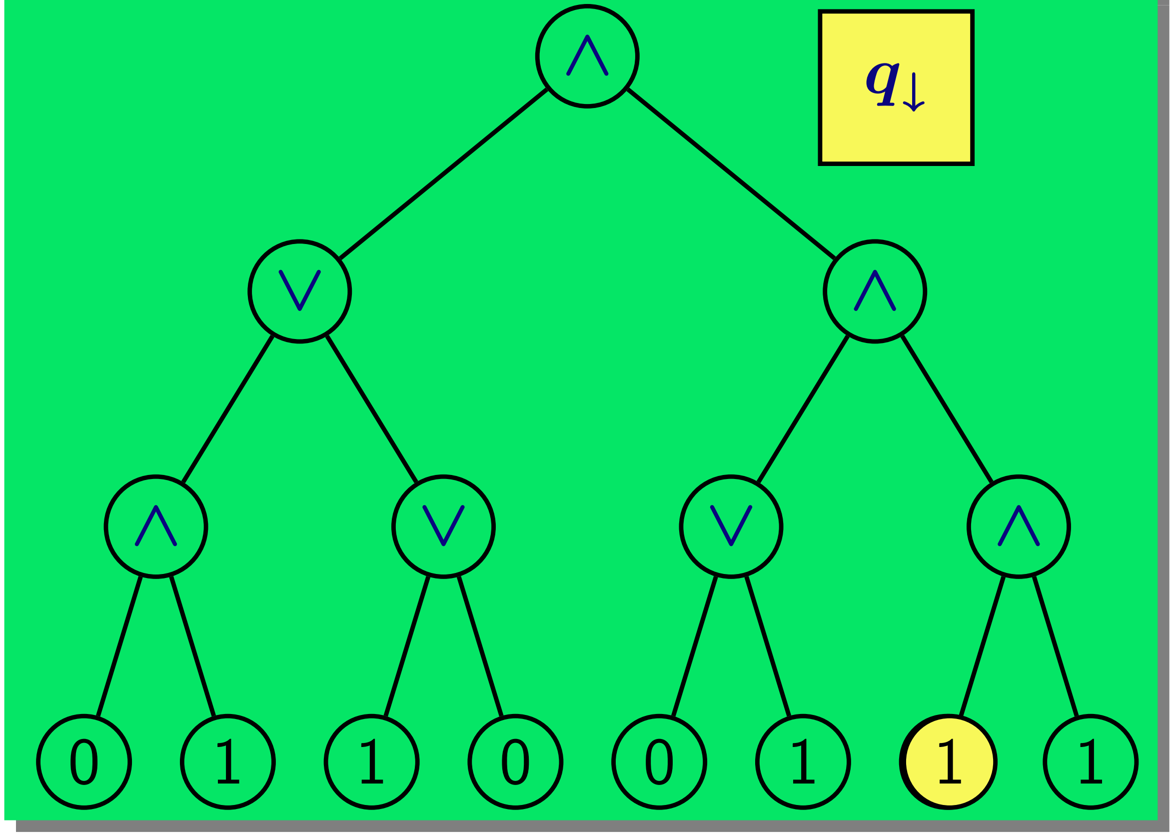


Tree-Walk Automata (cont.)

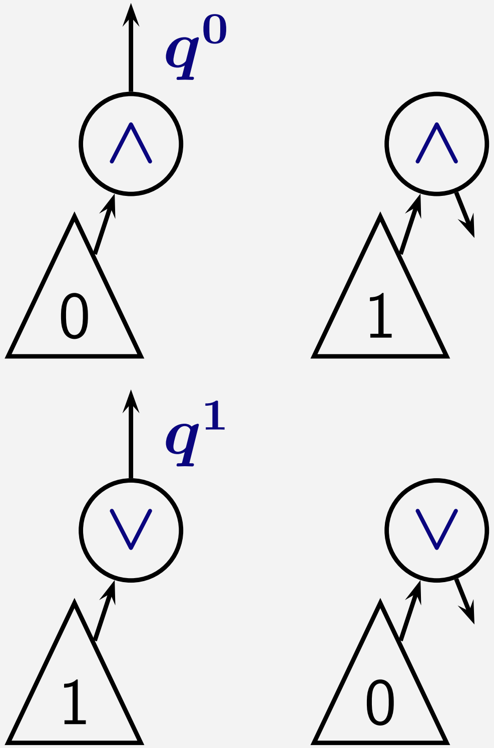
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



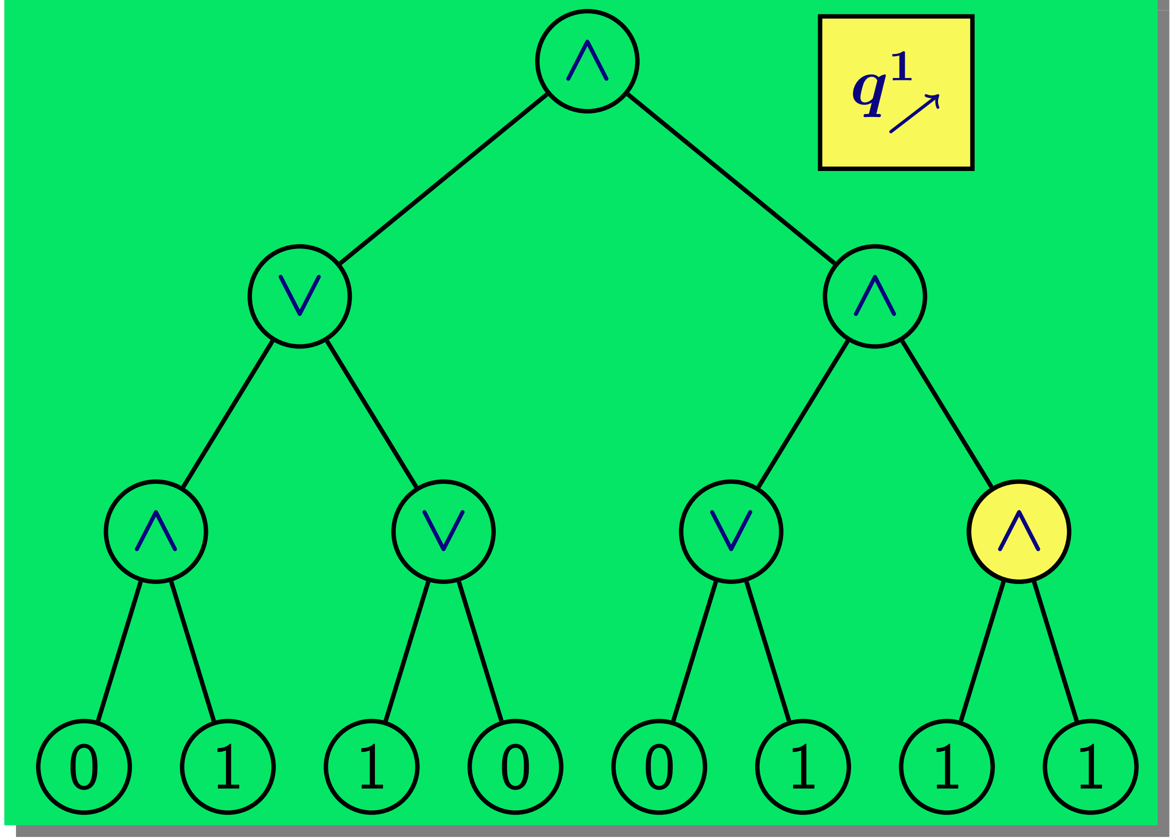
Idea



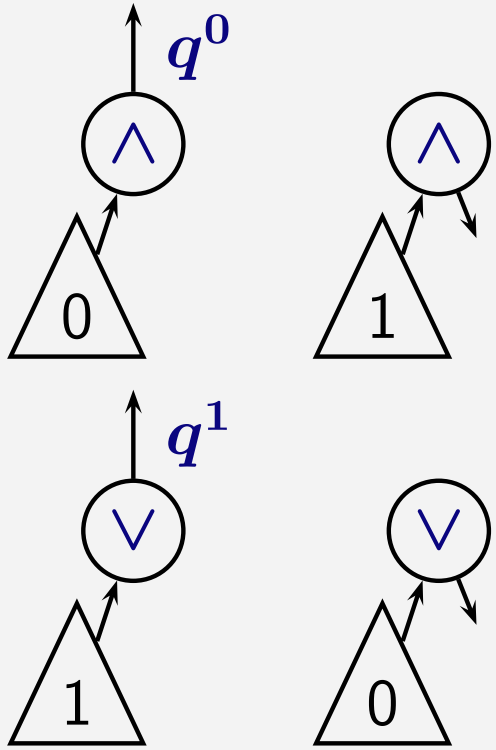
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



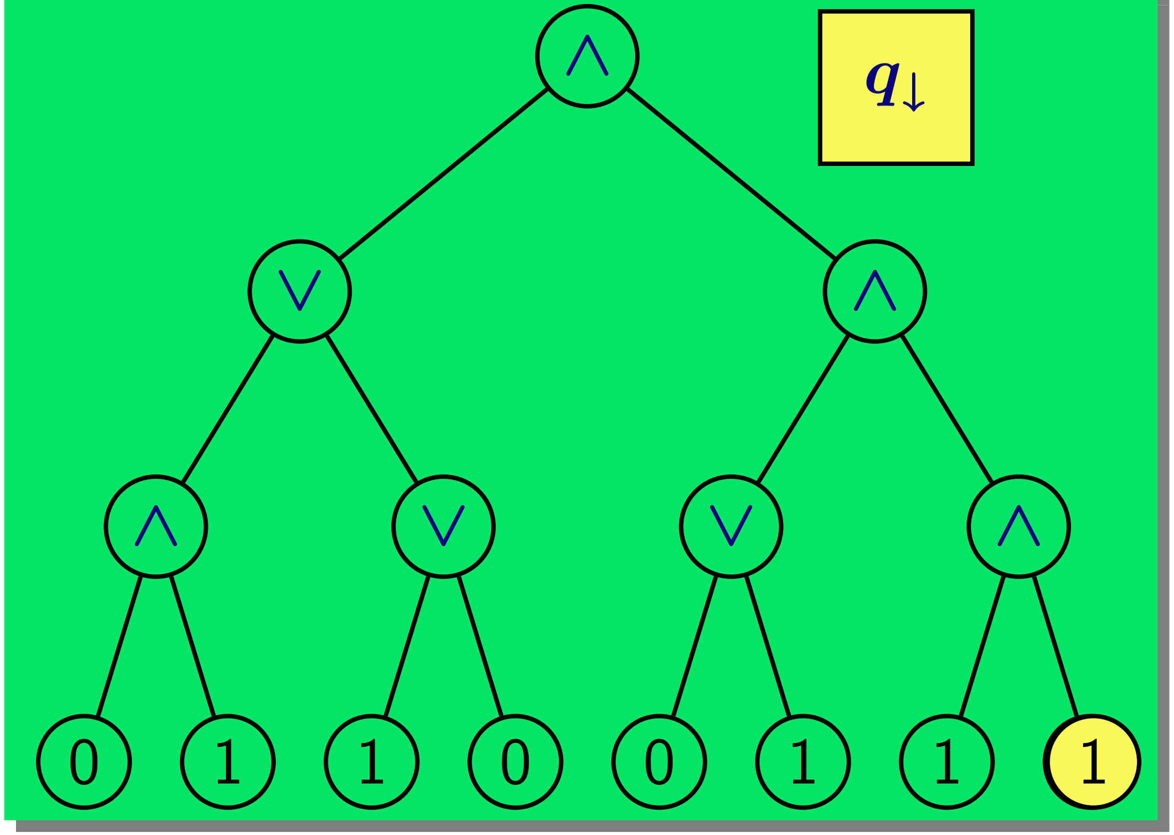
Idea



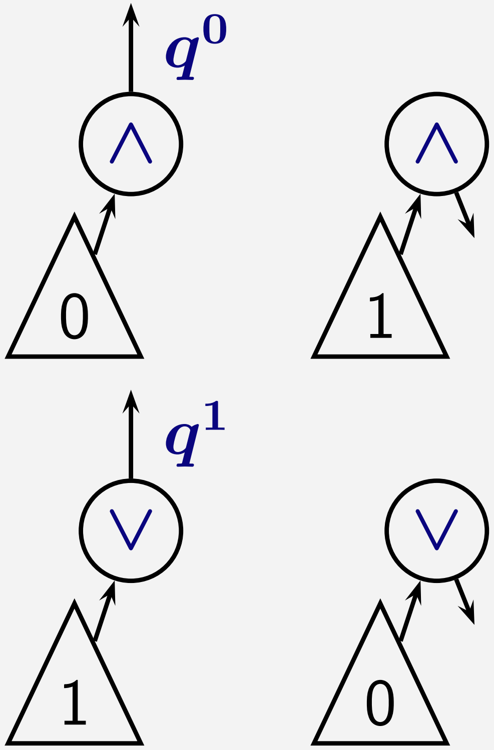
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea

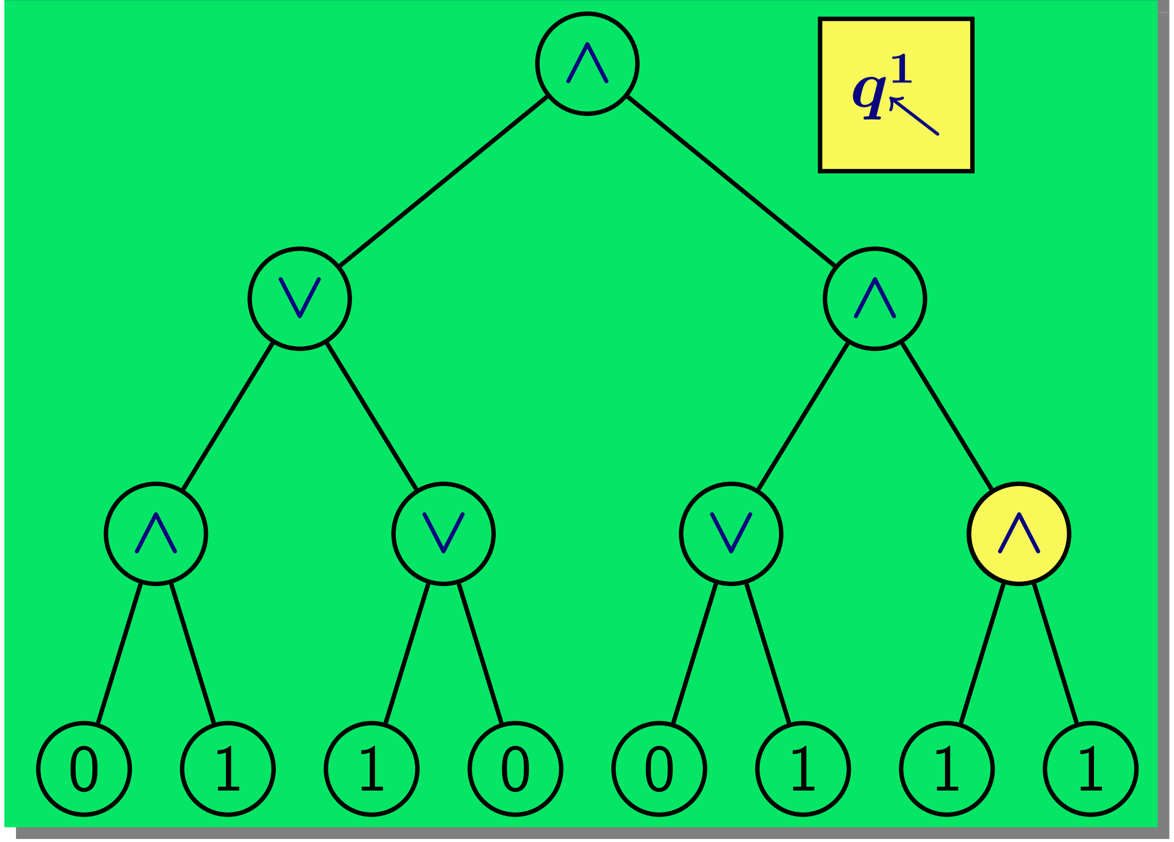


Tree-Walk Automata (cont.)

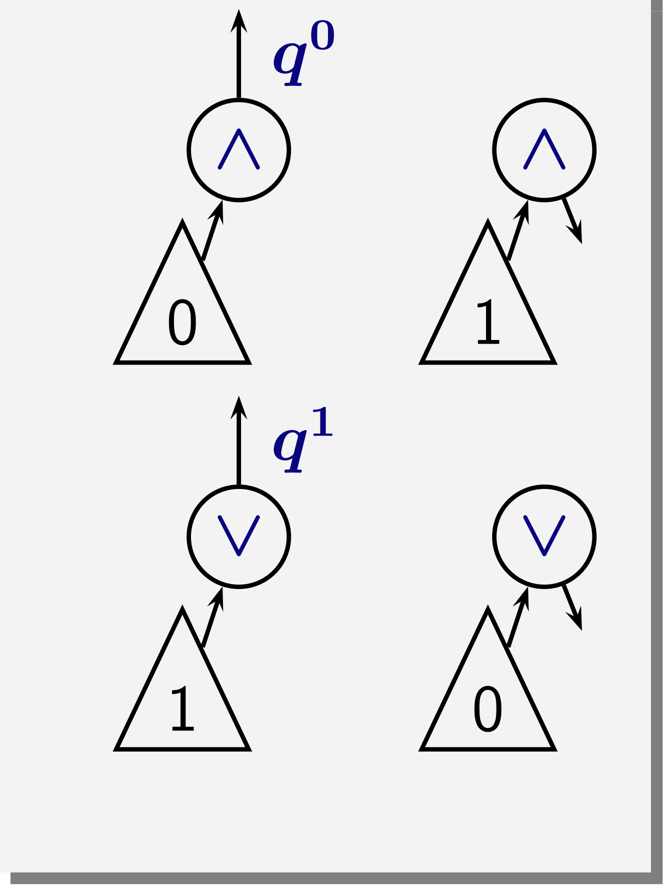
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



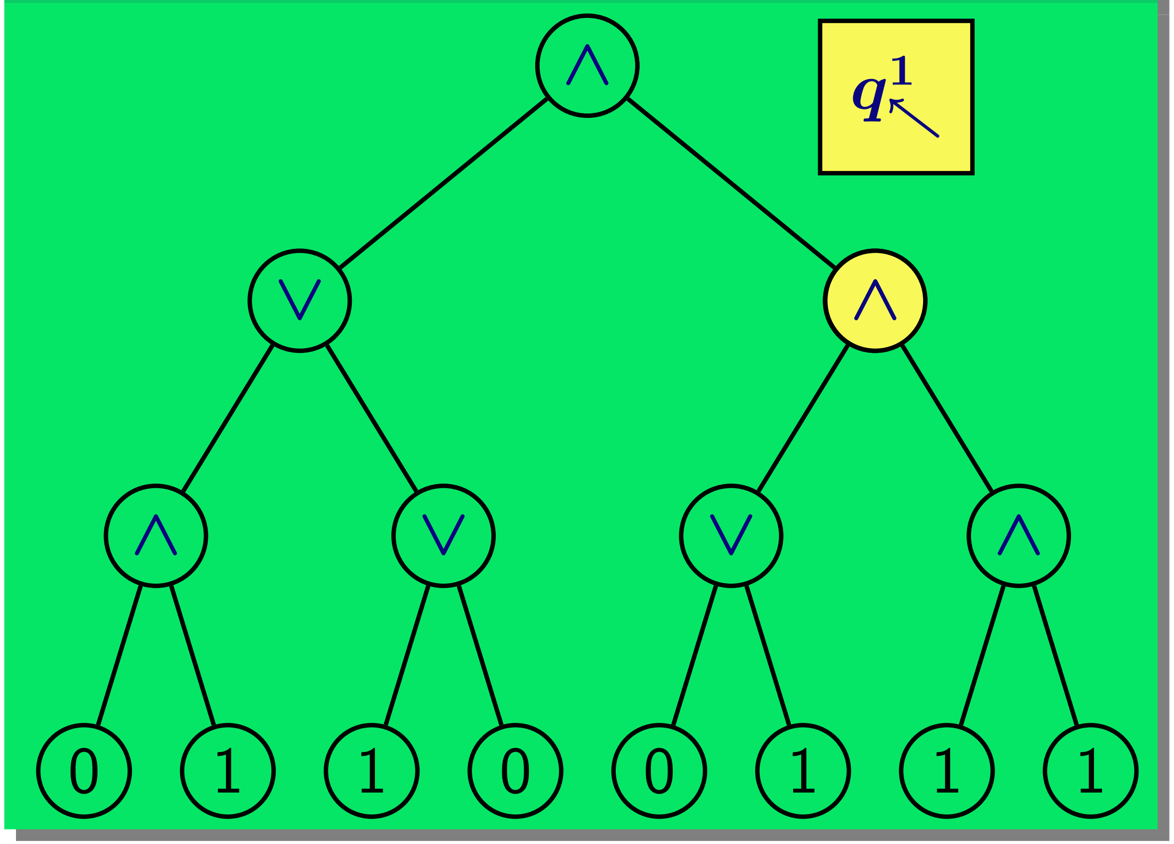
Idea



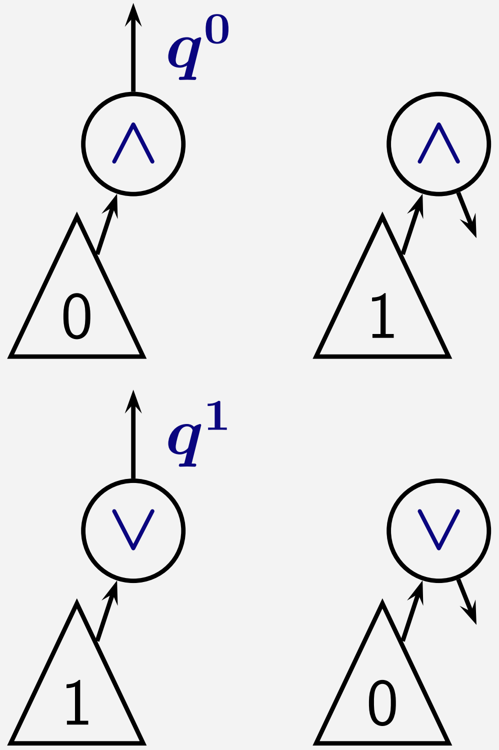
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



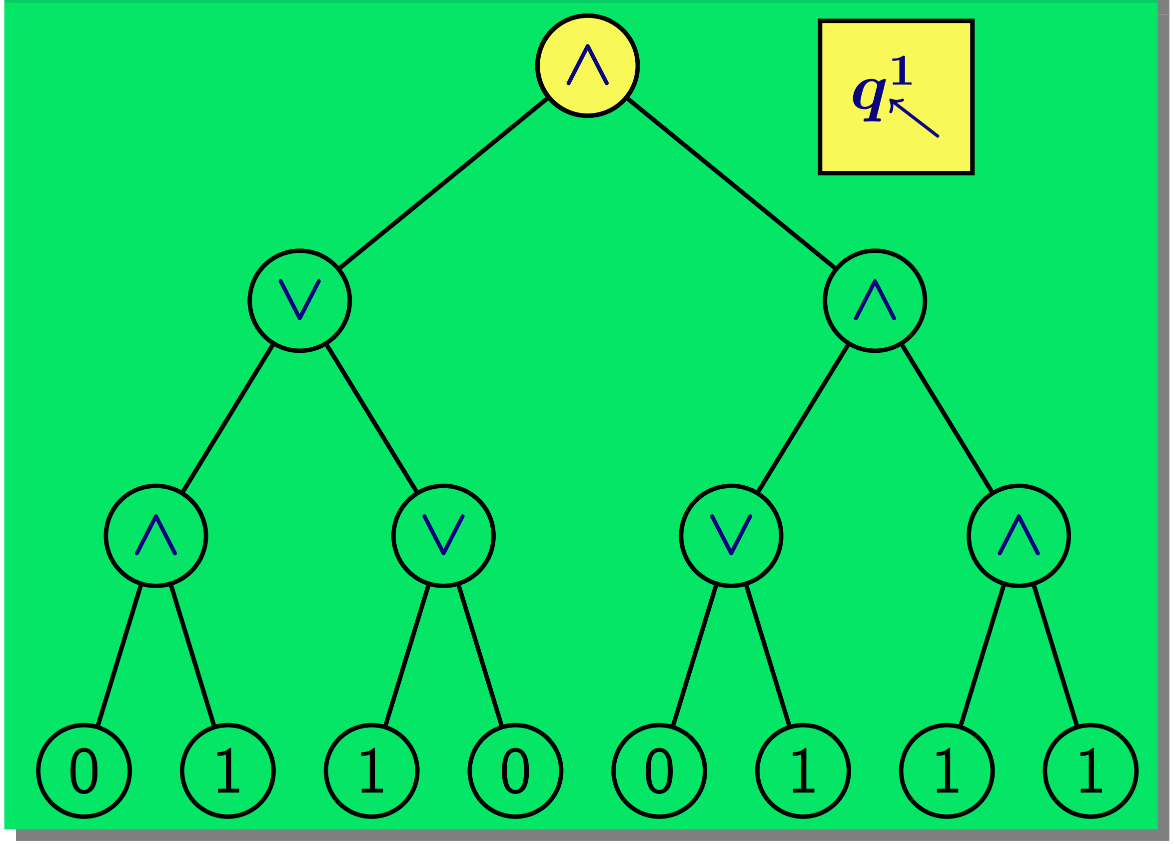
Idea



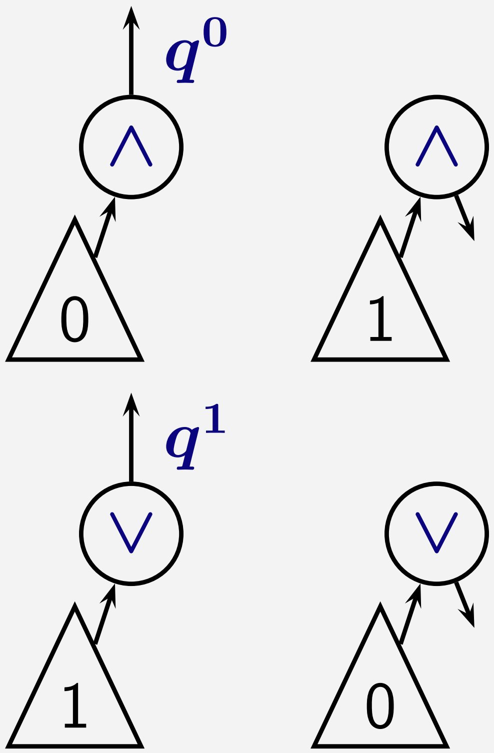
Fact

- Tree-walk automata can evaluate Boolean circuit trees
- 5 states

Example



Idea



Theorem (Bojanczyk, Colcombet 2004)

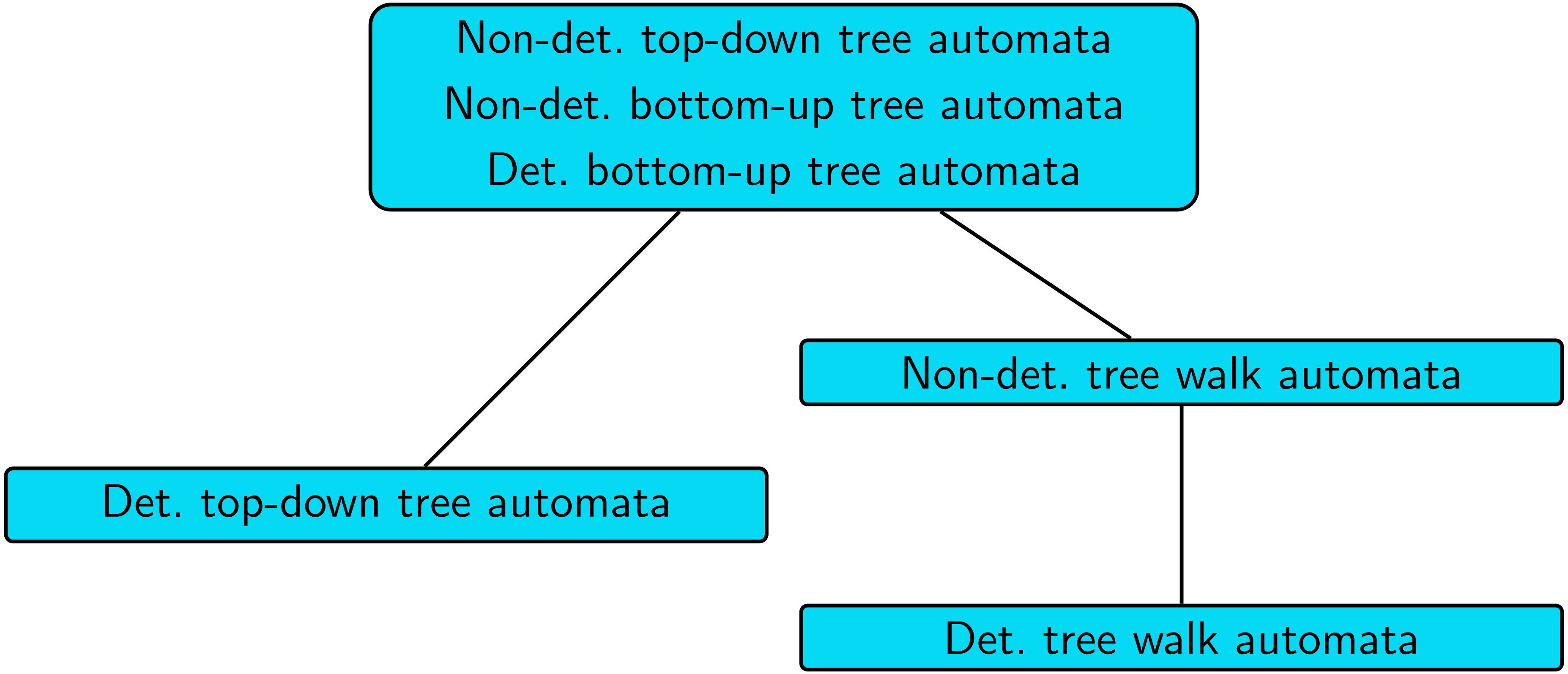
Deterministic TWAs are weaker than nondeterministic TWAs

Corollary

Deterministic TWAs do not capture all regular tree languages

Conjecture

Nondeterministic TWAs do not capture all regular tree languages

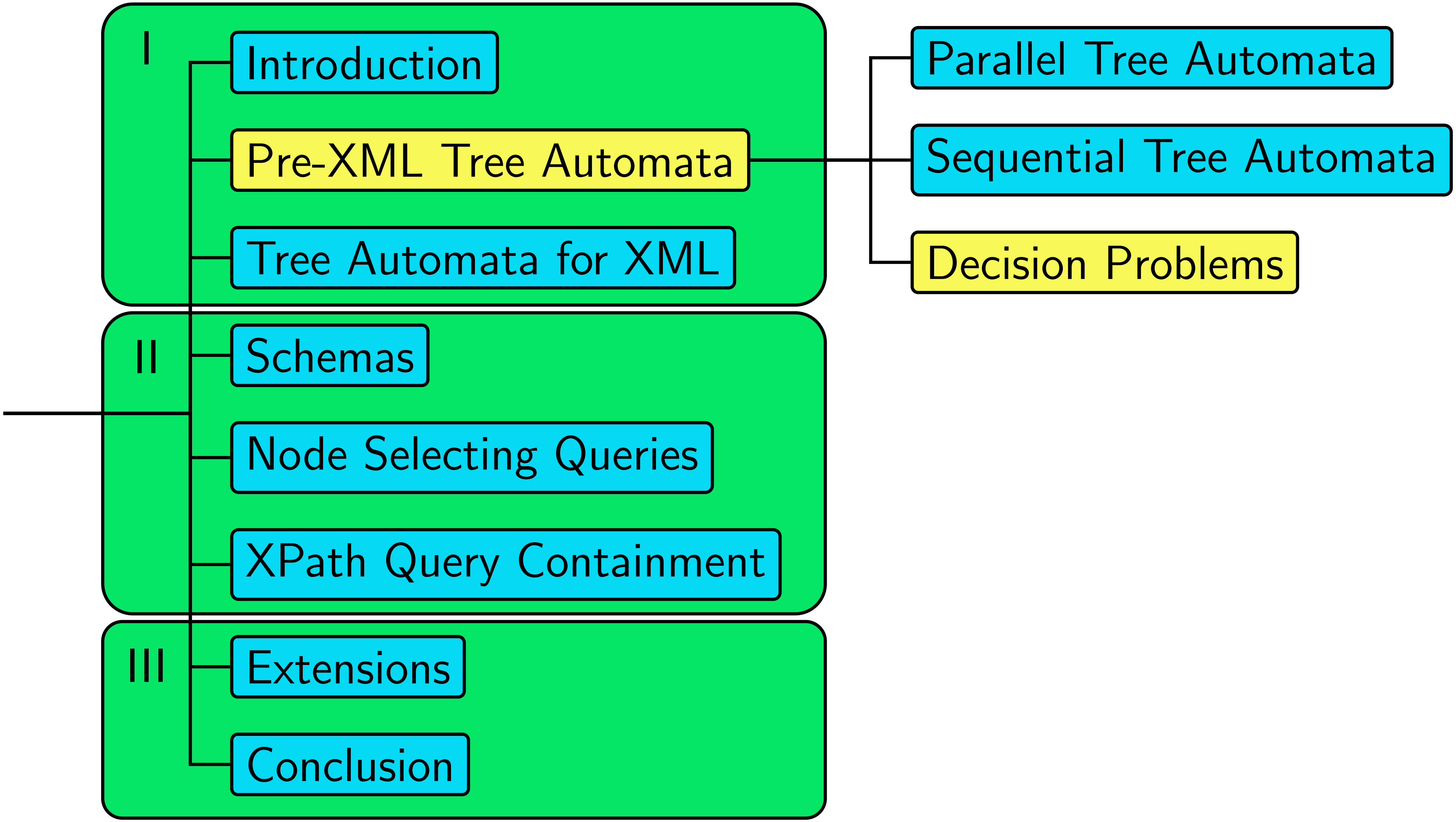


Non-det. top-down tree automata
Non-det. bottom-up tree automata
Det. bottom-up tree automata

Det. top-down tree automata

Non-det. tree walk automata

Det. tree walk automata



I

Introduction

Pre-XML Tree Automata

- Parallel Tree Automata
- Sequential Tree Automata
- Decision Problems

Tree Automata for XML

II

Schemas

Node Selecting Queries

XPath Query Containment

III

Extensions

Conclusion

Algorithmic problems

- We consider the following algorithmic problems
- All of them will be useful in the XML context

Membership test for $\mathcal{A}$

Given: Tree t

Question: Is $t \in L(\mathcal{A})$?

Membership test (combined)

Given: Tree Automaton $\mathcal{A}$, tree t

Question: Is $t \in L(\mathcal{A})$?

Non-emptiness

Given: Automaton $\mathcal{A}$

Question: Is $L(\mathcal{A}) \neq \emptyset$?

Containment

Given: Automata $\mathcal{A}_1, \mathcal{A}_2$

Question: Is $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$?

Equivalence

Given: Automata $\mathcal{A}_1, \mathcal{A}_2$

Question: Is $L(\mathcal{A}_1) = L(\mathcal{A}_2)$?

Facts

Time Bounds for the combined complexity of membership test for tree automata:

- Deterministic (parallel) tree automata: $O(|\mathcal{A}||t|)$
- Nondeterministic (parallel) tree automata: $O(|\mathcal{A}|^2|t|)$
  (Compute, for each node, the set of reachable states)
- Deterministic TWAs: $O(|\mathcal{A}|^2|t|)$
  (Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: **Behavior function**)
- Nondeterministic TWAs: $O(|\mathcal{A}|^3|t|)$
  (Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: **Behavior relation**)

Facts

Time Bounds for the combined complexity of membership test for tree automata:

→ Deterministic (parallel) tree automata: $\boldsymbol{O(|\mathcal{A}||t|)}$

- Nondeterministic (parallel) tree automata: $\boldsymbol{O(|\mathcal{A}|^2|t|)}$
  (Compute, for each node, the set of reachable states)
- Deterministic TWAs: $\boldsymbol{O(|\mathcal{A}|^2|t|)}$
  (Compute, for each node $\boldsymbol{v}$, the aggregated behavior of $\boldsymbol{\mathcal{A}}$ on its subtree: Behavior function)
- Nondeterministic TWAs: $\boldsymbol{O(|\mathcal{A}|^3|t|)}$
  (Compute, for each node $\boldsymbol{v}$, the aggregated behavior of $\boldsymbol{\mathcal{A}}$ on its subtree: Behavior relation)

Facts

Time Bounds for the combined complexity of membership test for tree automata:

- Deterministic (parallel) tree automata: $O(|\mathcal{A}||t|)$

→ Nondeterministic (parallel) tree automata: $O(|\mathcal{A}|^2|t|)$
(Compute, for each node, the set of reachable states)

- Deterministic TWAs: $O(|\mathcal{A}|^2|t|)$
(Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: Behavior function)

- Nondeterministic TWAs: $O(|\mathcal{A}|^3|t|)$
(Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: Behavior relation)

Facts

Time Bounds for the combined complexity

automata:

- Deterministic (parallel) tree automata:
- Nondeterministic (parallel) tree autom

  (Compute, for each node, the set of re

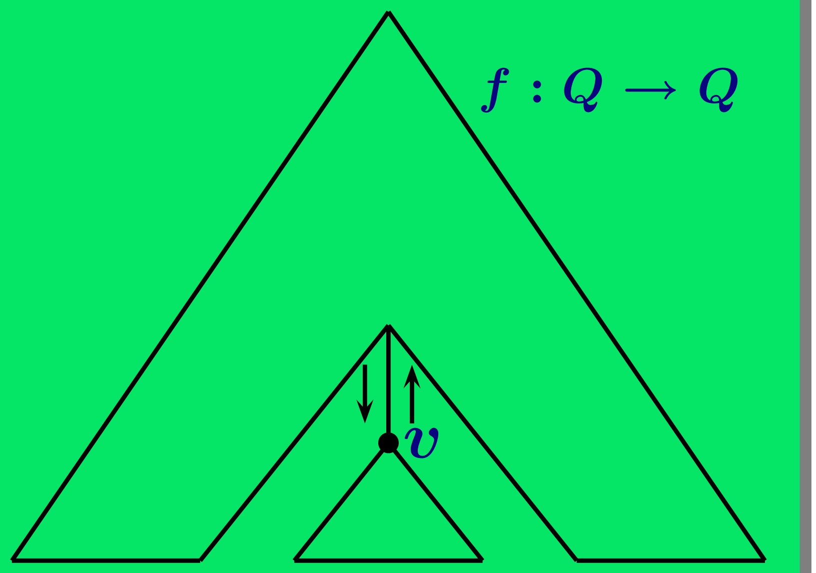
→ Deterministic TWAs: $O(|\mathcal{A}|^2|t|)$

  (Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: Behavior function)

- Nondeterministic TWAs: $O(|\mathcal{A}|^3|t|)$

  (Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: Behavior relation)

Behavior Function



Facts

Time Bounds for the combined complexity of membership test for tree automata:

- Deterministic (parallel) tree automata: $O(|\mathcal{A}||t|)$
- Nondeterministic (parallel) tree automata: $O(|\mathcal{A}|^2|t|)$
  (Compute, for each node, the set of reachable states)
- Deterministic TWAs: $O(|\mathcal{A}|^2|t|)$
  (Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: **Behavior function**)

→ Nondeterministic TWAs: $O(|\mathcal{A}|^3|t|)$
(Compute, for each node v, the aggregated behavior of $\mathcal{A}$ on its subtree: **Behavior relation**)

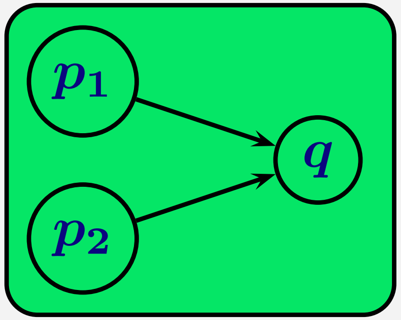
Question: What is the structural complexity for the various models?

(Lohrey 2001, Segoufin 2003)

Model	Time Complexity	Structural Complexity
Det. top-down TA	$O(\lvert\mathcal{A}\rvert\lvert t\rvert)$	**LOGSPACE**
Det. bottom-up TA	$O(\lvert\mathcal{A}\rvert\lvert t\rvert)$	**LOGDCFL**
Nondet. bottom-up TA	$O(\lvert\mathcal{A}\rvert^2\lvert t\rvert)$	**LOGCFL**
Nondet. top-down TA	$O(\lvert\mathcal{A}\rvert^2\lvert t\rvert)$	**LOGCFL**
Det. TWA	$O(\lvert\mathcal{A}\rvert^2\lvert t\rvert)$	**LOGSPACE**
Nondet. TWA	$O(\lvert\mathcal{A}\rvert^3\lvert t\rvert)$	**NLOGSPACE**

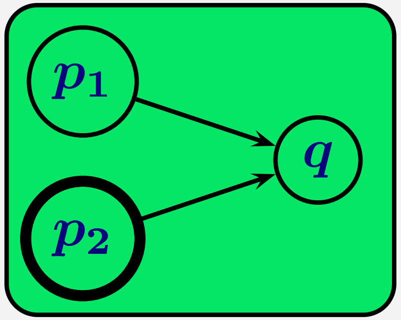
Facts

- Non-emptiness for string automata corresponds to `Graph Reachability` (complete for **NLOGSPACE**)
- Non-emptiness for tree automata corresponds to `Path Systems` :



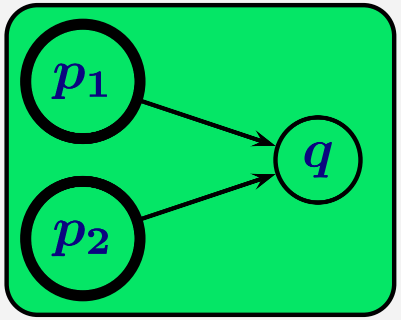
Facts

- Non-emptiness for string automata corresponds to `Graph Reachability` (complete for **NLOGSPACE**)
- Non-emptiness for tree automata corresponds to `Path Systems`:



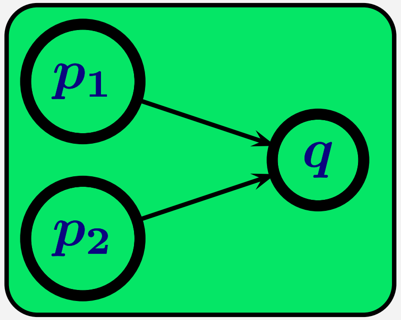
Facts

- Non-emptiness for string automata corresponds to `Graph Reachability` (complete for **NLOGSPACE**)
- Non-emptiness for tree automata corresponds to `Path Systems` :



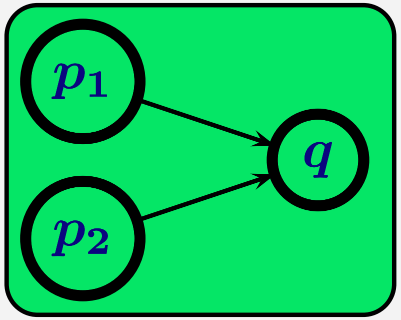
Facts

- Non-emptiness for string automata corresponds to `Graph Reachability` (complete for **NLOGSPACE**)
- Non-emptiness for tree automata corresponds to `Path Systems` :



Facts

- Non-emptiness for string automata corresponds to `Graph Reachability` (complete for **NLOGSPACE**)
- Non-emptiness for tree automata corresponds to `Path Systems` :



Result

- Non-emptiness for bottom-up tree automata can be checked in linear time
- It is complete for **PTIME**

Observations

- Of course:

$$L(\mathcal{A}_1) = L(\mathcal{A}_2) \iff [L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2) \text{ and } L(\mathcal{A}_2) \subseteq L(\mathcal{A}_1)]$$

- Complexity of containment problem is very different for deterministic and non-deterministic automata
- Deterministic automata: construct product automaton

Deterministic bottom-up tree automata

- Product automaton analogous as for string automata
  - Set of states: $Q_1 \times Q_2$
  - Transitions component-wise
- To check $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$:
  - Compute $\mathcal{B} = \mathcal{A}_1 \times \mathcal{A}_2$
  - Accepting states: $F_1 \times (Q_2 - F_2)$
  - Check whether $L(\mathcal{B}) = \emptyset$
  - If so, $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$ holds

Theorem

Complexity of Containment for deterministic bottom-up tree automata:

$$O(|\mathcal{A}_1| \times |\mathcal{A}_2|)$$

Non-deterministic automata

- Naive approach:
  - Make $\mathcal{A}_2$ deterministic (size: $O(2^{|\mathcal{A}_2|})$)
  - Construct product automaton

⇒ Exponential time

Containment: Complexity (cont.)

Non-deterministic automata

- Naive approach:
  - Make $\mathcal{A}_2$ deterministic (size: $O(2^{|\mathcal{A}_2|})$)
  - Construct product automaton

$\Rightarrow$ Exponential time

Unfortunately...

There is essentially no better way

Containment: Complexity (cont.)

Non-deterministic automata

- Naive approach:
  - Make $\mathcal{A}_2$ deterministic (size: $O(2^{|\mathcal{A}_2|})$)
  - Construct product automaton
  - $\Rightarrow$ Exponential time

Unfortunately…

There is essentially no better way

Theorem (Seidl 1990)

Containment for non-deterministic tree automata is complete for **EXPTIME**

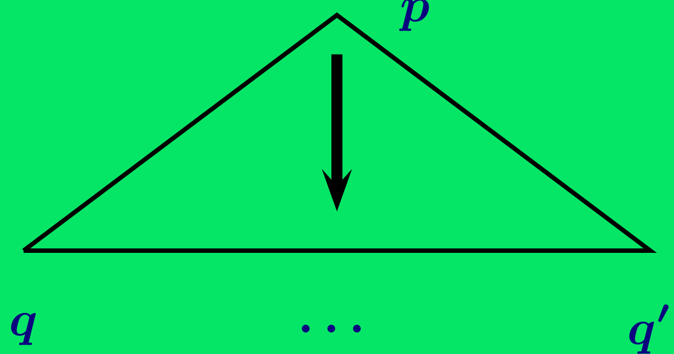
Theorem

Nonemptiness for deterministic top-down automata $\mathcal{A}$ can be checked in polynomial time

Proof Idea

Check for each state p of $\mathcal{A}$ and each pair (q, q') of the leaves automaton $\mathcal{B}$:
Is there is a tree t such that $\mathcal{A}$ starts from state p and obtains a leave string which brings $\mathcal{B}$ from q to q'?

Illustration



Theorem

Containment for deterministic top-down automata $\mathcal{A}$ can be checked in polynomial time

Proof Idea

- Tree automata $\mathcal{A}_1$, $\mathcal{A}_2$ with leaves automata $\mathcal{B}_1, \mathcal{B}_2$
- Check
  - for each pair (p_1, p_2) of states of $\mathcal{A}_1$ and $\mathcal{A}_2$ and
  - for each two pairs (q_1, q_1') and (q_2, q_2') of $\mathcal{B}_1$ and $\mathcal{B}_2$, resp.:

  Is there is a tree t such that for both $i = 1$, $i = 2$: T_i starts from state p_i and obtains a leave string which brings $\mathcal{B}_i$ from q_i to q_i'?

Complexities of basic algorithmic problems

Model	Membership	Non-emptiness	Containment
Det. top-down TA	**LOGSPACE**	**PTIME**	**PTIME**
Det. bottom-up TA	**LOGDCFL**	**PTIME**	**PTIME**
Nondet. bottom-up TA	**LOGCFL**	**PTIME**	**EXPTIME**
Nondet. top-down TA	**LOGCFL**	**PTIME**	**EXPTIME**
Det. TWA	**LOGSPACE**	**PTIME** $(*)$	**PTIME** $(*)$
Nondet. TWA	**NLOGSPACE**	**PTIME** $(*)$	**EXPTIME** $(*)$

($*$: upper bounds)

A further result to remember

Theorem (Stockmeyer, Meyer 1971) Containment and Equivalence for regular expressions on strings are complete for **PSPACE**

I

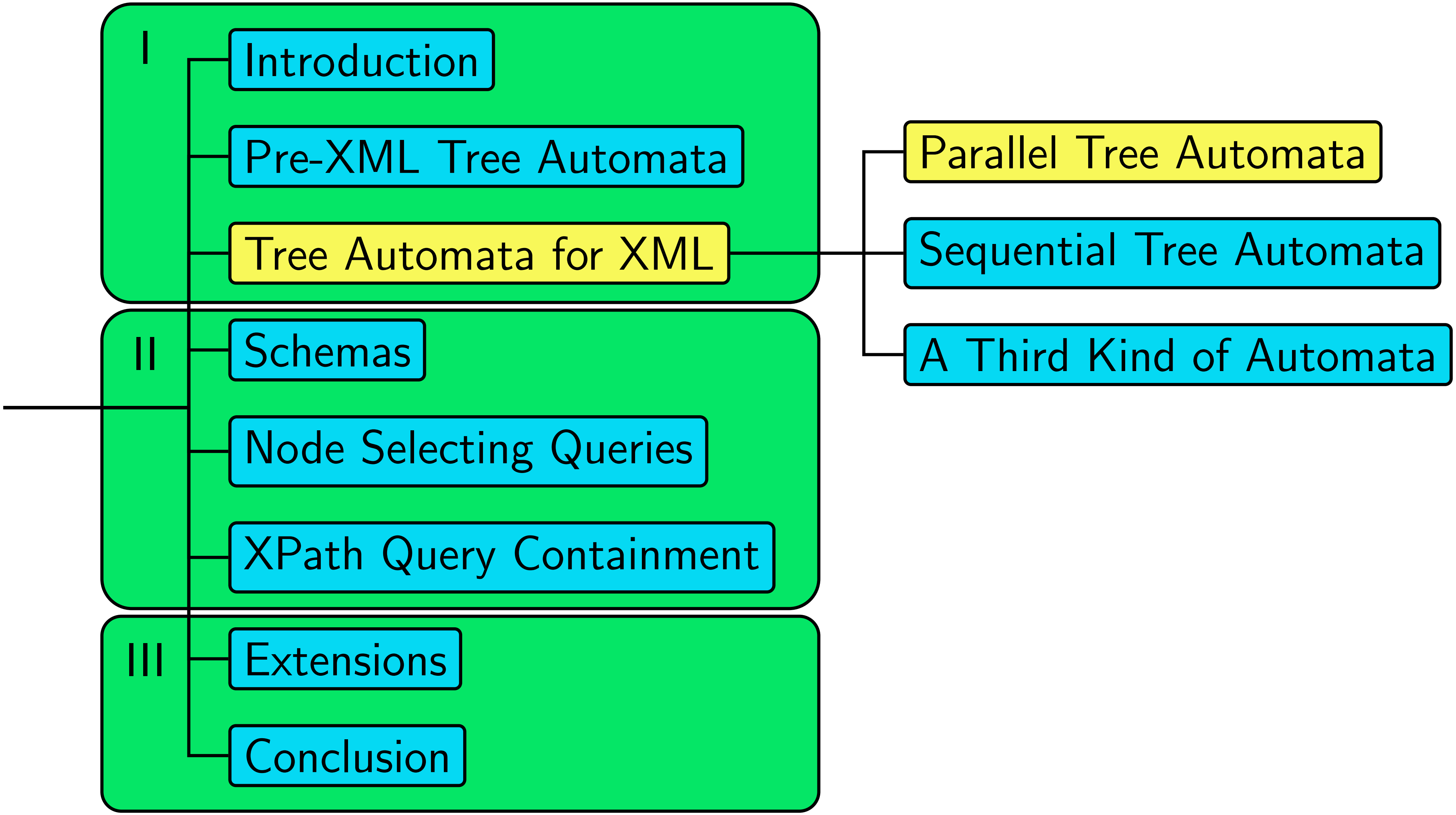
- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II

- Schemas
- Node Selecting Queries
- XPath Query Containment

III

- Extensions
- Conclusion



I

Introduction

Pre-XML Tree Automata

Tree Automata for XML

- Parallel Tree Automata
- Sequential Tree Automata
- A Third Kind of Automata

II

Schemas

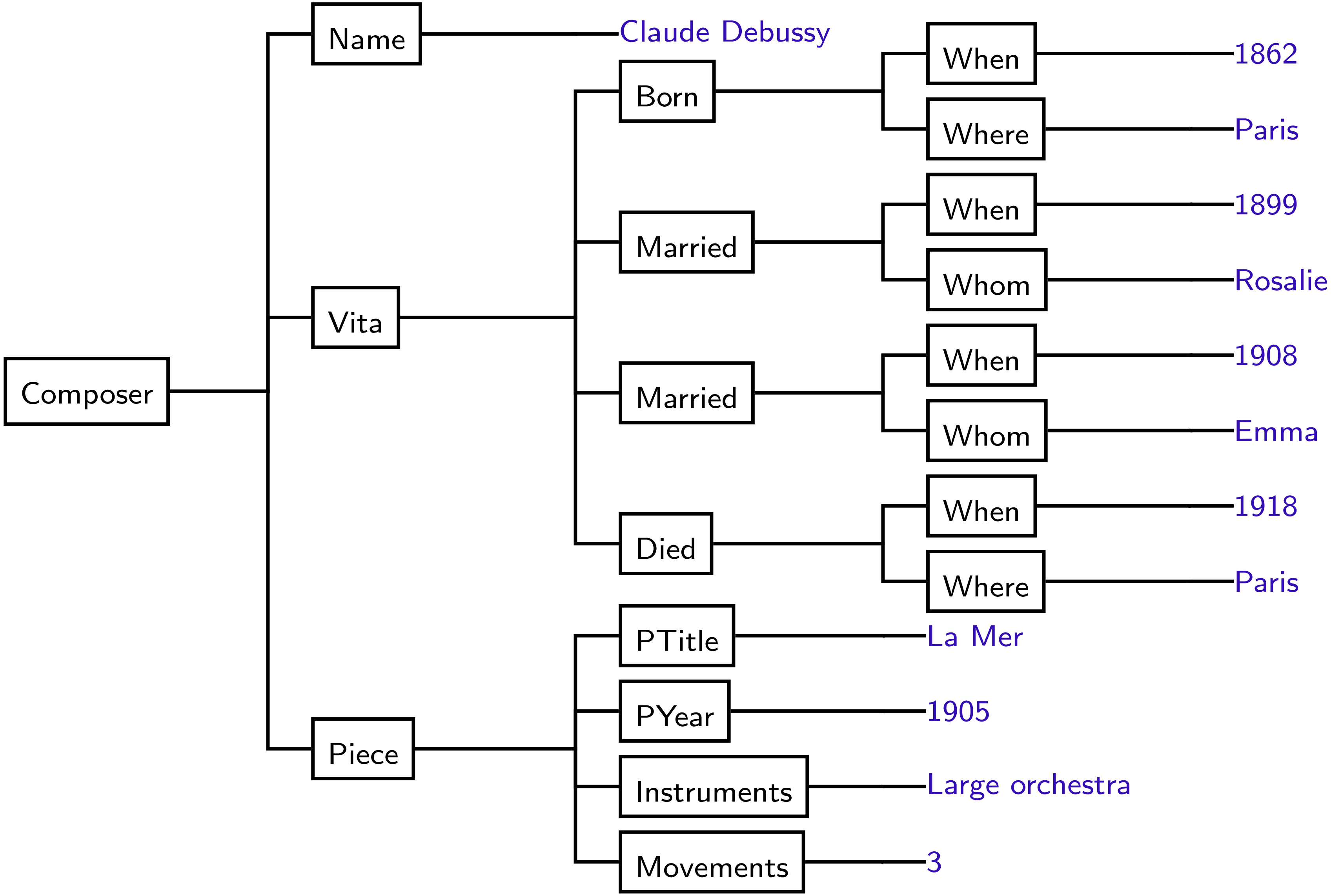
Node Selecting Queries

XPath Query Containment

III

Extensions

Conclusion

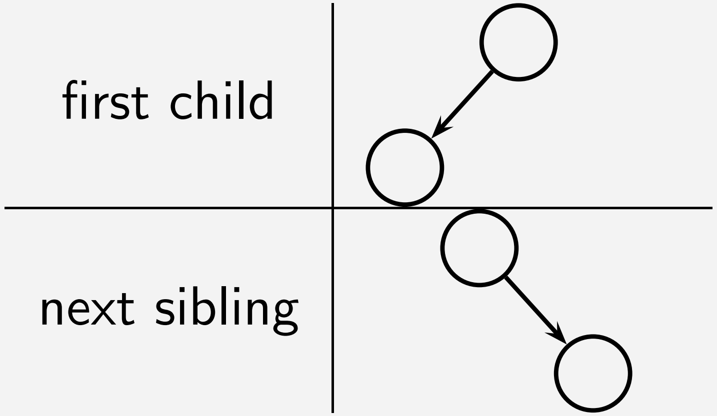


- Composer
  - Name
    - Claude Debussy
  - Vita
    - Born
      - When
        - 1862
      - Where
        - Paris
    - Married
      - When
        - 1899
      - Whom
        - Rosalie
    - Married
      - When
        - 1908
      - Whom
        - Emma
    - Died
      - When
        - 1918
      - Where
        - Paris
  - Piece
    - PTitle
      - La Mer
    - PYear
      - 1905
    - Instruments
      - Large orchestra
    - Movements
      - 3

Agenda

- Now we move from ranked to unranked trees
- There is a basic choice:
  - Either: we encode unranked trees as binary trees and go on with ranked automata
  - Or: we adapt the ranked automata models
- In both cases: not many surprises, most results remain

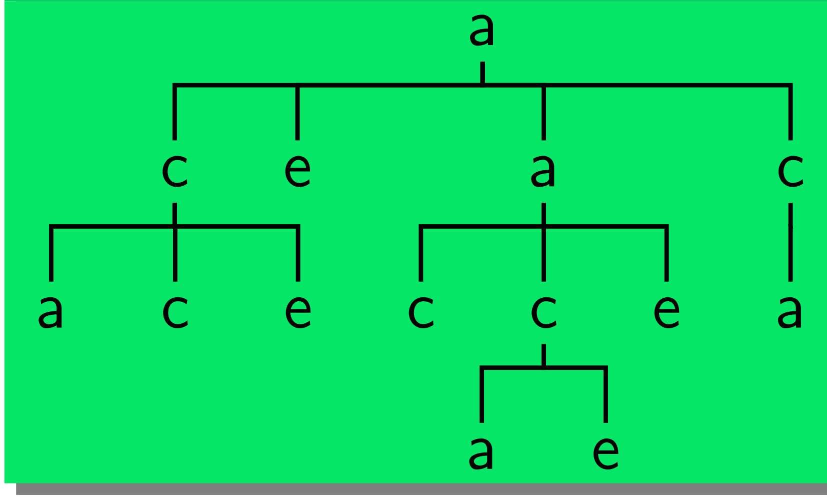
Encoding via ...



first child

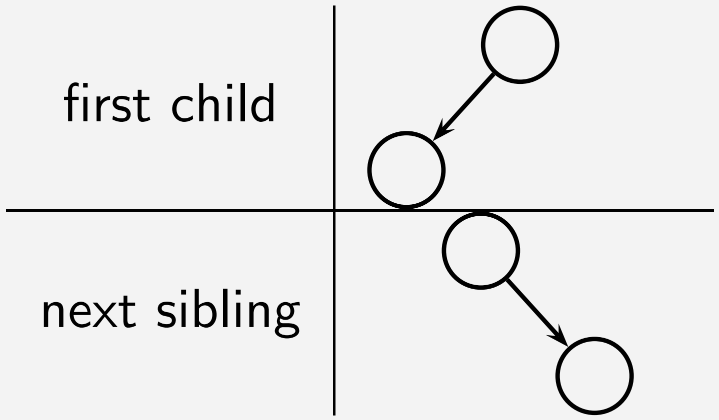
next sibling

Example: Unranked Tree



Encoding Unranked Trees as Binary Trees

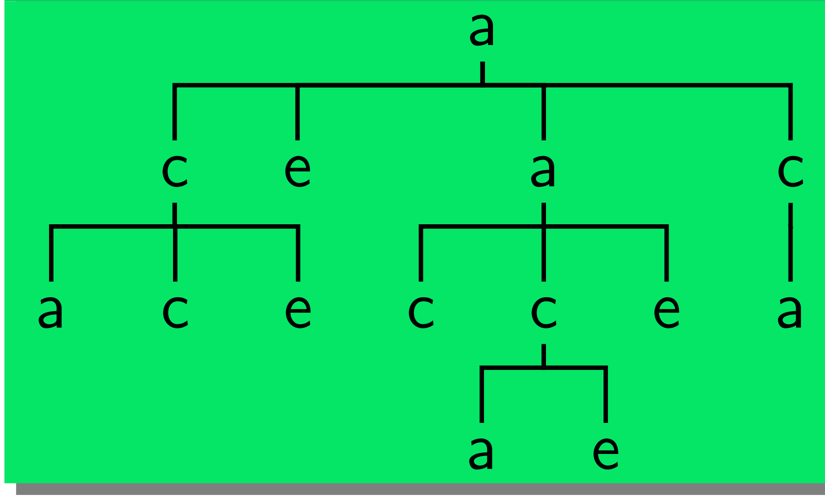
Encoding via ...



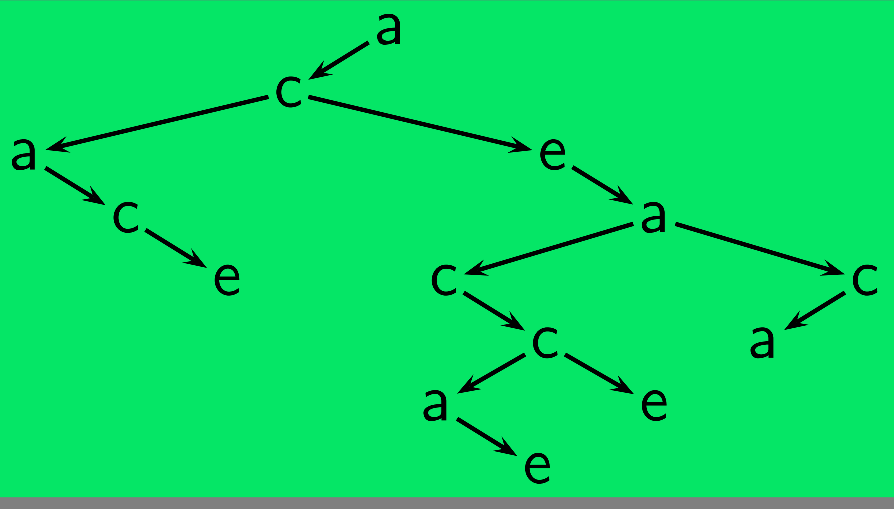
first child

next sibling

Example: Unranked Tree



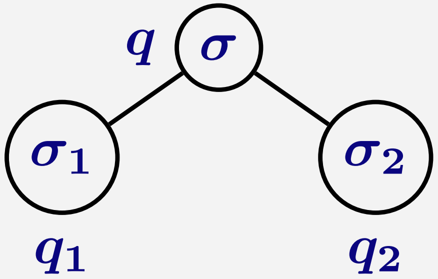
Encoding as Binary Tree



Ranked trees

Transitions are described by finite sets:

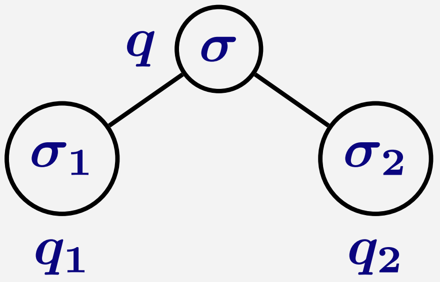
$$\delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\}$$



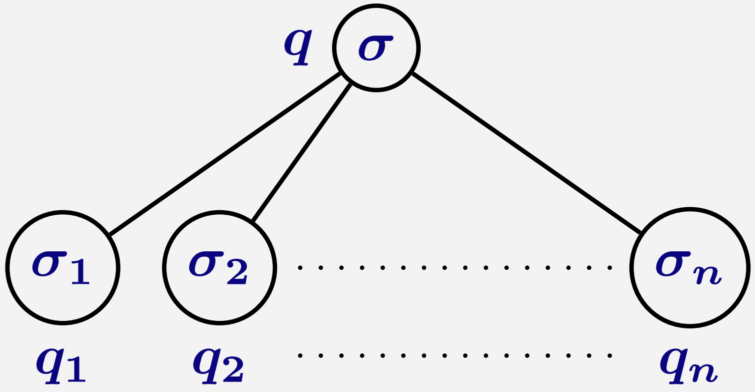
Ranked trees

Transitions are described by finite sets:

$\delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\}$



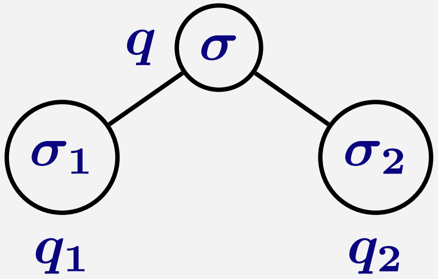
Unranked trees



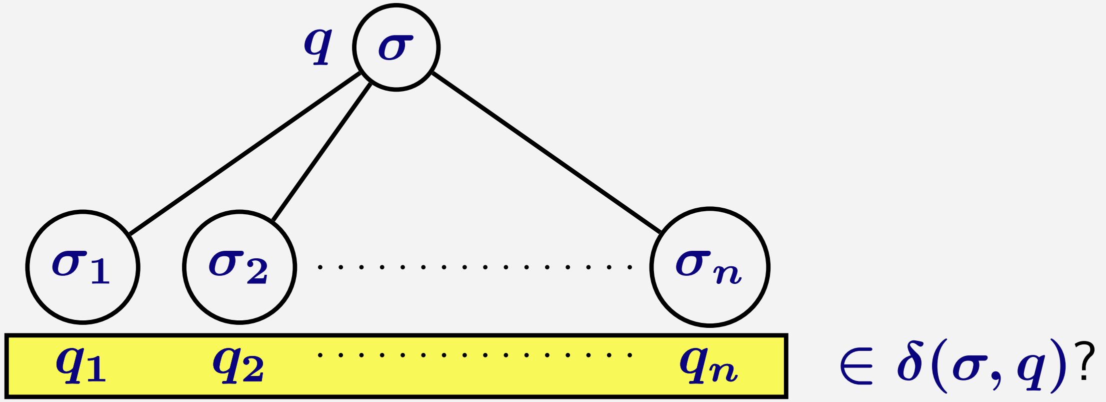
Ranked trees

Transitions are described by finite sets:

$\delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\}$



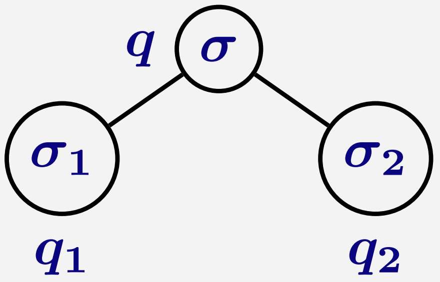
Unranked trees



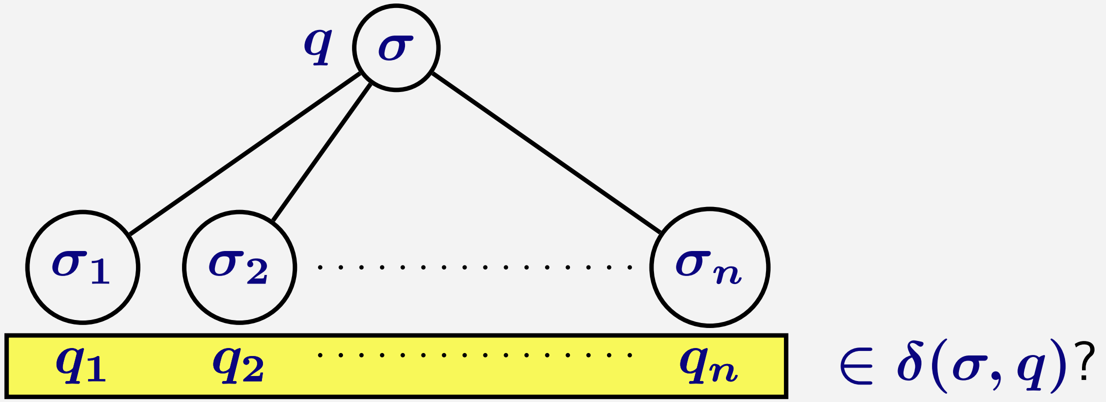
Ranked trees

Transitions are described by finite sets:

$\delta(\sigma, q) = \{(q_1, q_2), (q_3, q_4), \ldots\}$



Unranked trees



$\delta(\sigma, q)$

- For unranked trees, $\delta(\sigma, q)$ is a regular language
- $\delta(\sigma, q)$ can be specified by regular expression or finite string automaton [Brüggemann-Klein, Murata, Wood 2001]

Remark

- Representation of $\delta(\sigma, q)$ has influence on complexity
- Natural choice:
  - For nondeterministic tree automata:
    represent $\delta(\sigma, q)$ by NFAs or regular expressions
  - For deterministic tree automata:
    represent $\delta(\sigma, q)$ by DFAs

$\Rightarrow$ Same complexity results as for ranked trees

Theorem

The following are equivalent for a set L of unranked trees:

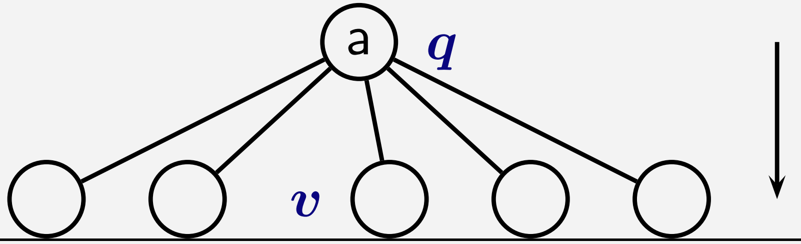
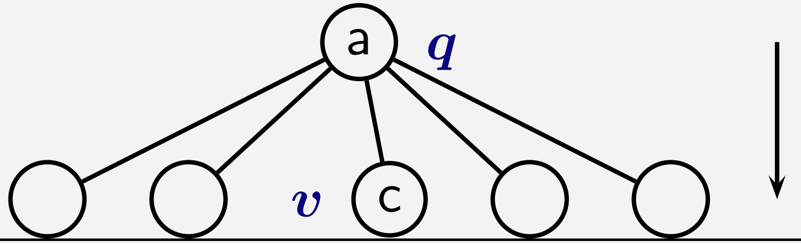
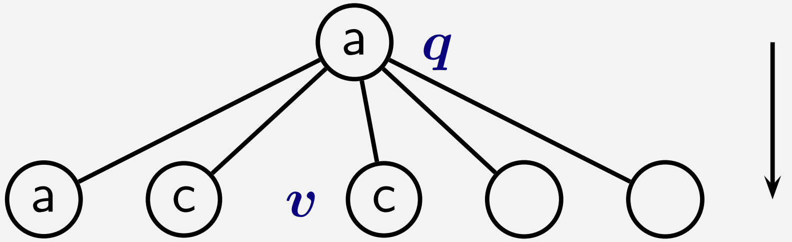
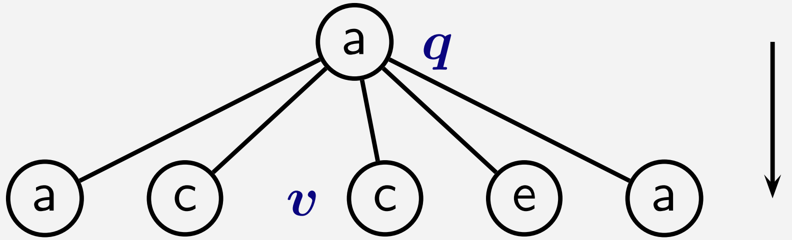
(a) L is accepted by a nondeterministic bottom-up automaton

(b) L is accepted by a deterministic bottom-up automaton

(c) L is accepted by a nondeterministic top-down automaton

(d) L is characterized by an MSO-formula

State at v might depend on ...

state and symbol of parent		
state and symbol of parent and symbol of v		simple
state and symbols of parent and symbols at v and its left siblings		left-siblings aware
state and symbol of parent and symbols at v and its siblings		

Fact

A simple deterministic top-down automata can check the existence of vertical paths with regular properties

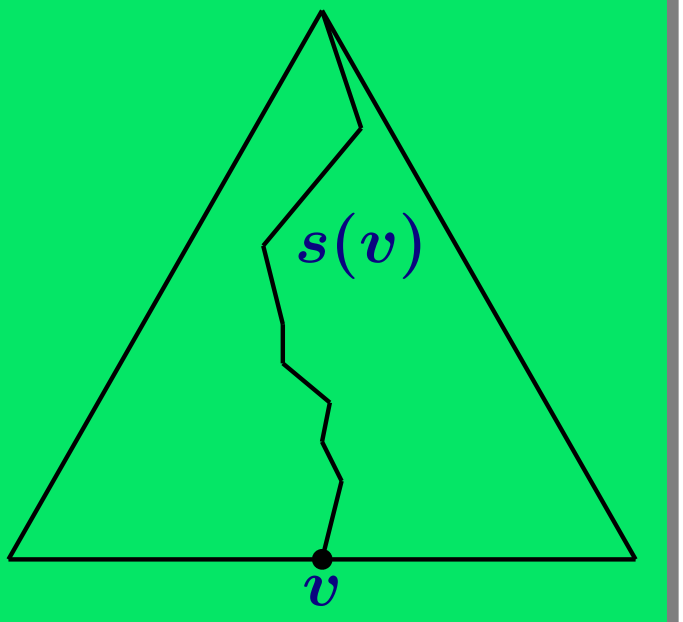
Construction

- For a node v let $s(v)$ denote the sequence of labels from the root to v
- Let $\mathcal{A}$ be a deterministic string automaton
- $\mathcal{A}' :=$ top-down automaton which takes at v state of $\mathcal{A}$ after reading $s(v)$

$\Rightarrow$ $\mathcal{A}'$ is deterministic

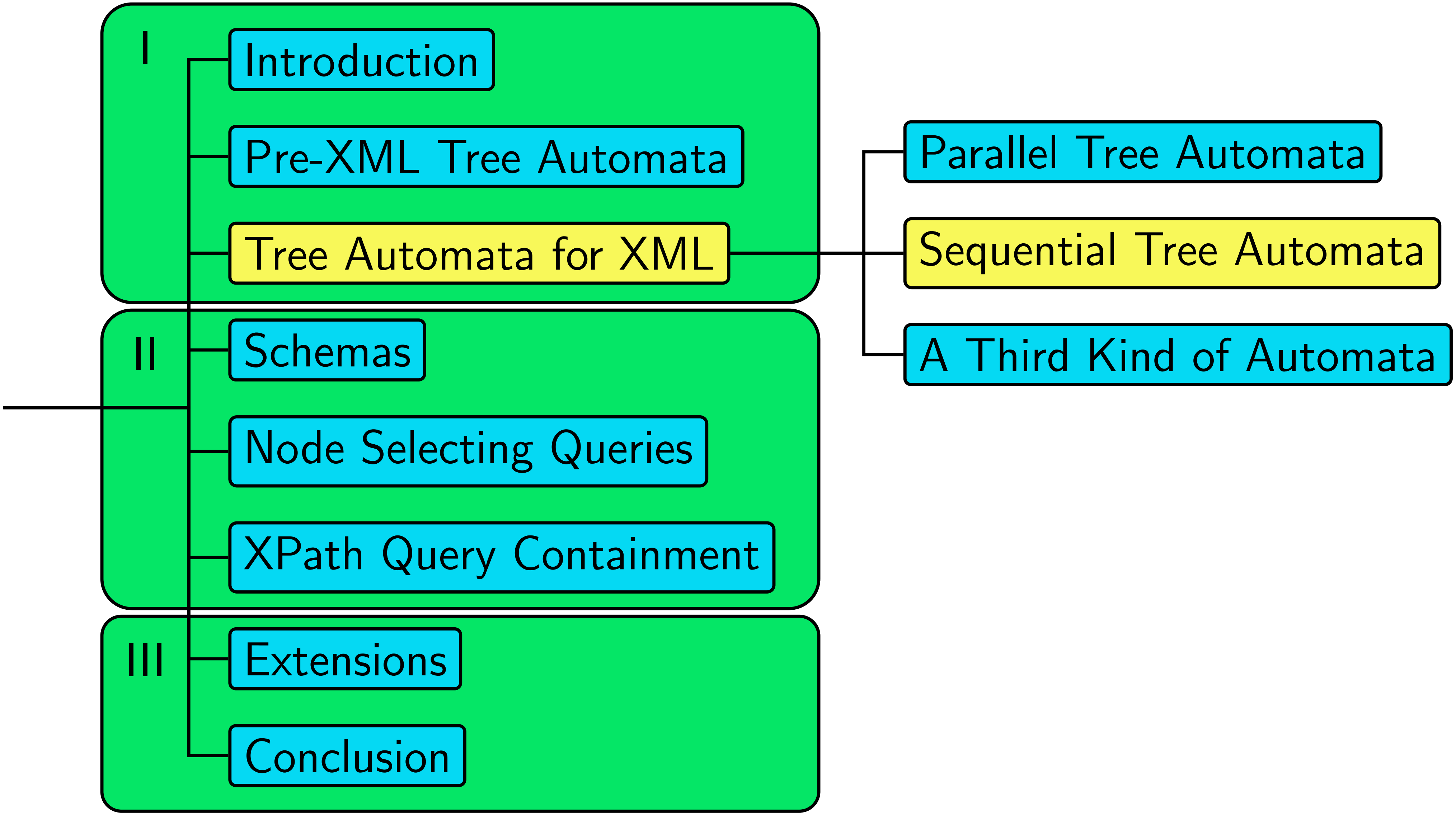
- There is a path from the root to a leaf v with $s(v) \in L(\mathcal{A})$ iff $\mathcal{A}'$ assumes at least one state from F at a leave

Illustration



Streaming XML

Similar construction used for `XPath` evaluation on streams [Green et al. 2003]



I

- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML
  - Parallel Tree Automata
  - Sequential Tree Automata
  - A Third Kind of Automata

II

- Schemas
- Node Selecting Queries
- XPath Query Containment

III

- Extensions
- Conclusion

Generalization of Tree-Walk Automata

Allowed transitions: `Go up`

`Go to first child`

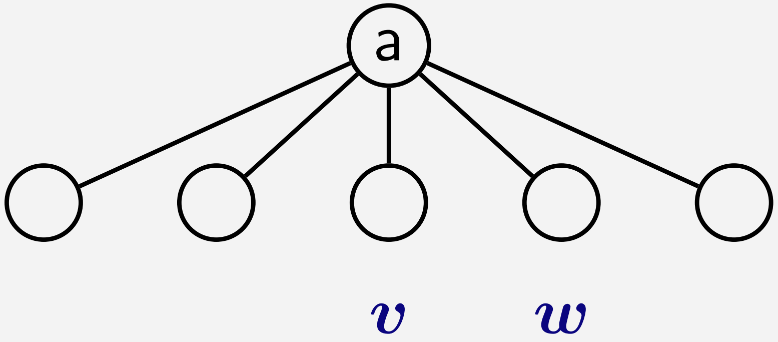
`Go to left sibling`

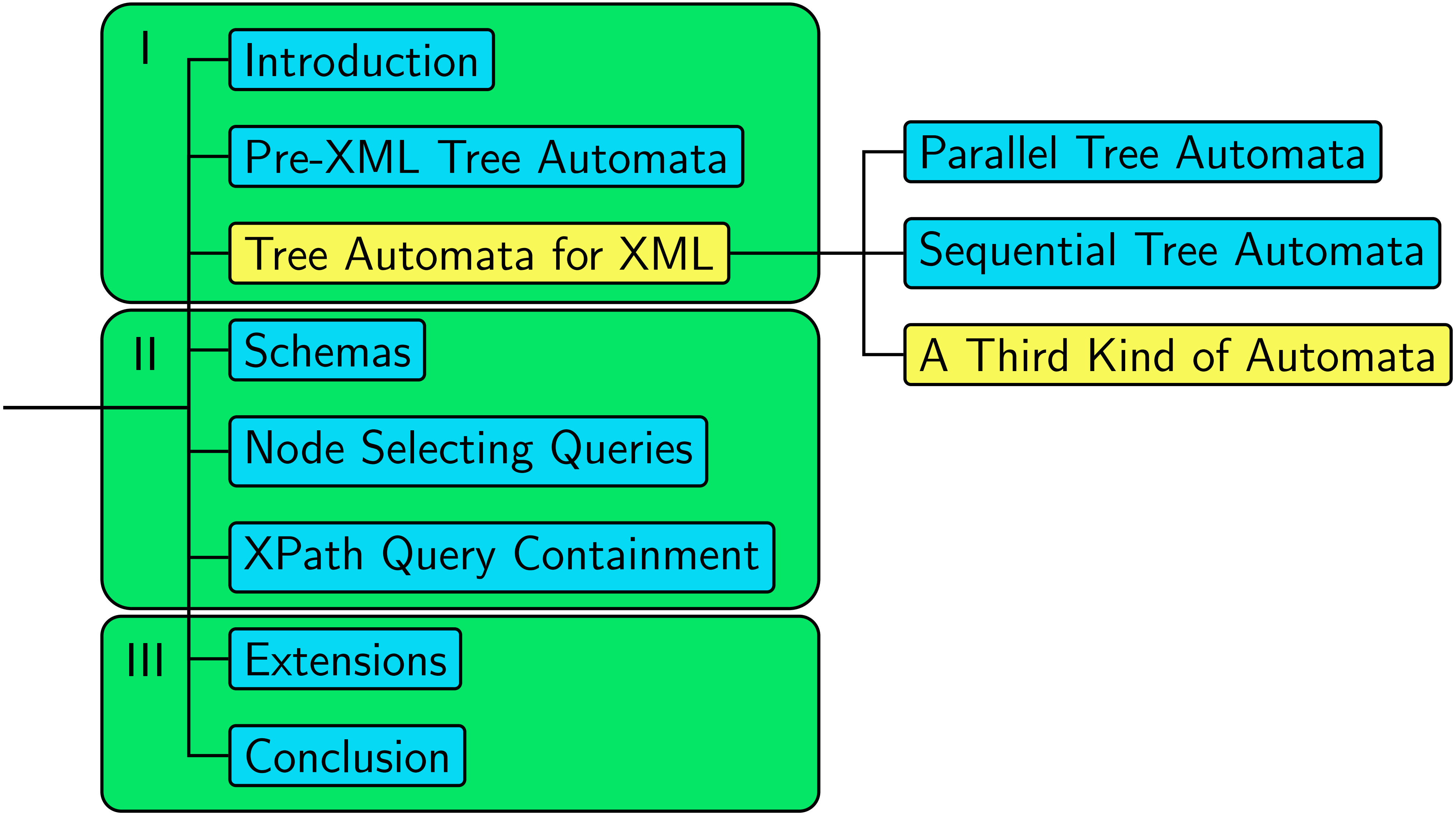
`Go to right sibling`

⟶ Caterpillar automata [Brüggemann-Klein, Wood 2000]

Basic design choice

Should a transition to a sibling be aware of the label of the parent?





- I
  - Introduction
  - Pre-XML Tree Automata
  - Tree Automata for XML
    - Parallel Tree Automata
    - Sequential Tree Automata
    - A Third Kind of Automata
- II
  - Schemas
  - Node Selecting Queries
  - XPath Query Containment
- III
  - Extensions
  - Conclusion

A third kind of automata for XML

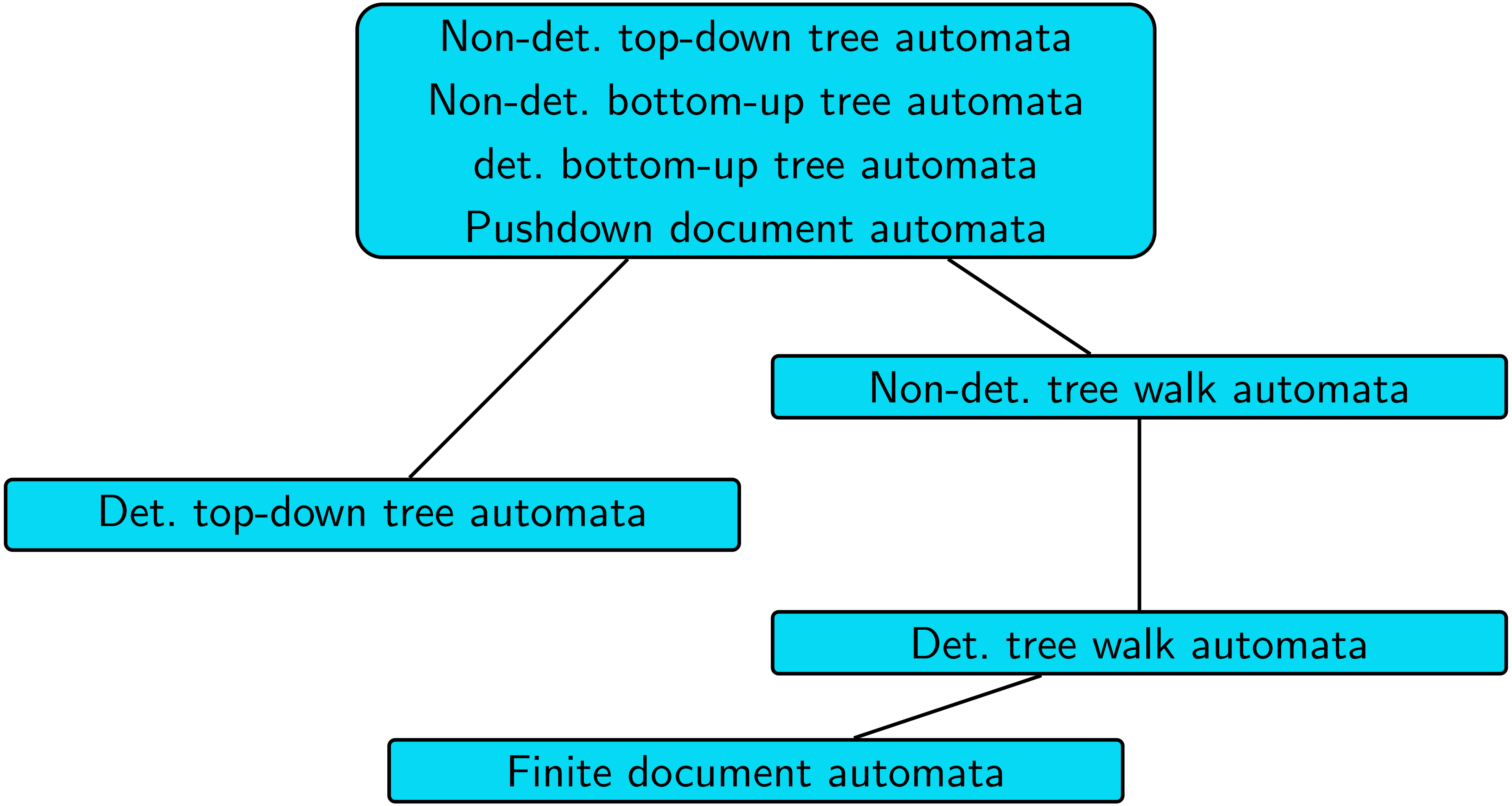
- **Document automata** are string automata reading XML documents as text
- Tags are represented by symbols from a given alphabet
- Variants:
  - Finite document automata
  - Pushdown document automata
- Useful especially in the context of streaming XML

Theorem (Segoufin, Vianu 2002)

- Regular languages of XML-trees can be recognized by deterministic push-down document automata.
- Depth of push-down is bounded by depth of tree

Summary

- Moving from ranked to unranked automata requires some adaptations
- Transitions can be defined with regular string languages $\delta(\sigma, q)$
- By and large, things work smoothly
- In particular:
  - there is an equally robust notion of regular tree languages
  - The complexities are the same as for ranked automata (if the sets $\delta(\sigma, q)$ are represented in a sensible way)



Non-det. top-down tree automata
Non-det. bottom-up tree automata
det. bottom-up tree automata
Pushdown document automata

Non-det. tree walk automata

Det. top-down tree automata

Det. tree walk automata

Finite document automata

I

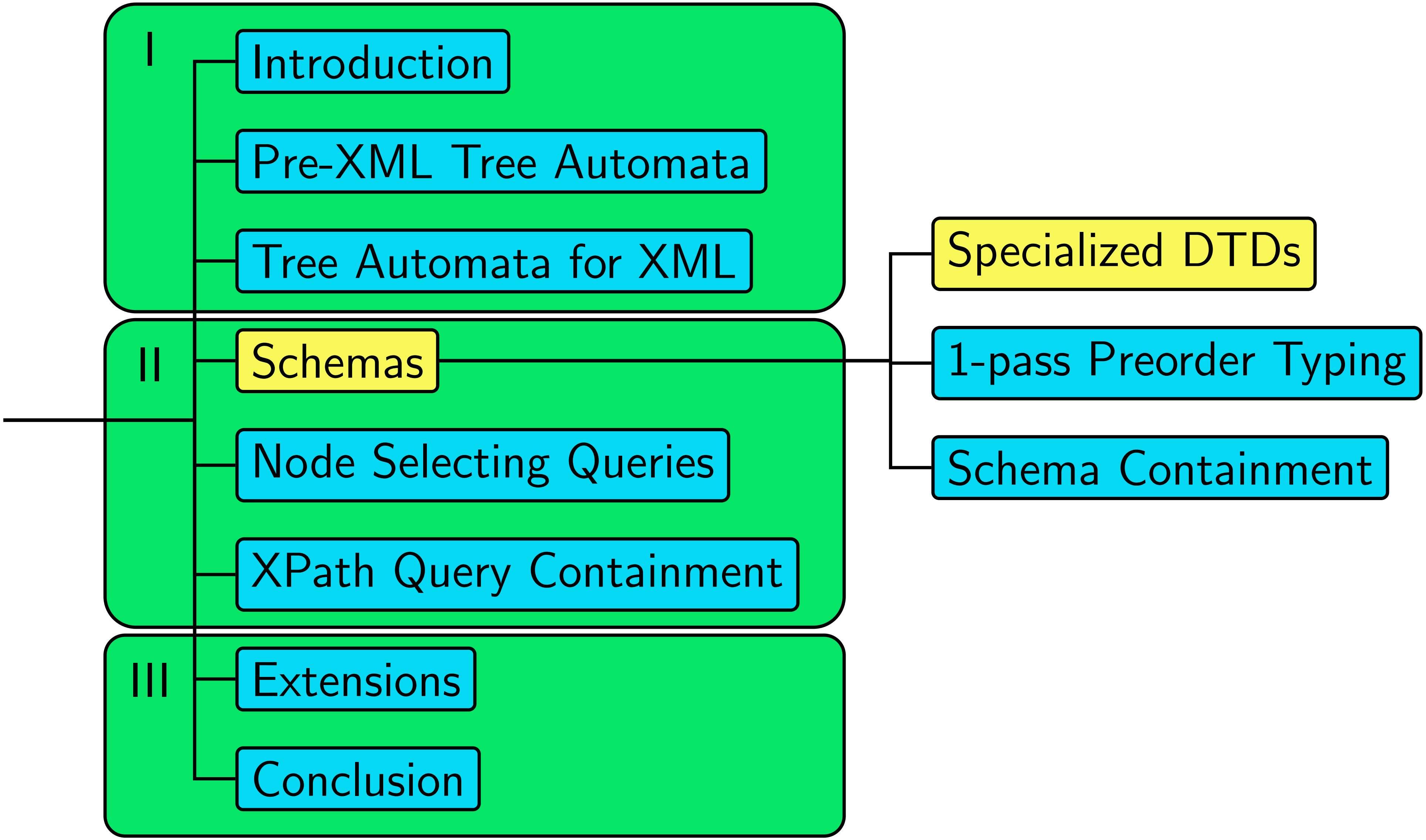
- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II

- Schemas
- Node Selecting Queries
- XPath Query Containment

III

- Extensions
- Conclusion

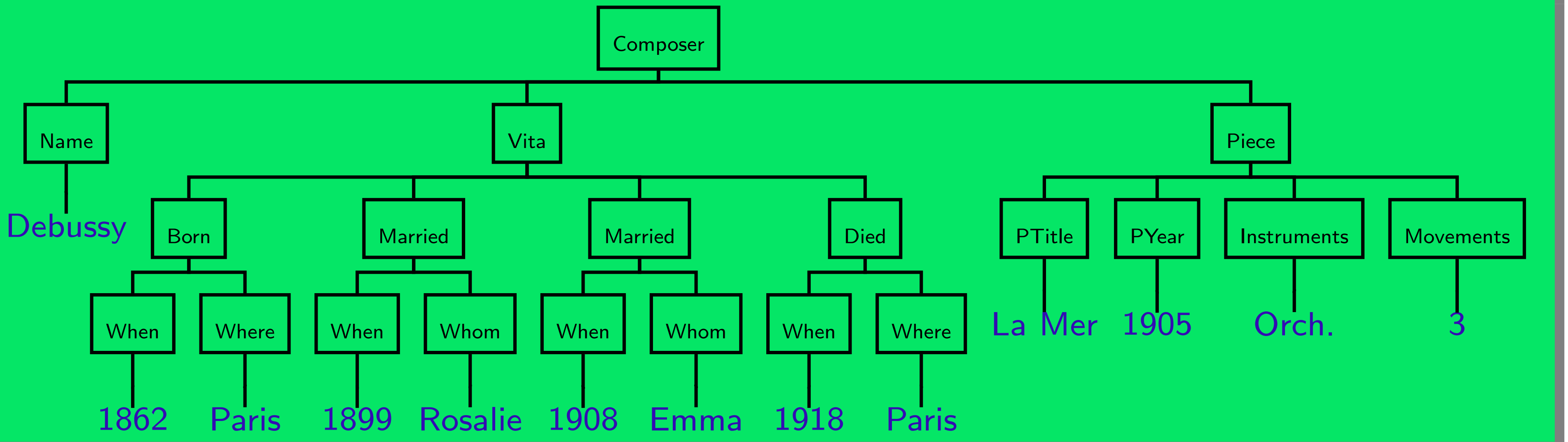


I
- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II
- Schemas
  - Specialized DTDs
  - 1-pass Preorder Typing
  - Schema Containment
- Node Selecting Queries
- XPath Query Containment

III
- Extensions
- Conclusion

Example Tree



Example DTD

```
<!DOCTYPE Composers [
    <!ELEMENT Composers (Composer*)>
    <!ELEMENT Composer (Name, Vita, Piece*)>
    <!ELEMENT Vita (Born, Married*, Died?)>
    <!ELEMENT Born (When, Where)>
    <!ELEMENT Married (When, Whom)>
    <!ELEMENT Died (When, Where)>
    <!ELEMENT Piece (PTitle, PYear,
         Instruments, Movements)>
    ]>
```

Validation Algorithm

For each node:

Check that the children are ok wrt the parent's rule

Observation

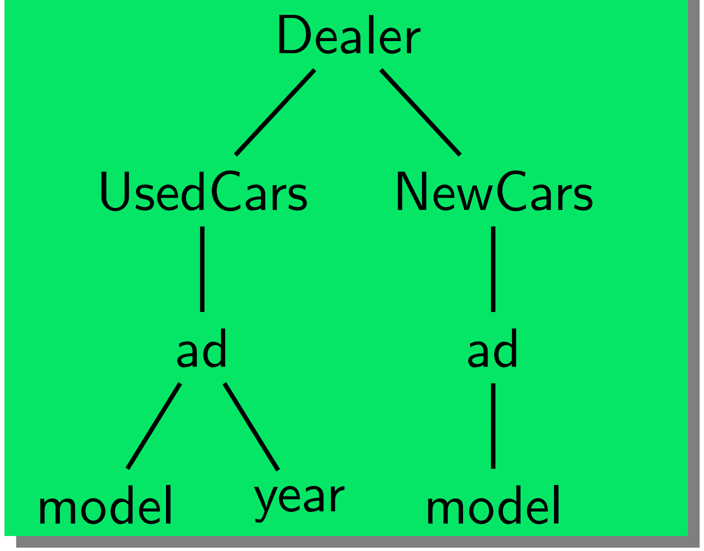
- Validation wrt DTDs is a simple task
- Can be done by
  - Bottom-up automata
  - Deterministic top-down automata (if siblings contribute to new state)
  - Deterministic tree-walk automata: Just a depth-first left-to-right traversal
- In particular: Validation possible in linear time during one pass through the document (**1-pass validation**)
- But DTDs are also rather weak...

A classical example

```
<!DOCTYPE Dealer [
    <!ELEMENT Dealer (UsedCars NewCars)>
    <!ELEMENT UsedCars (ad*)>
    <!ELEMENT NewCars (ad*)>
    <!ELEMENT ad ((model, year) | model)> ]>
```

Intention

Intention:



Observation

- Elements with the same name may have different structure in different contexts

→ It would be nice to have types for elements

→ Specialized DTDs

Definition (Papakonstantinou, Vianu 2000)

A **specialized DTD** (SDTD) over alphabet Σ is a pair (d, μ), where

- d is a DTD over the alphabet Σ' of **types**
- $\mu : \Sigma' \longrightarrow \Sigma$ maps types to tag names

Note

Concerning the name:

"specialized" refers to types, not to DTDs

Example

Dealer $\longrightarrow$ UsedCars NewCars

UsedCars $\longrightarrow$ adUsed*

NewCars $\longrightarrow$ adNew*

adUsed $\longrightarrow$ model year

adNew $\longrightarrow$ model

μ(Dealer) $=$ Dealer

μ(UsedCars) $=$ UsedCars

μ(NewCars) $=$ NewCars

μ(adUsed) $=$ ad

μ(adNew) $=$ ad

Example: SDTD for Boolean circuit trees

1-AND → (1-OR | 1-AND | 1-leaf)*

1-OR → .* (1-OR | 1-AND | 1-leaf) .*

0-AND → .* (0-OR | 0-AND | 0-leaf) .*

0-OR → (0-OR | 0-AND | 0-leaf)*

1-leaf → ϵ

0-leaf → ϵ

Tag	h(Tag)
1-AND	AND
0-AND	AND
1-OR	OR
0-OR	OR
1-leaf	1
0-leaf	0

Observation

- A tree conforms to a specialized DTD (d, μ) if there is a labeling of its nodes by types which is valid wrt. d
-

Observation

- A tree conforms to a specialized DTD (d, μ) if there is a labeling of its nodes by types which is valid wrt. d
- This reminds us of something...

Observation

- A tree conforms to a specialized DTD (d, μ) if there is a labeling of its nodes by types which is valid wrt. d
- This reminds us of something...

Theorem

Specialized DTDs capture exactly the regular tree languages

Observation

- A tree conforms to a specialized DTD (d, μ) if there is a labeling of its nodes by types which is valid wrt. d
- This reminds us of something...

Theorem

Specialized DTDs capture exactly the regular tree languages

Question: What about 1-pass validation?

Definition (Validation)

Given: Specialized DTD d, tree t

Qeustion: Is t valid wrt d?

Definition (Typing)

Given: Specialized DTD d, tree t

Output: Consistent type assignment for the nodes of t

Facts

- Specialized DTDs $\equiv$ regular tree languages

$\longrightarrow$ Validation by a deterministic push-down automaton

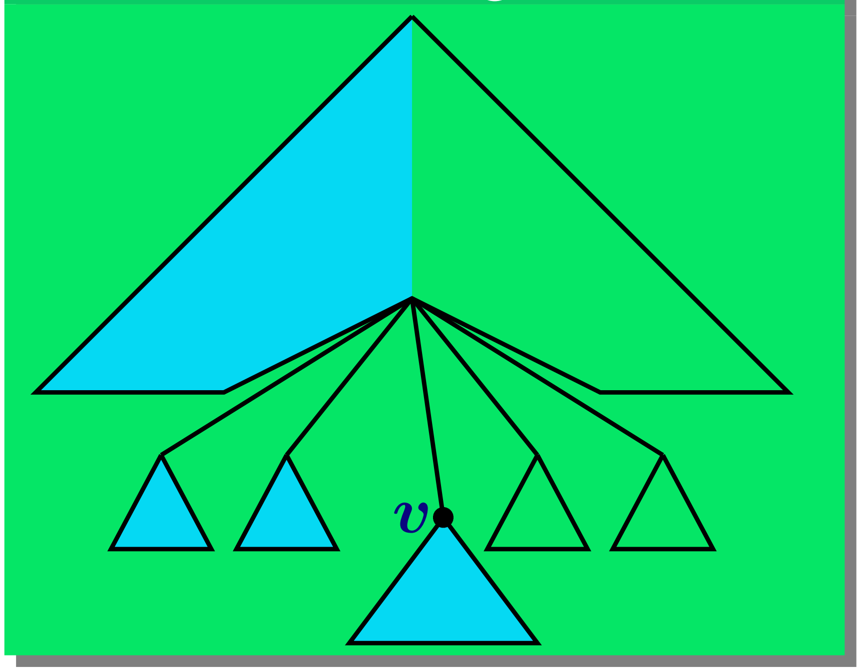
- Validation in linear time during one pass through the document

Question: What about 1-pass typing?

Observations

- Type of a node $\equiv$ state of deterministic bottom-up automaton
- Deterministic push-down automaton can assign types during 1 pass
- But the type of a node v is determined after visiting its subtree
-

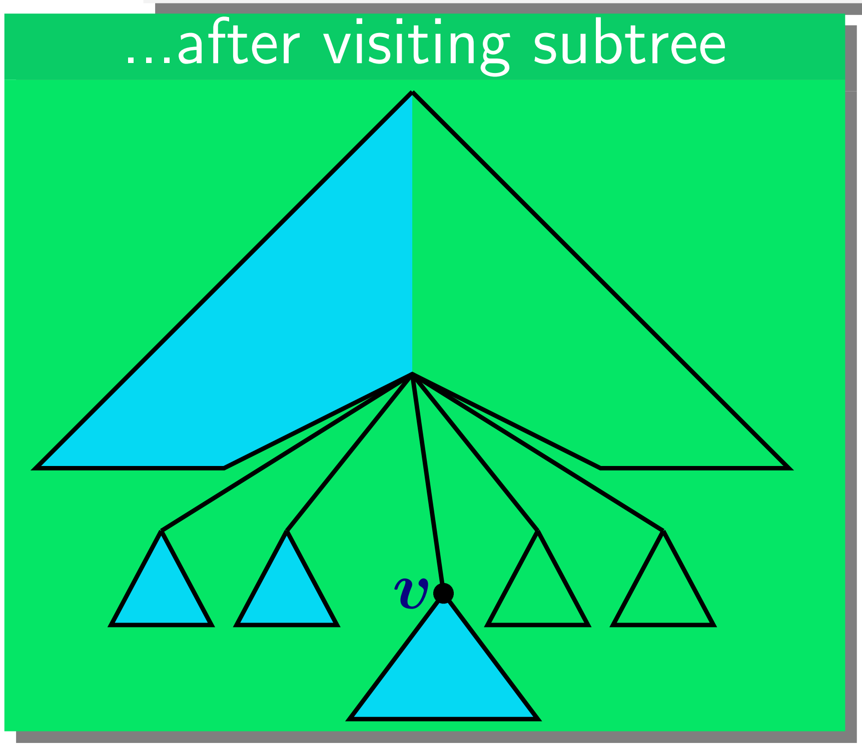
...after visiting subtree

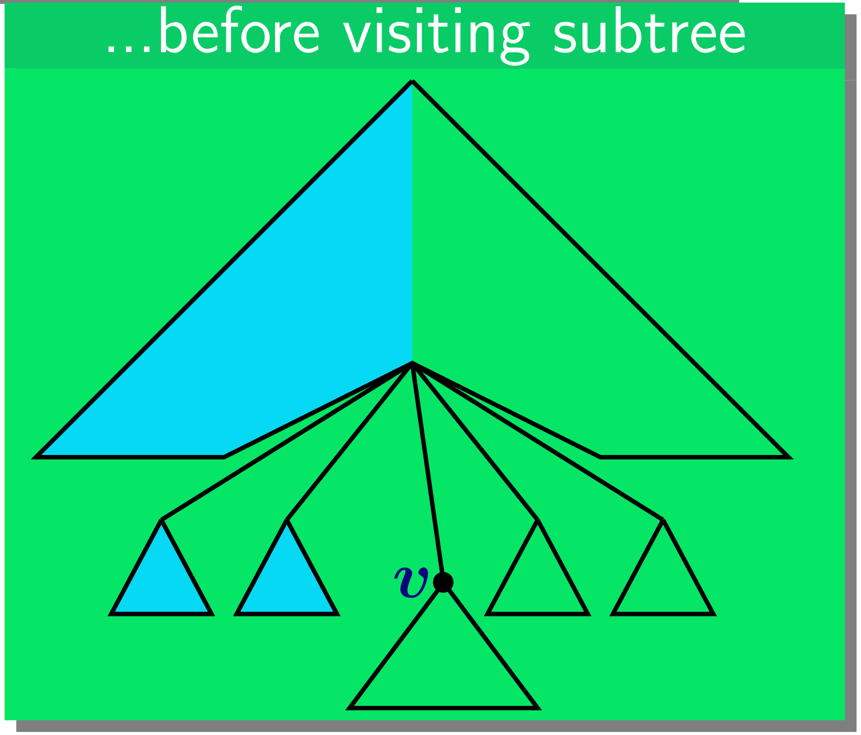


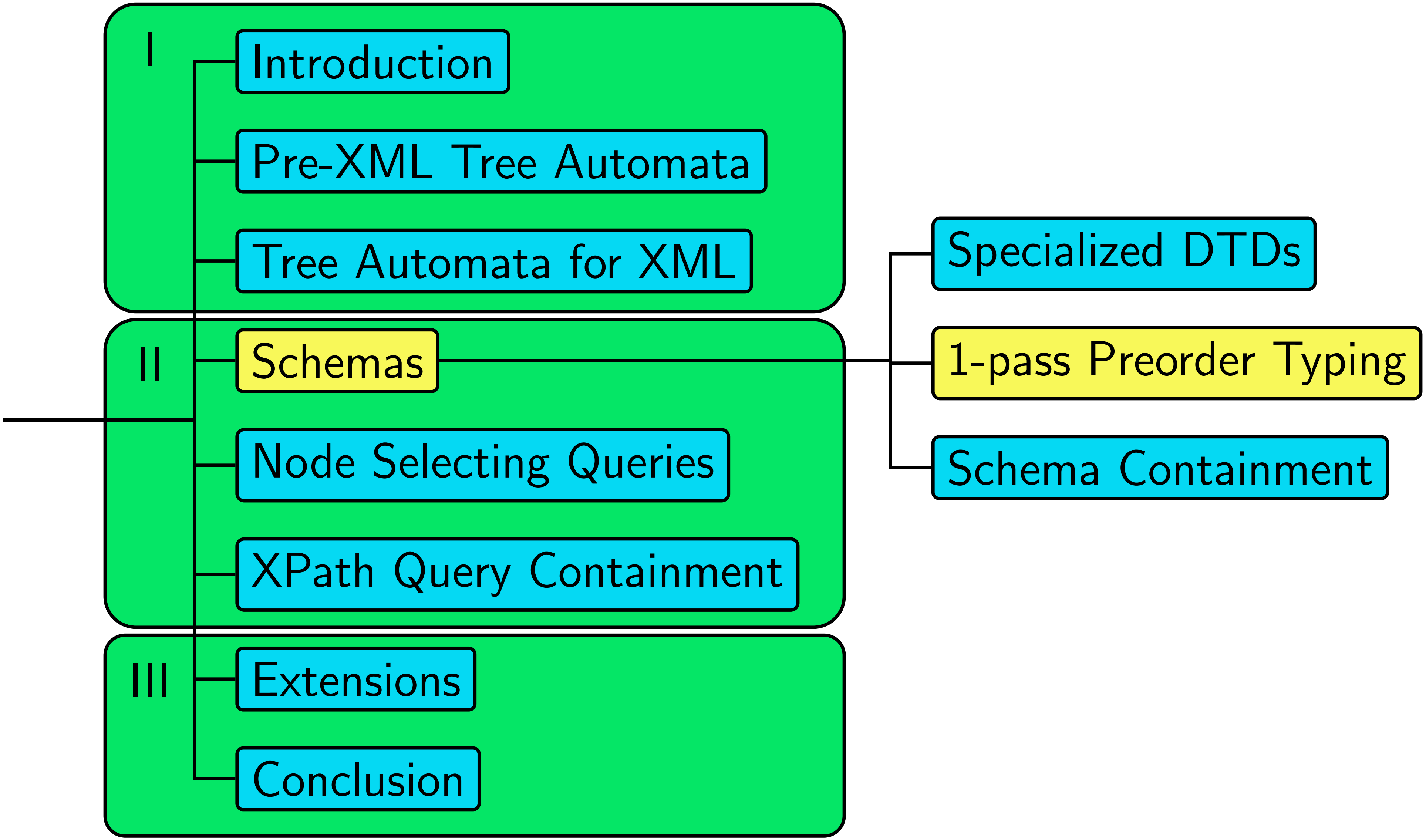
Observations

- Type of a node $\equiv$ state of deterministic bottom-up automaton
- Deterministic push-down automaton can assign types during 1 pass
- But the type of a node v is determined after visiting its subtree
- 1-pass preorder typing:

  determine type of v before visiting the subtree of v


...after visiting subtree


...before visiting subtree



I

- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II

- Schemas
  - Specialized DTDs
  - 1-pass Preorder Typing
  - Schema Containment
- Node Selecting Queries
- XPath Query Containment

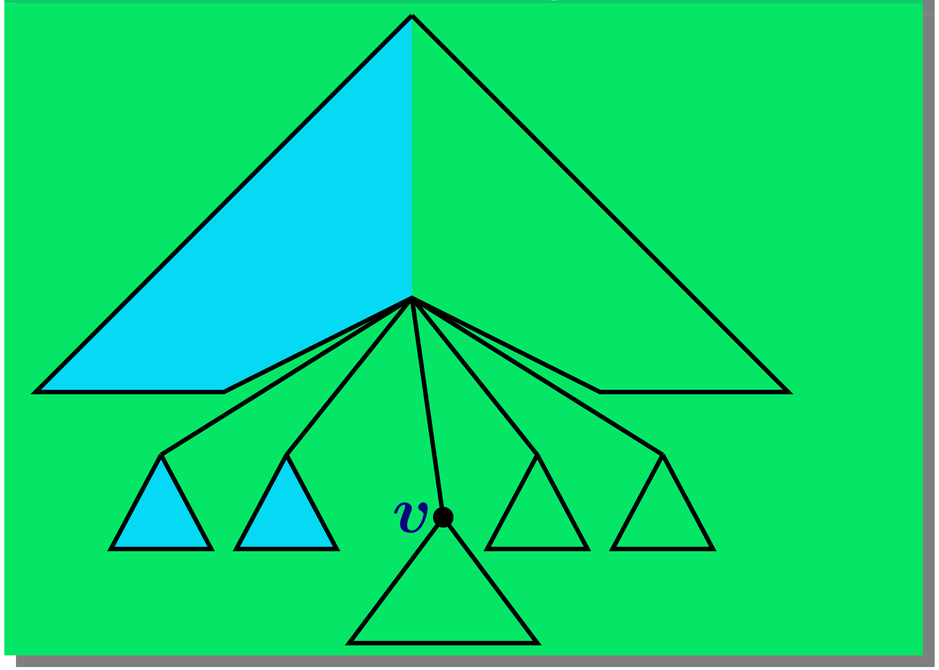
III

- Extensions
- Conclusion

Question

When would it be important to know the type of $\boldsymbol{v}$ before visiting the subtree of $\boldsymbol{v}$?

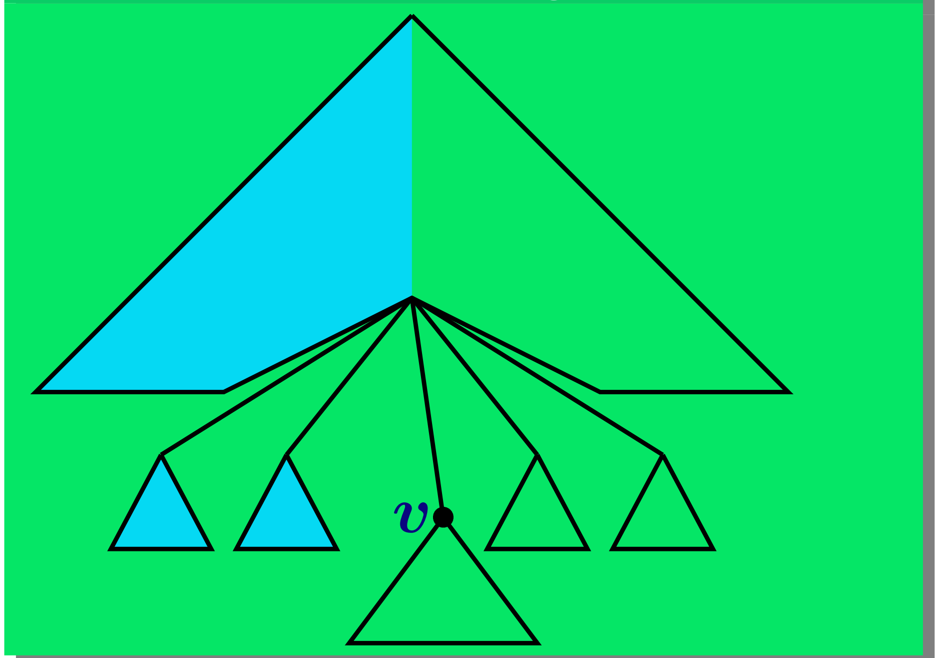
…before visiting subtree



Question

When would it be important to know the type of v before visiting the subtree of v?

…before visiting subtree



Answer

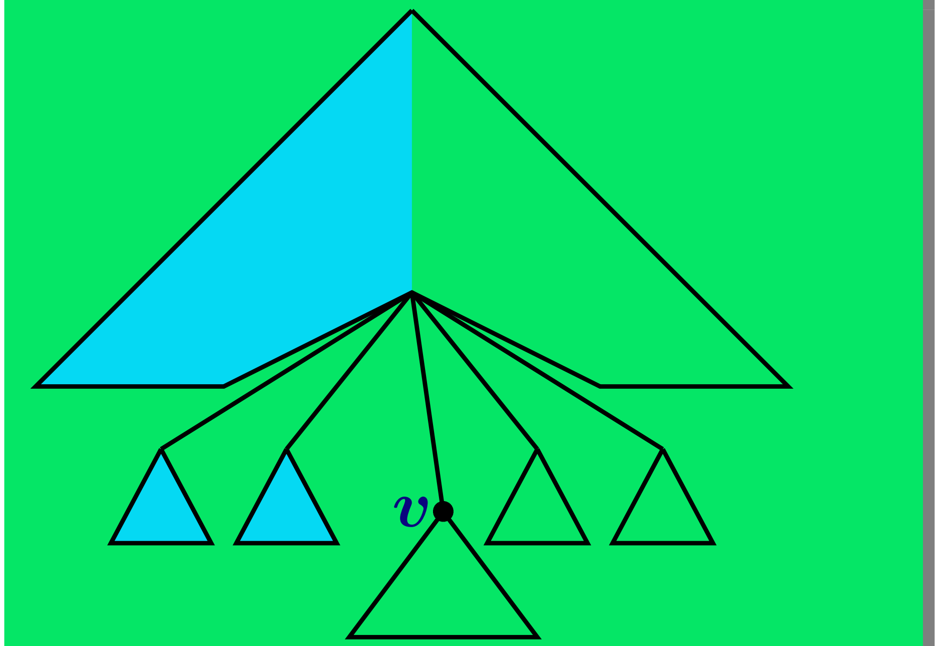
Whenever the processing proceeds in document order, e.g.:

- Streaming XML: Typing as the first operator in a pipeline
- SAX-based processing

Question

When would it be important to know the type of v before visiting the subtree of v?

…before visiting subtree



Answer

Whenever the processing proceeds in document order, e.g.:

- Streaming XML: Typing as the first operator in a pipeline
- SAX-based processing

Our next goal

Find out which schemas admit 1-pass preorder typing

Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types b, b' compete if $\mu(b) = \mu(b')$
- A specialized DTD is single-type if no competing types occur in the same rule (e.g., $a \longrightarrow bcb'$ is not single-type)
- A specialized DTD is restrained-competition if no rule allows strings wbv, $wb'v'$ with competing types b, b'
  (e.g., $a \longrightarrow c(b + d^* b')$ is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

→ Two types b, b' **compete** if $\mu(b) = \mu(b')$

- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)
- A specialized DTD is **restrained-competition** if no rule allows strings wbv, $wb'v'$ with competing types b, b'
  (e.g., $a \rightarrow c(b + d^*b')$ is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types b, b' compete if $\mu(b) = \mu(b')$

→ A specialized DTD is single-type if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)

- A specialized DTD is restrained-competition if no rule allows strings wbv, $wb'v'$ with competing types b, b'
  (e.g., $a \rightarrow c(b + d^*b')$ is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types b, b' **compete** if $\mu(b) = \mu(b')$
- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \longrightarrow bcb'$ is not single-type)

→ A specialized DTD is **restrained-competition** if no rule allows strings wbv, $wb'v'$ with competing types b, b' (e.g., $a \longrightarrow c(b + d^*b')$ is not restrained-competition)

- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types b, b' **compete** if $\mu(b) = \mu(b')$
- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \longrightarrow bcb'$ is not single-type)
- A specialized DTD is **restrained-competition** if no rule allows strings wbv, $wb'v'$ with competing types b, b'
(e.g., $a \longrightarrow c(b + d^*b')$ is not restrained-competition)

→ The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types b, b' **compete** if $\mu(b) = \mu(b')$
- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)
- A specialized DTD is **restrained-competition** if no rule allows strings wbv, $wb'v'$ with competing types b, b' (e.g., $a \rightarrow c(b + d^*b')$ is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Fact

Both restrictions are sufficient to get 1-pass preorder typing!

Restricted Schemas

(Murata, Lee, Mani 2001) introduced* restrictions on specialized DTDs to ensure efficient validation (*: in a slightly different framework)

- Two types b, b' **compete** if $\mu(b) = \mu(b')$
- A specialized DTD is **single-type** if no competing types occur in the same rule (e.g., $a \rightarrow bcb'$ is not single-type)
- A specialized DTD is **restrained-competition** if no rule allows strings wbv, $wb'v'$ with competing types b, b'
  (e.g., $a \rightarrow c(b + d^*b')$ is not restrained-competition)
- The authors argue that XML-Schema roughly corresponds to single-type SDTDs

Fact

Both restrictions are sufficient to get 1-pass preorder typing!

Question: Are they also necessary?

1-Pass Preorder Typing (cont.)

Remarks

- The definition of “1-pass preorder typing” does not yet restrict the efficiency of determining the type of a node
- Typing could be 1-pass preorder but very time consuming
- It turns out that essentially this never happens

Remarks

- The definition of “1-pass preorder typing” does not yet restrict the efficiency of determining the type of a node
- Typing could be 1-pass preorder but very time consuming
- It turns out that essentially this never happens

Theorem (Martens, Neven, Sch. 2004)

For a regular tree language L the following are equivalent

(a) L can be described by a 1-pass preorder typable SDTD

(b) L can be described by a restrained-competition SDTD

(c) L has linear time 1-pass pre-order typing

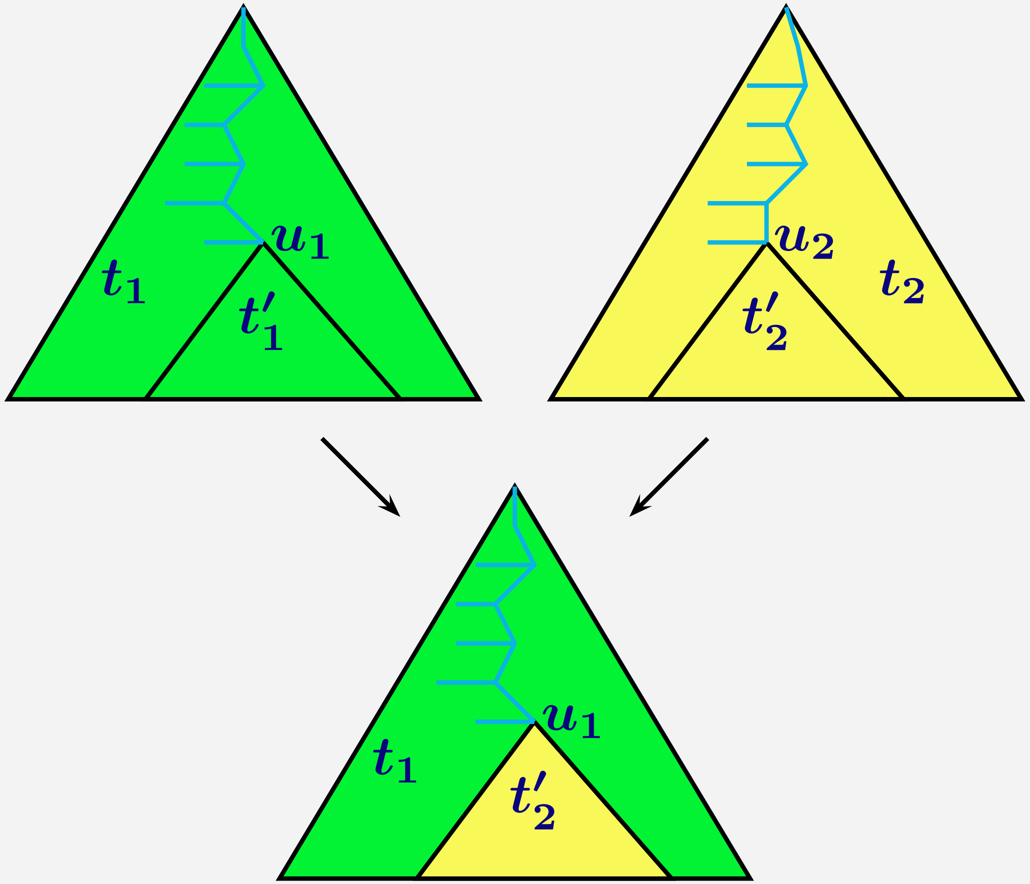
(d) L can be preorder-typed by a deterministic pushdown document automaton

(e) Types for trees in L can be computed by a left-siblings-aware top-down deterministic tree automaton

Further characterizations

- This class has further interesting characterizations
- E.g., by closure under ancestor-sibling-guarded subtree exchange

Illustration



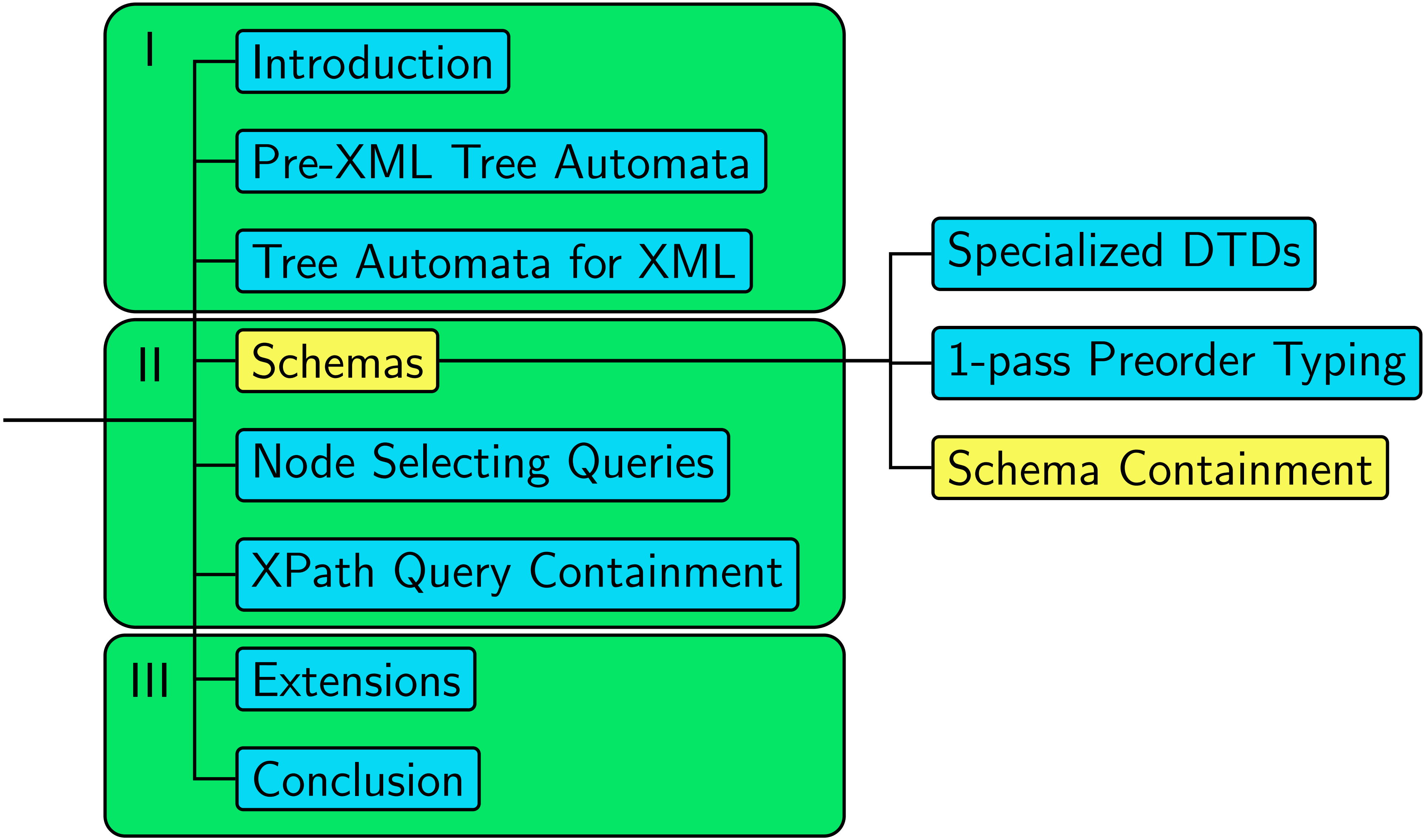
Theorem (Martens, Neven, Sch. 2004)

For a regular tree language L the following are equivalent

(a) L can be described by a single-type SDTD

(b) Types for trees in L can be computed by a simple top-down deterministic tree automaton

(c) L is closed under ancestor-guarded subtree exchange



I

Introduction

Pre-XML Tree Automata

Tree Automata for XML

II

Schemas

- Specialized DTDs
- 1-pass Preorder Typing
- Schema Containment

Node Selecting Queries

XPath Query Containment

III

Extensions

Conclusion

Schema Containment

Given: Schemas d_1, d_2

Question: Is $L(d_1) \subseteq L(d_2)$?

Observations

- Important, e.g., for data integration
- Recall: Specialized DTDs are essentially non-deterministic tree automata

$\Rightarrow$ Containment of specialized DTDs is in **EXPTIME**

- But the restricted forms have lower complexity
- Complexity of containment depends on the allowed regular expressions

Schema Containment: Complexity

Results (partly from Martens, Neven, Sch. 2004)

Schema type	unrestricted	deterministic expressions
DTDs	**PSPACE**	**PTIME**
single-type SDTDs	**PSPACE**	**PTIME**
restrained-competition SDTDs	**PSPACE**	**PTIME**
unrestricted SDTDs	**EXPTIME**	**EXPTIME**

Observations

- For unrestricted SDTDs the complexity is dominated by tree automata containment
- For the others it is dominated by the sub-task of checking containment for regular expressions

Observations (cont.)

- … for the others it is dominated by the sub-task of checking containment for regular expressions
- Actually, this observation can be made more precise

Theorem (Martens, Neven, Sch. 2004)

For a class $\mathcal{R}$ of regular expressions and a complexity class $\mathcal{C}$, the following are equivalent

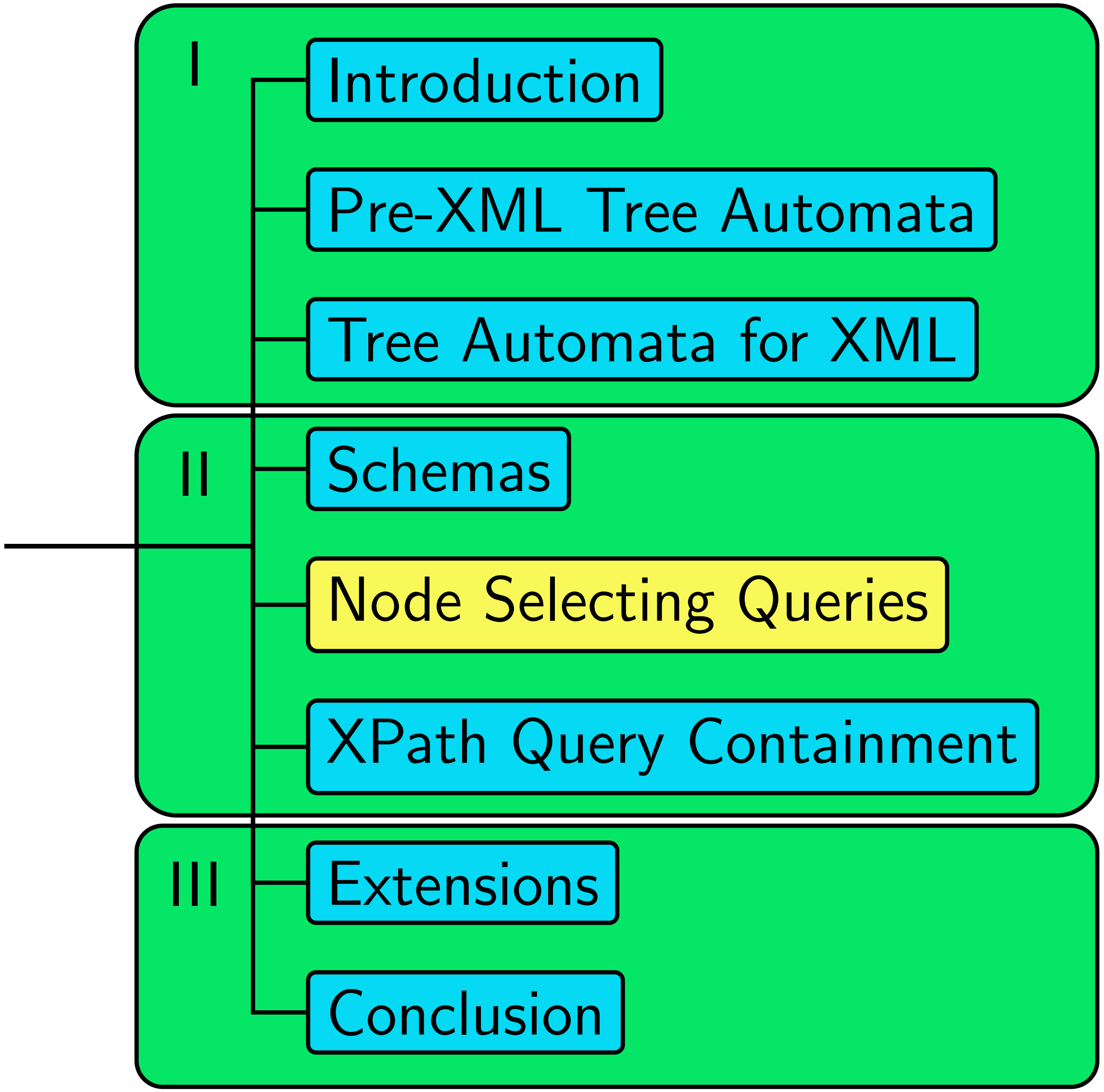
(a) The containment problem for $\mathcal{R}$ expressions is in $\mathcal{C}$.

(b) The containment problem for DTDs with regular expressions from $\mathcal{R}$ is in $\mathcal{C}$.

(c) The containment problem for single-type SDTDs with regular expressions from $\mathcal{R}$ is in $\mathcal{C}$.

Summary

- Regular tree languages are a nice framework for schema languages
  - Linear time validation
  - Static analysis is expensive
- They also serve as a basis for restricted classes with better algorithmic properties:
  - 1-pass preorder typing
  - more feasible static analysis, in particular if the $\delta(\sigma, q)$ are given by deterministic automata
- Restrained competition $\equiv$ Deterministic top-down automata $\equiv$ 1-pass preorder typable



I

Introduction

Pre-XML Tree Automata

Tree Automata for XML

II

Schemas

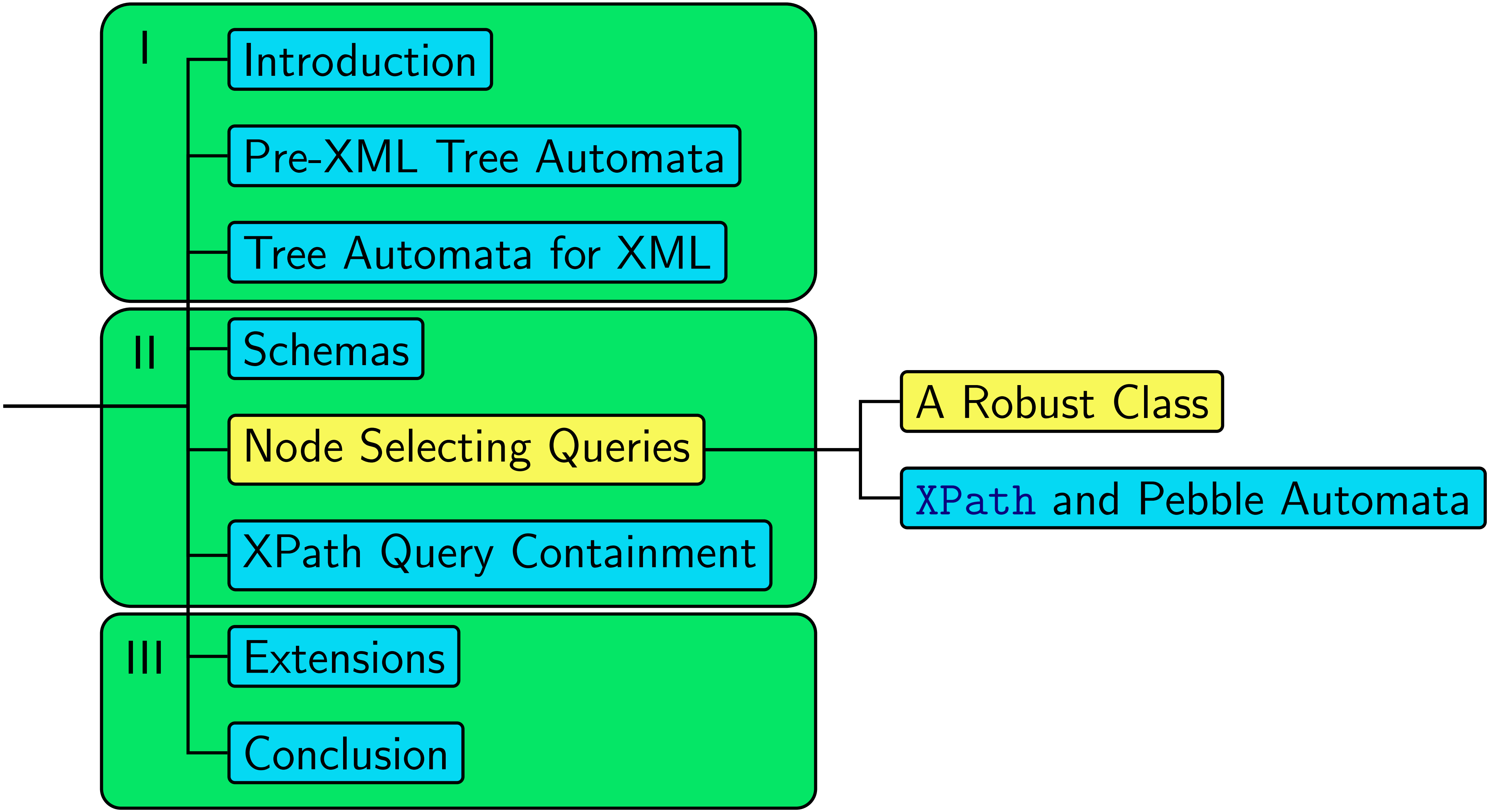
Node Selecting Queries

XPath Query Containment

III

Extensions

Conclusion



I

Introduction

Pre-XML Tree Automata

Tree Automata for XML

II

Schemas

Node Selecting Queries

- A Robust Class
- `XPath` and Pebble Automata

XPath Query Containment

III

Extensions

Conclusion

Example document

```
⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
    ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
    ⟨Movements⟩ 3 ⟨/Movements⟩
    ...
  ⟨/Piece⟩
  ...
⟨/Composer⟩
...
```

Example query

`//Vita/Died/*`

Example document

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Example query

`//Vita/Died/*`

Observation

`XPath` expressions define sets of nodes → node-selecting queries

Question

Is there a class of node-selecting queries, as robust as the regular tree languages?

Observation

- There is a simple way to define node selecting queries by monadic second-order formulas:
- Simply use one free variable: $\varphi(x)$
- Is there a corresponding automaton model?
- It is relatively easy to add node selection to nondeterministic bottom-up automata

Definition (Nondetermistic bottom-up node-selecting automata)

- Nondeterministic bottom-up automata plus select function:

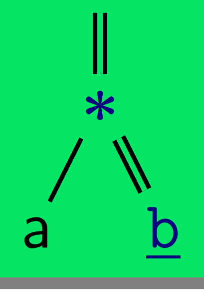
$$s : Q \times \Sigma \to \{0, 1\}$$

- Node v is in result set for tree t $:\Longleftrightarrow$ there is an accepting computation on t in which v gets a state q such that $s(q, \lambda(v)) = 1$

Example query

//*[a]//b

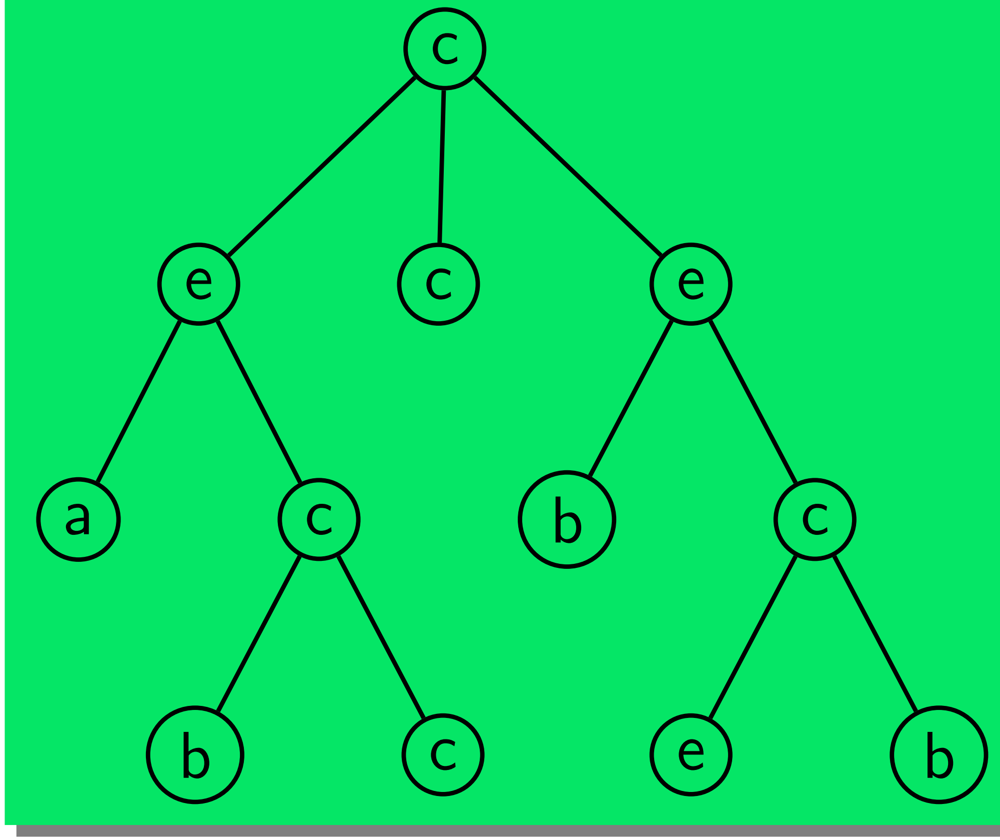
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$
  $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

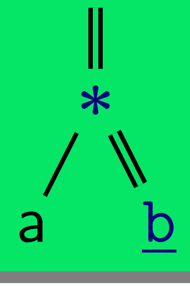
Example tree: Run 1



Example query

//$*$[a]//b

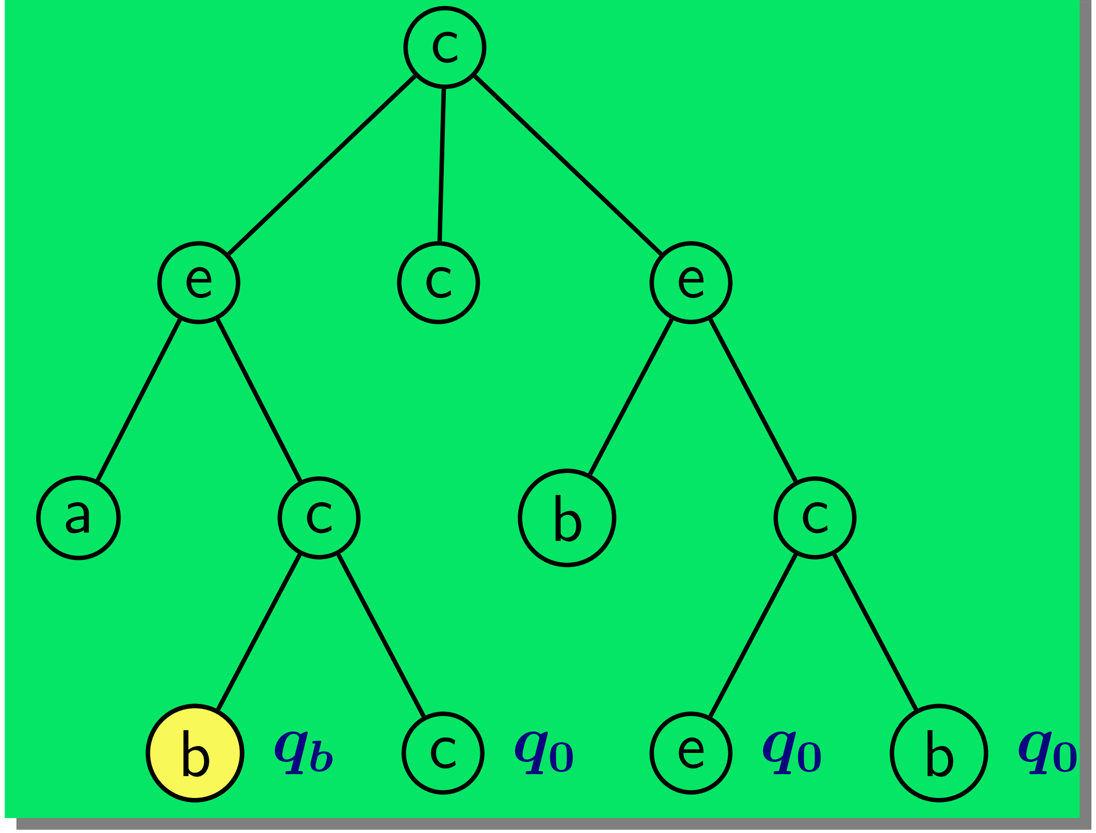
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$
  $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

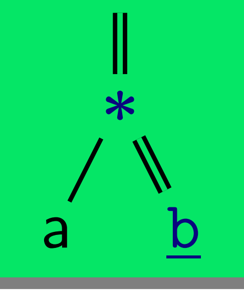
Example tree: Run 1



Example query

//$*$[a]//b

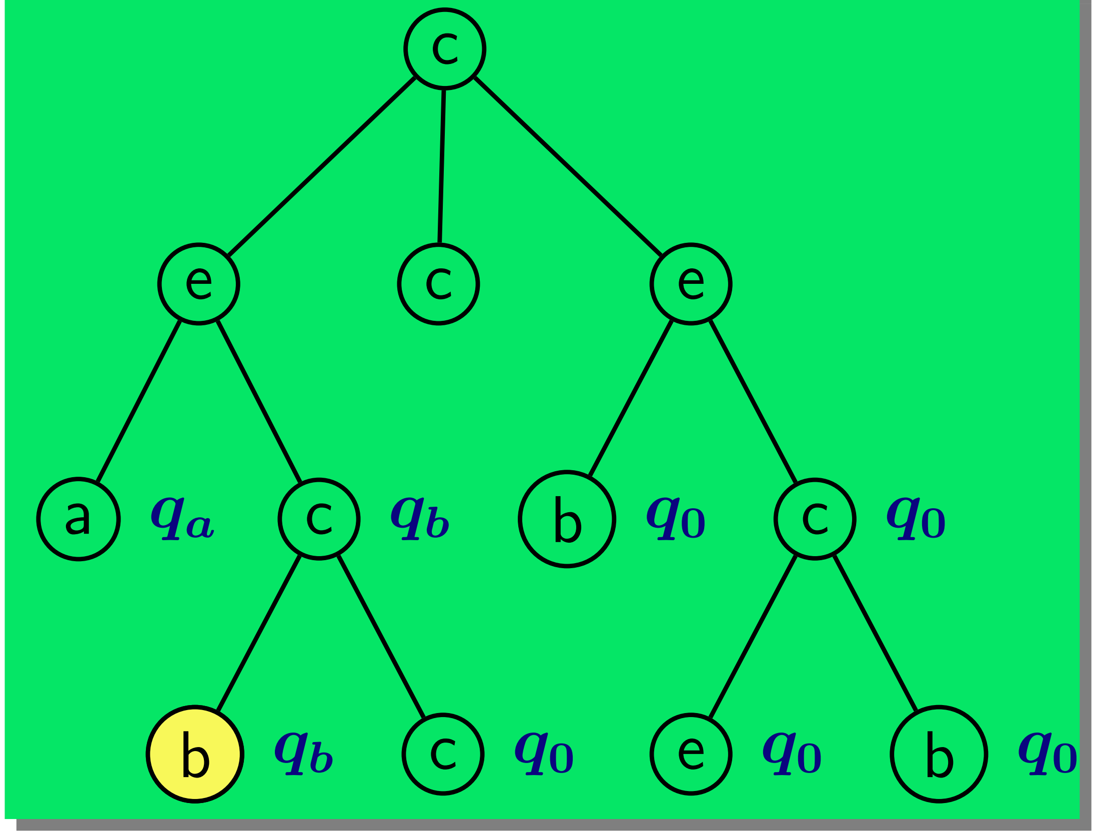
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$ $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

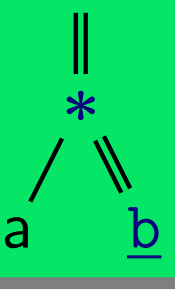
Example tree: Run 1



Example query

//∗[a]//b

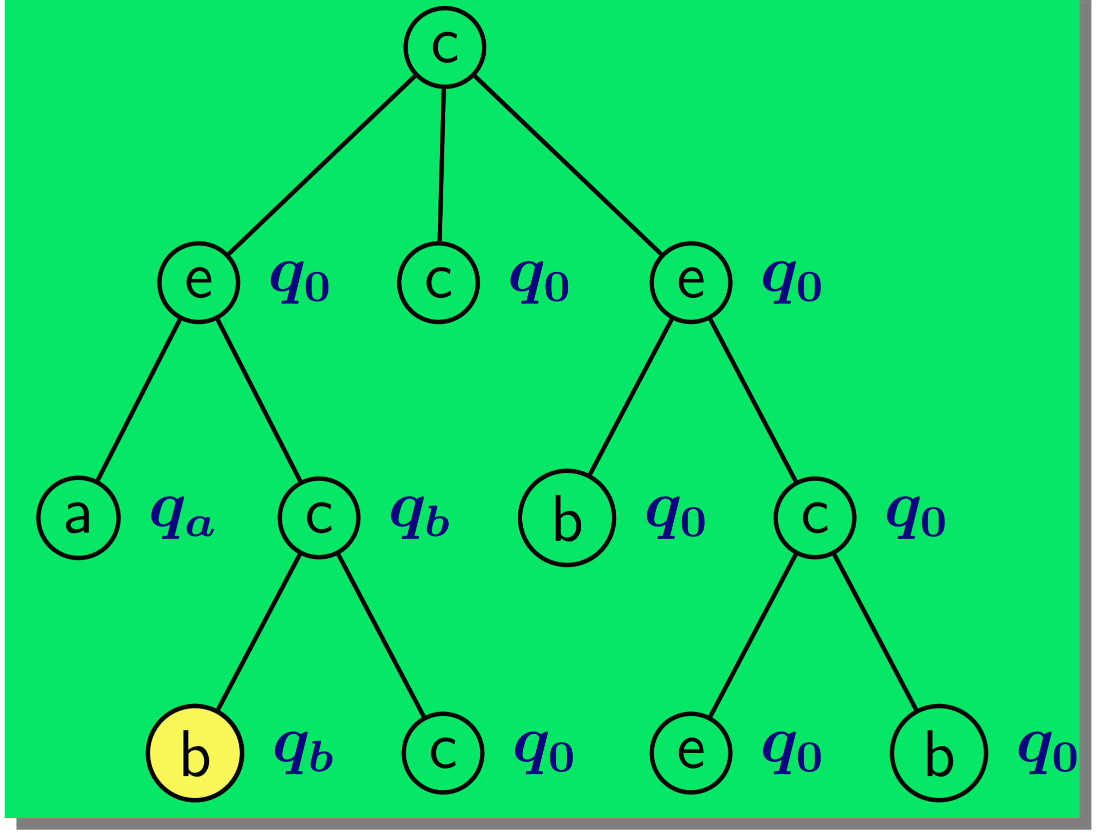
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$
  $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

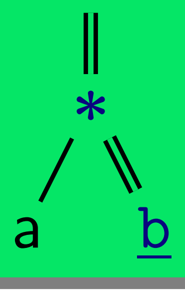
Example tree: Run 1



Example query

//*[a]//b

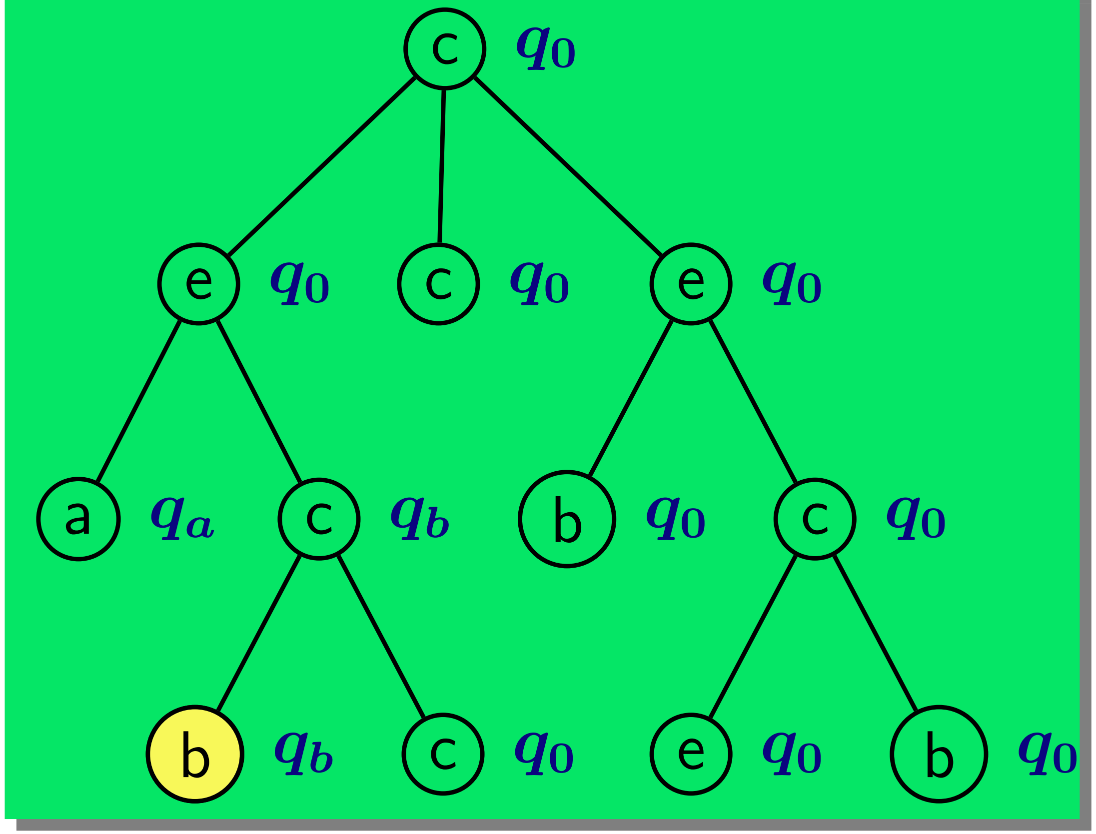
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$
  $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

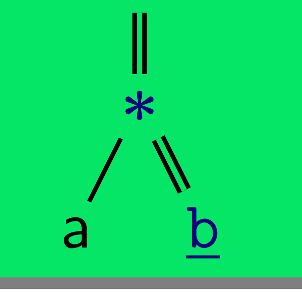
Example tree: Run 1



Example query

//*[a]//b

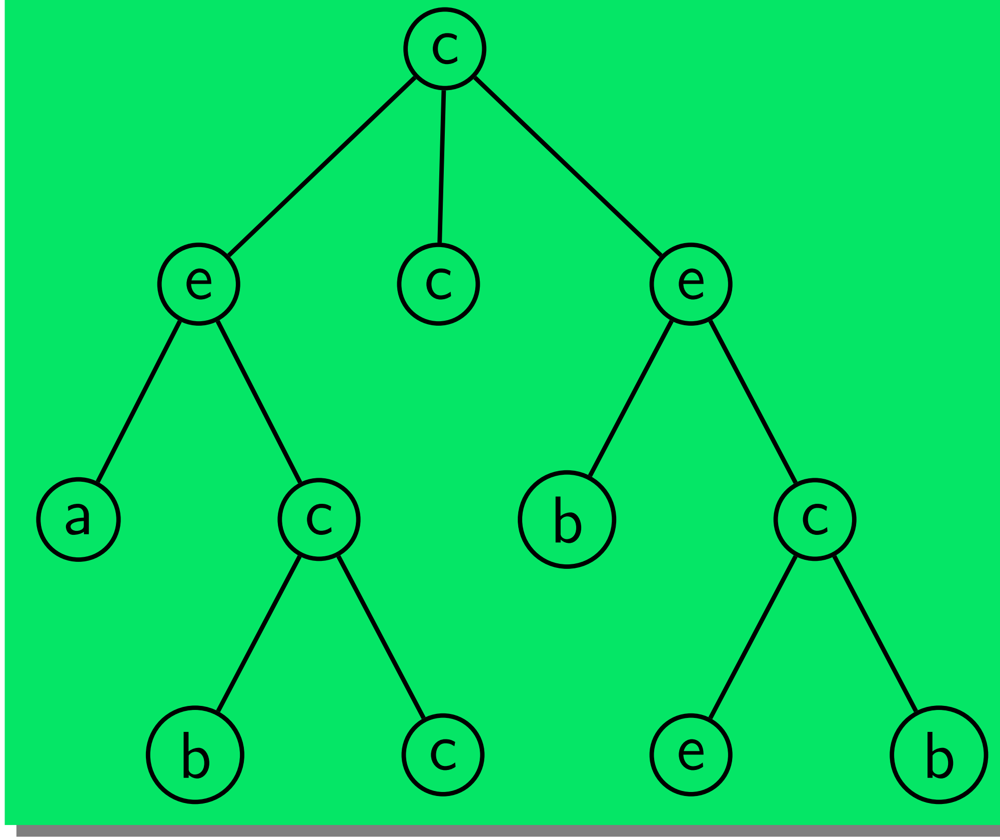
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$ $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

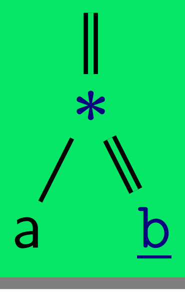
Example tree: Run 2



Example query

//∗[a]//b

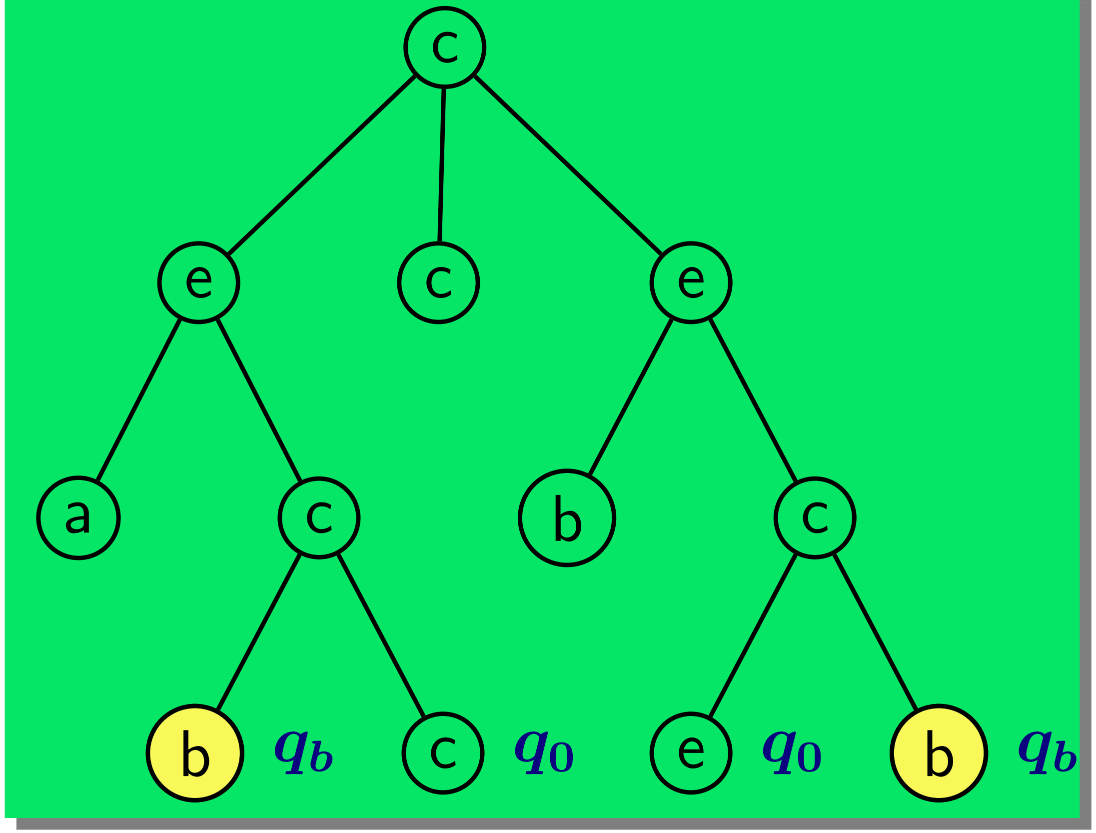
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$
  $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

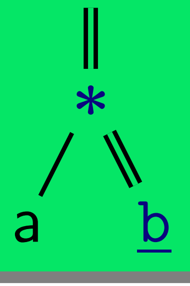
Example tree: Run 2



Example query

//*[a]//b

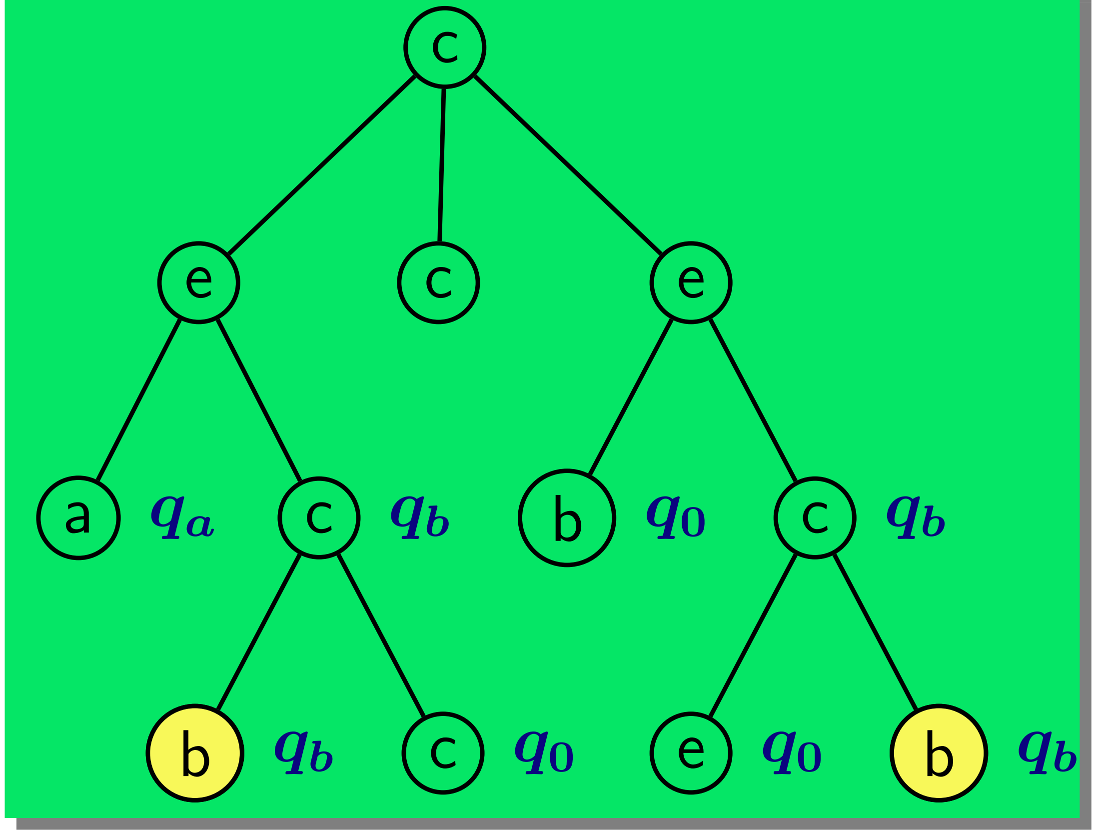
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$
  $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

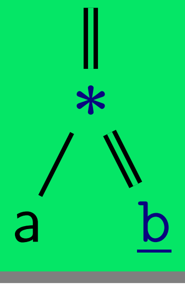
Example tree: Run 2



Example query

//$*$[a]//b

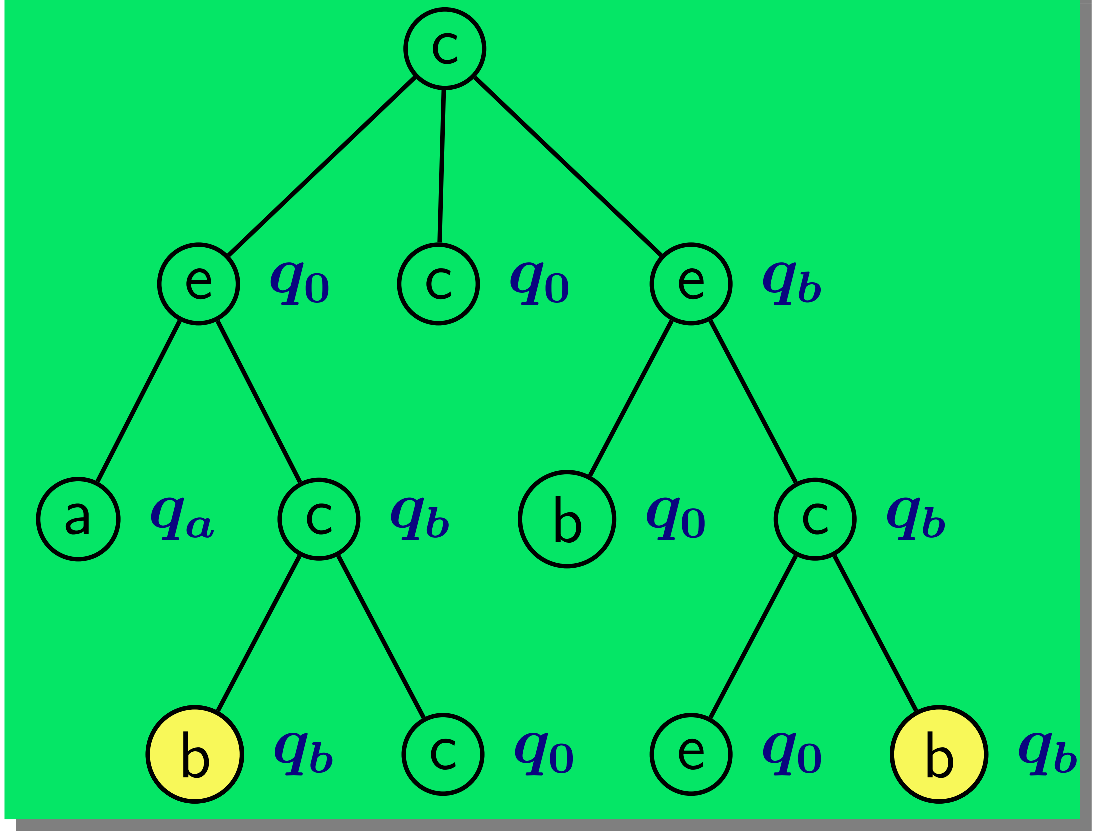
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$
  $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

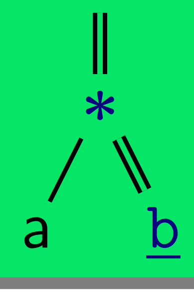
Example tree: Run 2



Example query

$//*[a]//b$

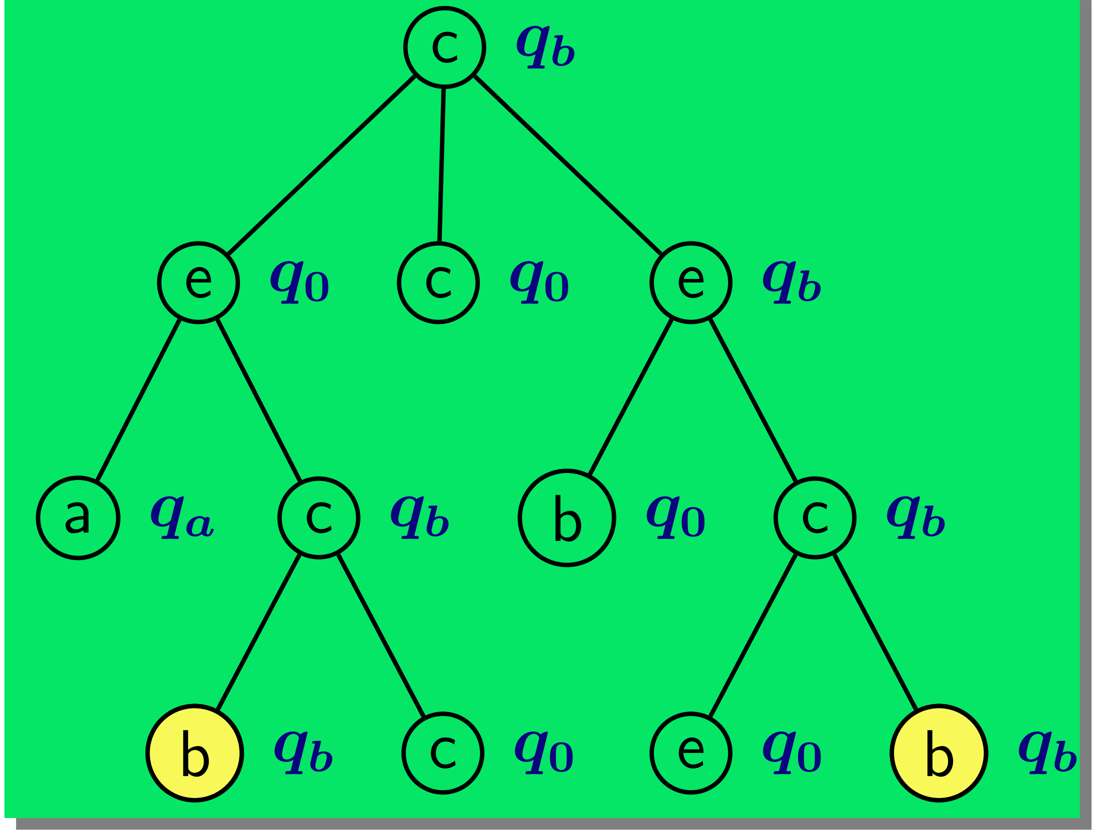
Query tree



Example automaton

- $Q = \{q_0, q_a, q_b\}$
- $L(q_a, a) = Q^*$
- $L(q_b, \sigma) = Q^*$
- $L(q_0, \sigma) =$ $\epsilon + q_0^* + Q^* q_a Q^*$
- all other sets empty
- $s(q_b, b) = 1$
- all others: 0
- Accepting: q_0

Example tree: Run 2



Fact

- Existential semantics: a node is in the result if there exists an accepting run which selects it
- Universal semantics: a node is in the result if every accepting run selects it
- Both semantics define the same class of queries

Result

A node selecting query is MSO-definable iff it is expressible by a nondeterministic bottom-up node selecting automaton

Node-Selecting Automata (cont.)

Result

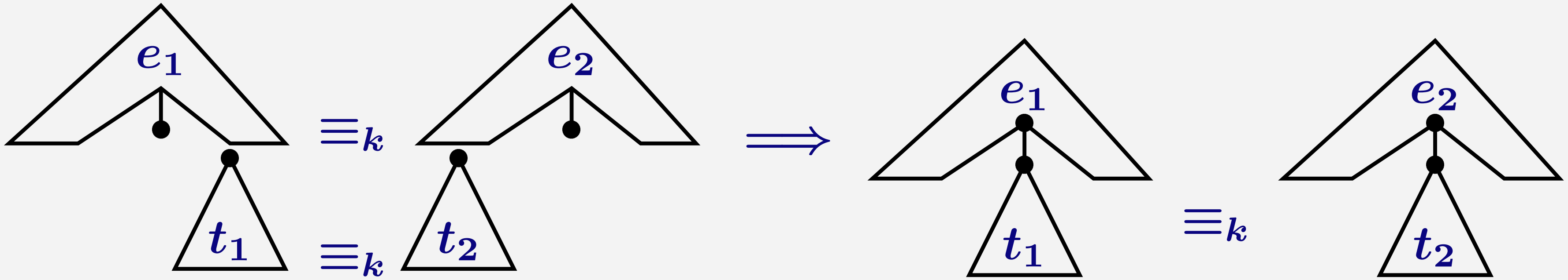
A node selecting query is MSO-definable iff it is expressible by a nondeterministic bottom-up node selecting automaton

Proof Idea

- Given formula $\varphi(x)$ of quantifier-depth k and tree t, for each node v the automaton does the following:
  - Compute k-type of subtree at v
  - Guess k-type of "envelope tree" at v
  - Conclude whether v is in the output
  - Check consistency upwards towards the root

$\Rightarrow$ one unique accepting run

Crucial fact



More query models

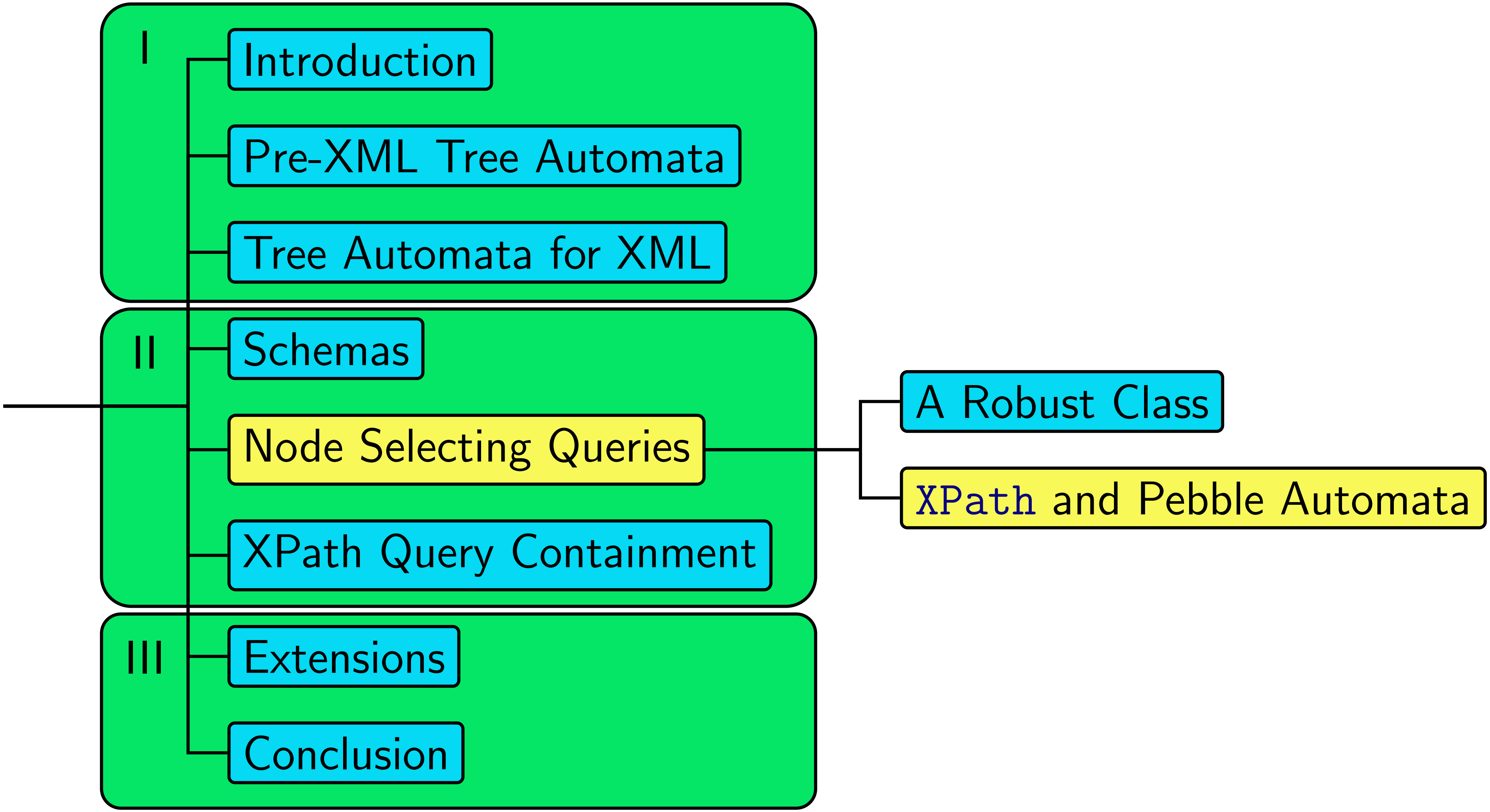
- Unfortunately, the translation from formula to automaton can be prohibitively expensive: number of states $\sim \left.\mathbf{2}^{\mathbf{2}^{\cdot^{\cdot^{\cdot^{\mathbf{2}^{\mathbf{2}^{|\varphi|}}}}}}}\right\}|\varphi|$
- Actually: If **P** $\neq$ **NP** there is no elementary f, such that MSO-formulas can be evaluated in time $f(|\text{formula}| \times p(|\text{tree}|))$ with polynomial p [Frick, Grohe 2002]

⟶ query languages with better complexity properties needed

- Good candidate: Monadic Datalog [Gottlob, Koch 2002] and its restricted dialects like TMNF
- Further models:
  - Attributed Grammars [Neven, Van den Bussche 1998]
  - μ-formulas [Neumann 1998]
  - Context Grammars [Neumann 1999]
  - Deterministic Node-Selecting Automata [Neven, Sch. 1999]

Some facts about query evaluation

- MSO node-selecting queries can be evaluated in two passes through the tree
  - first pass, bottom-up: essentially computes the types of the subtrees
  - second pass, top-down: essentially computes the types of the envelopes and combines it with the subtree information
- This can be implemented by a 2-pass pushdown document automaton which in its first pass attaches information to each node [Neumann, Seidl 1998; Koch 2003]
- In particular: queries can be evaluated in linear time



I

Introduction

Pre-XML Tree Automata

Tree Automata for XML

II

Schemas

Node Selecting Queries

XPath Query Containment

III

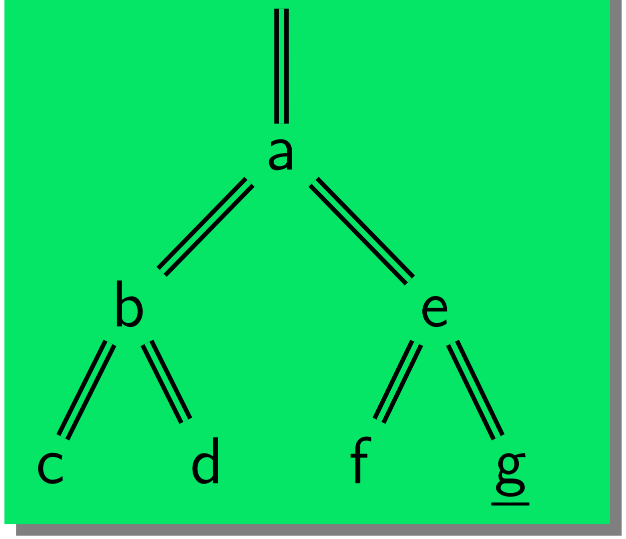
Extensions

Conclusion

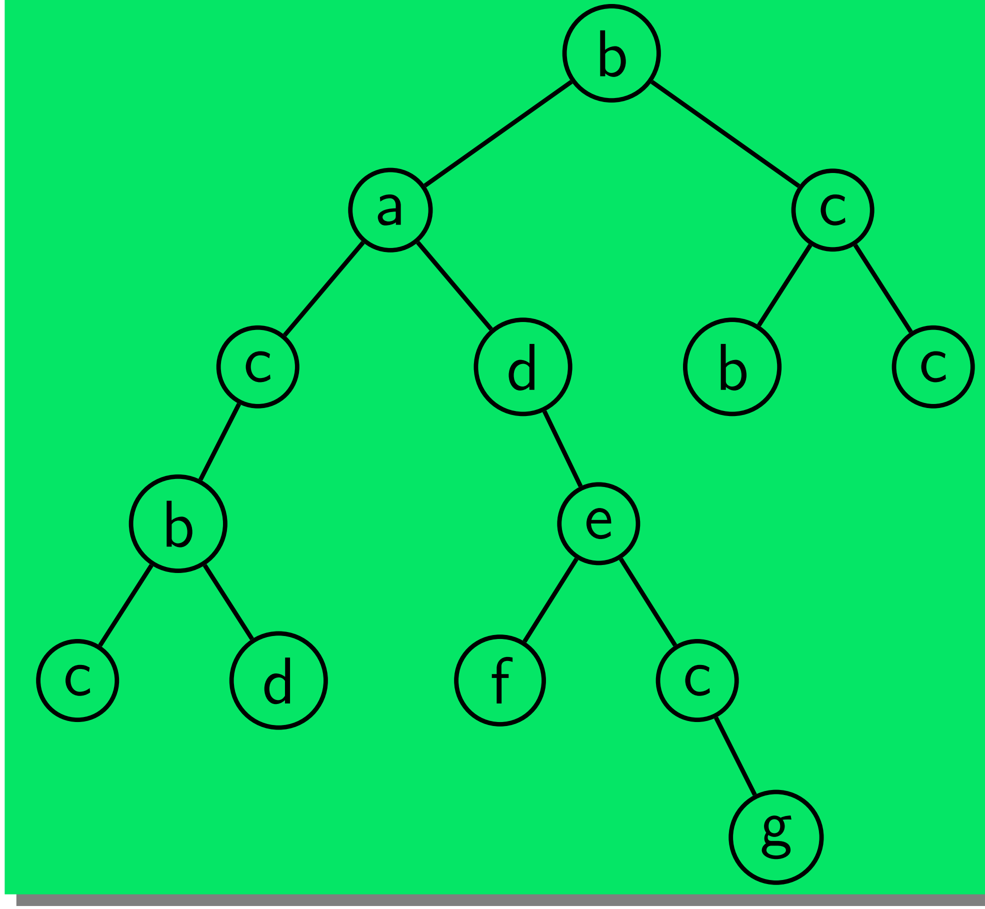
A Robust Class

`XPath` and Pebble Automata

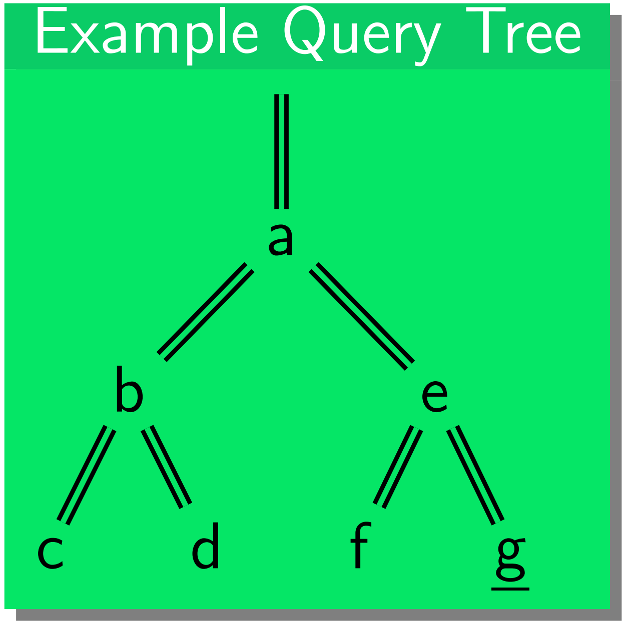
Example Query Tree

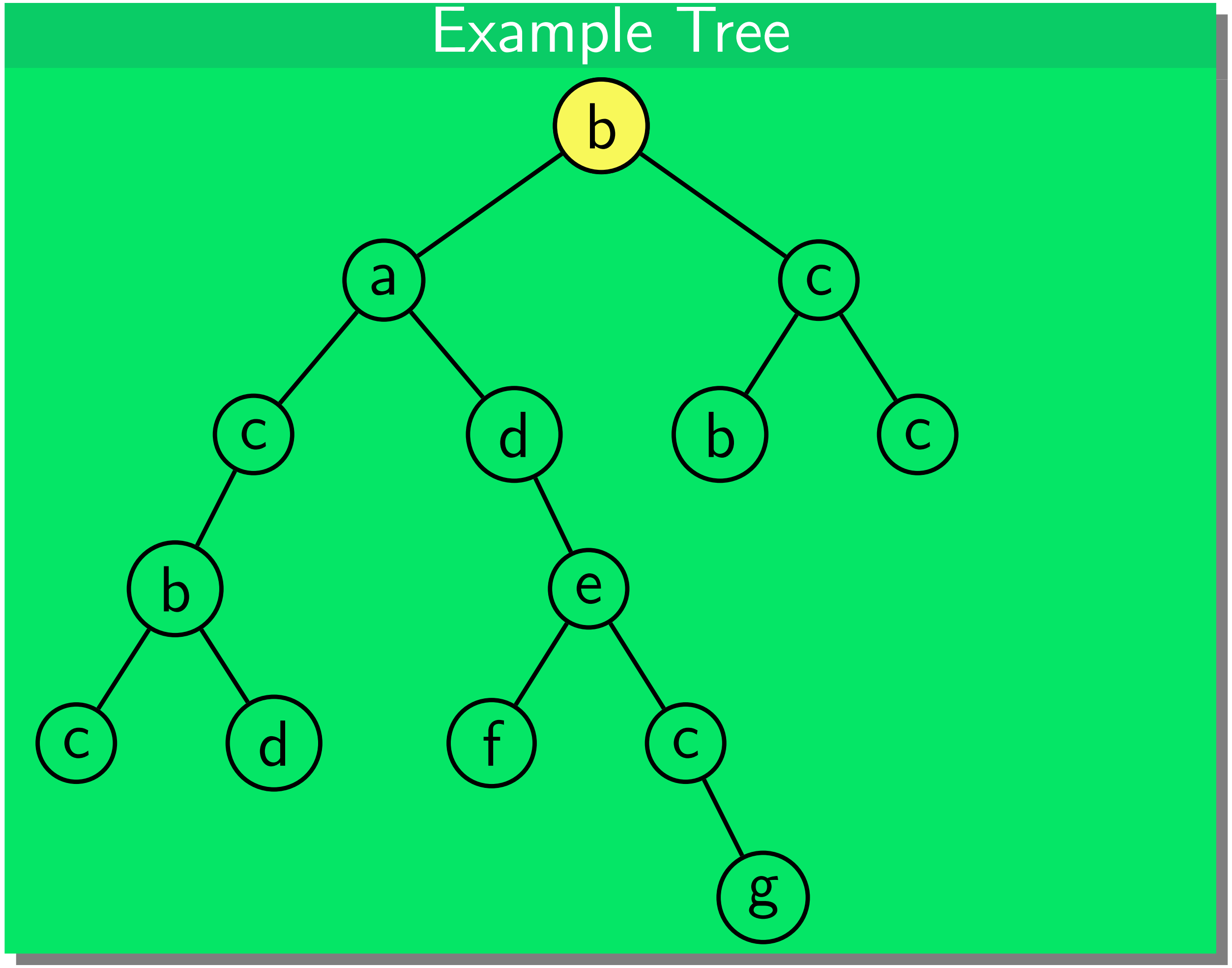


Example Tree

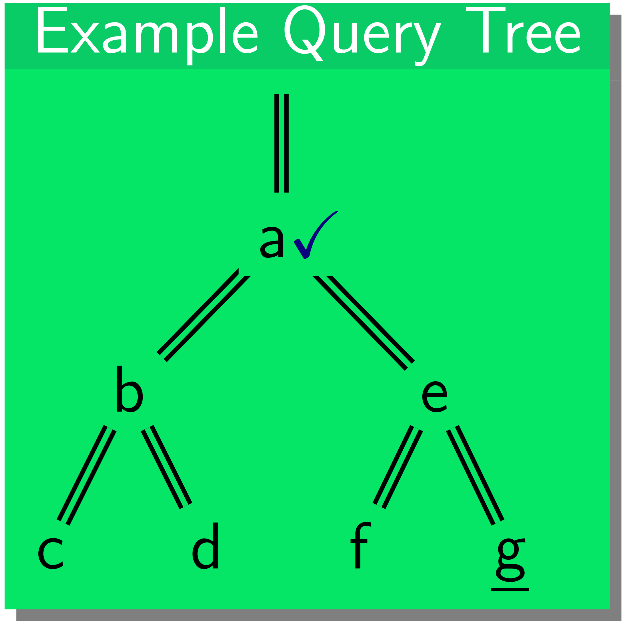


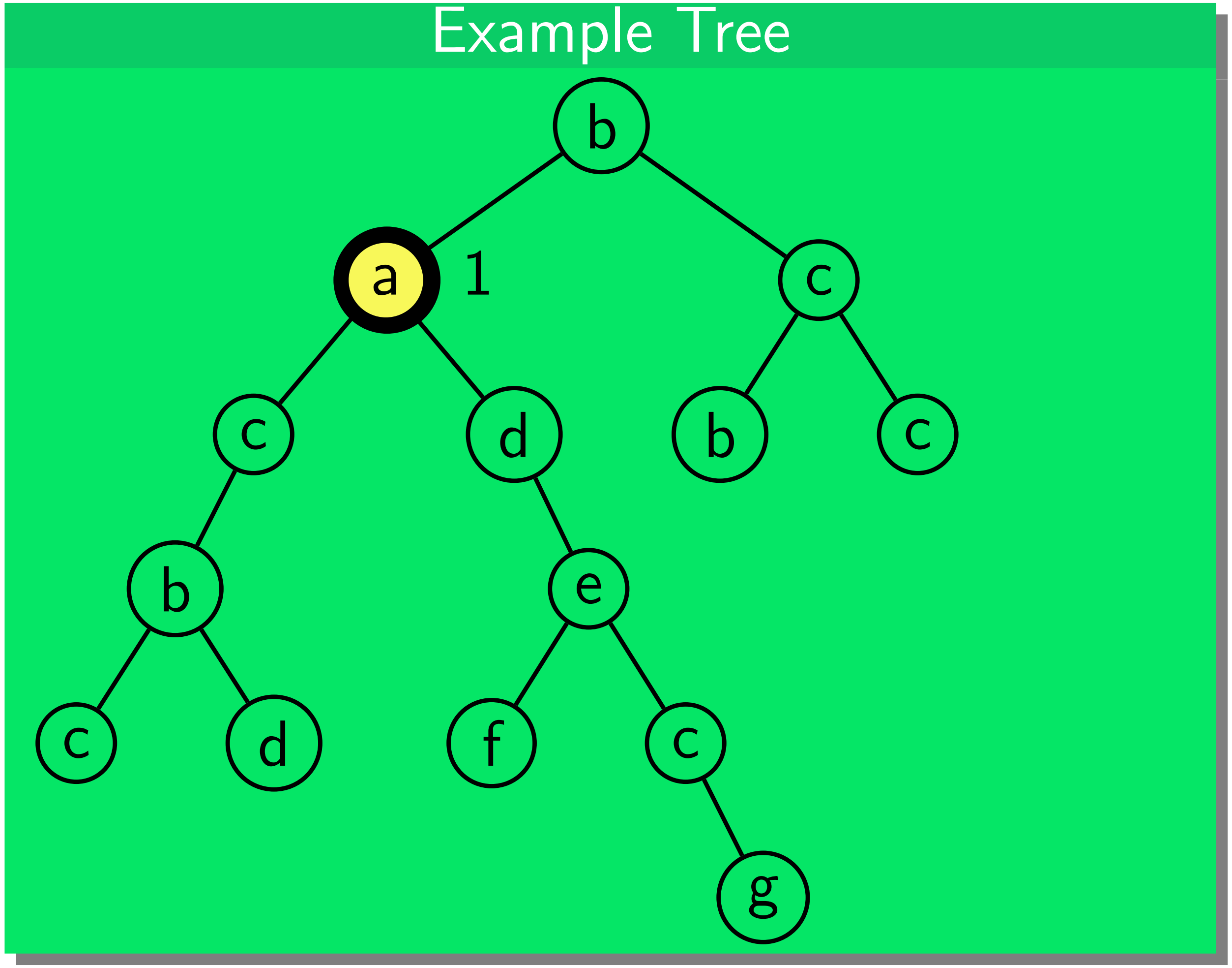
Pebble Automata: Example


Example Query Tree

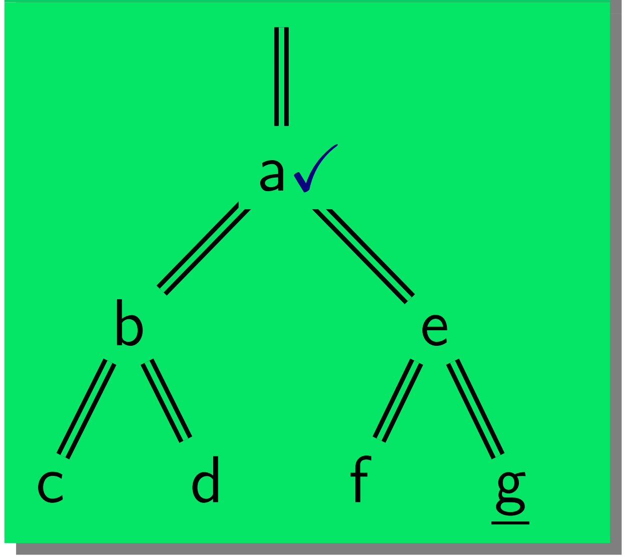

Example Tree

Pebble Automata: Example

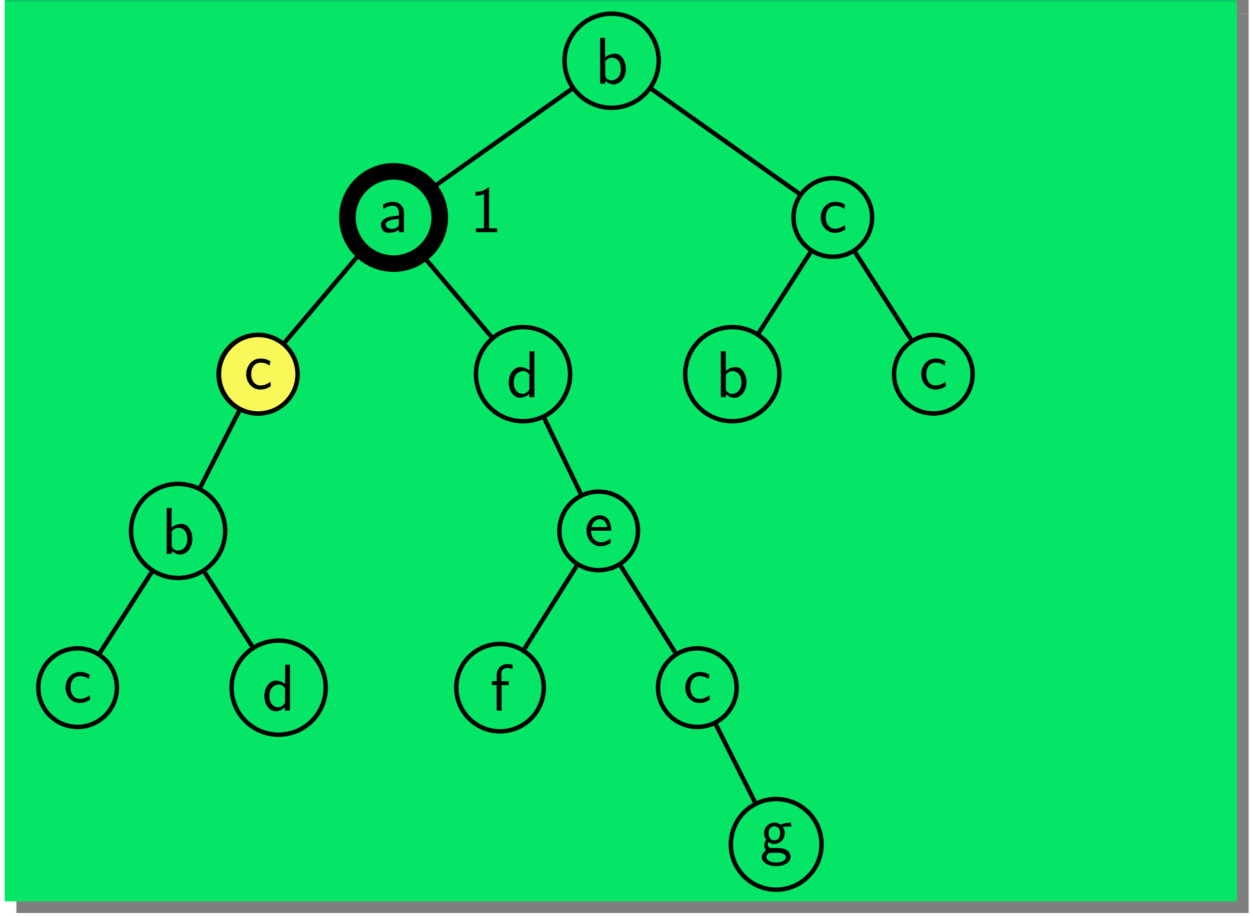




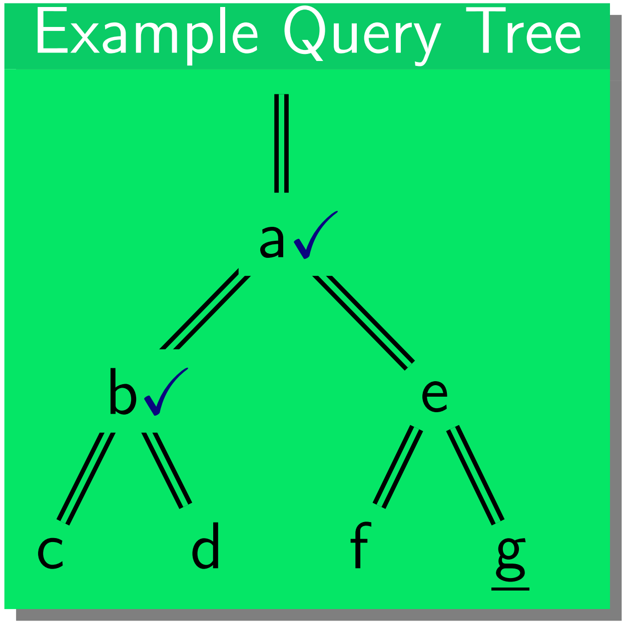
Example Query Tree

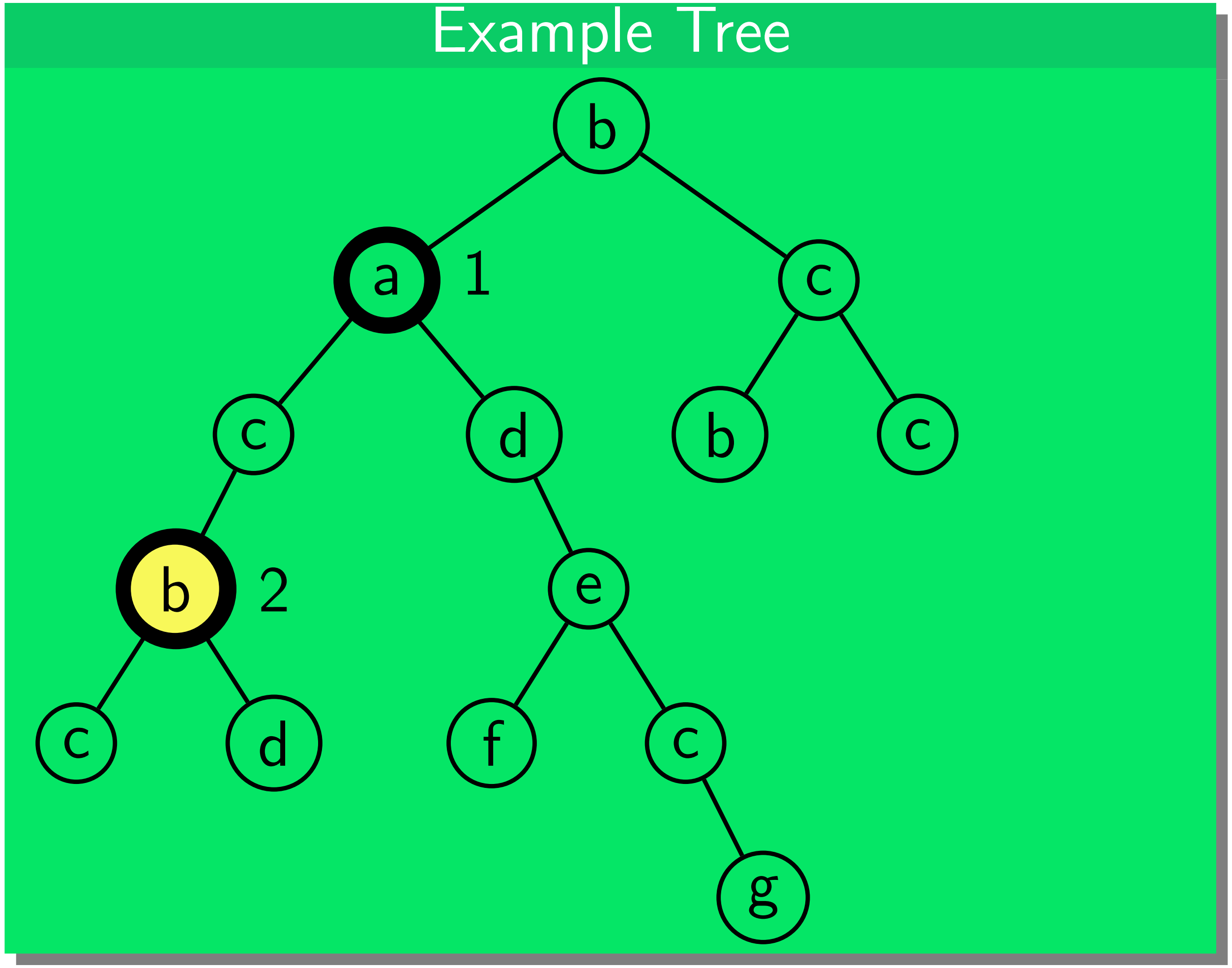


Example Tree



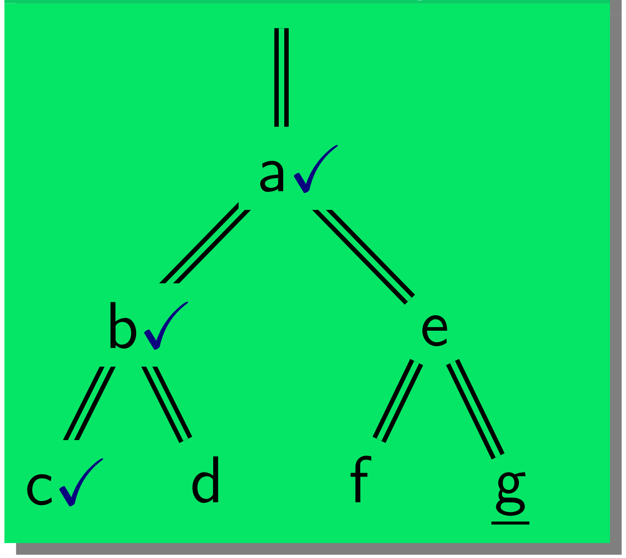
Pebble Automata: Example


Example Query Tree

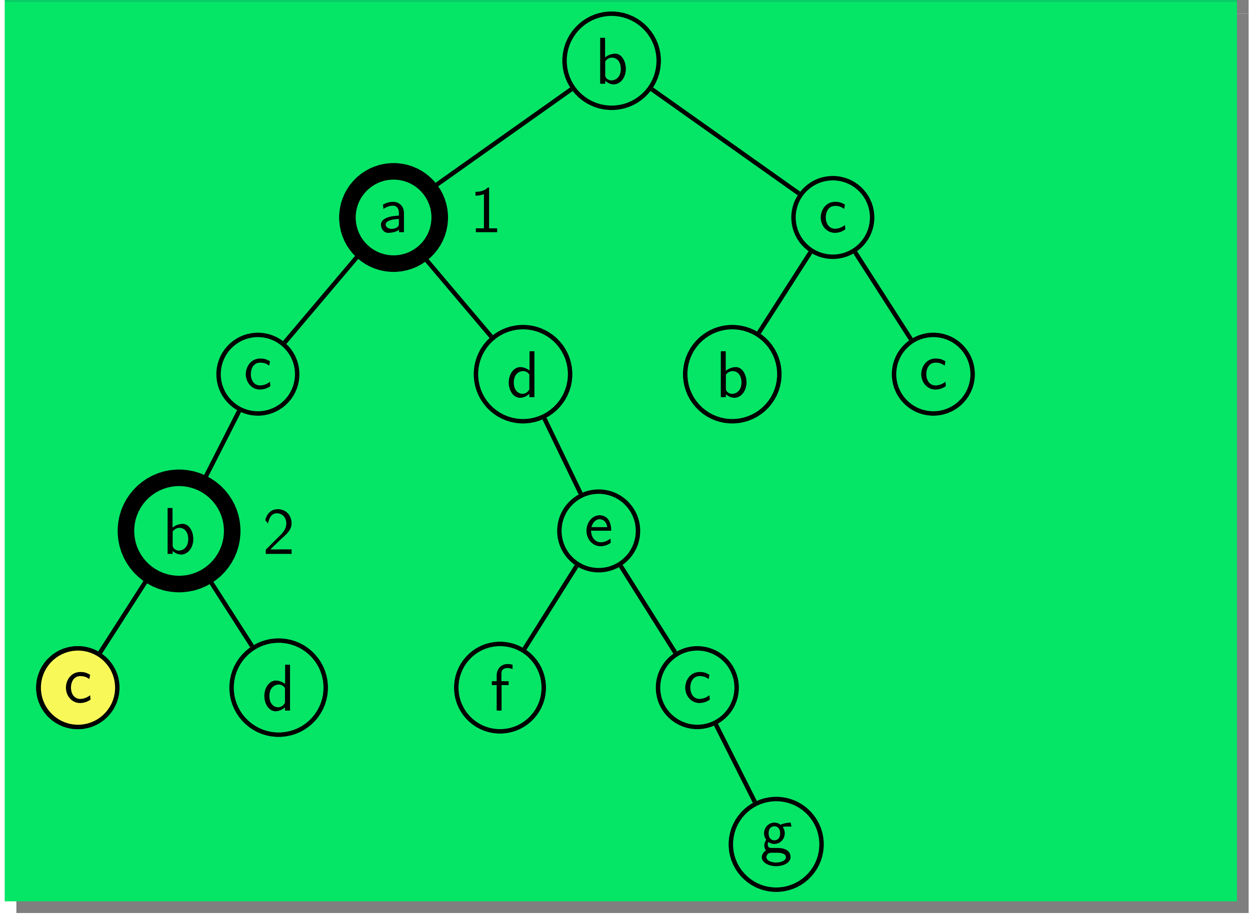

Example Tree

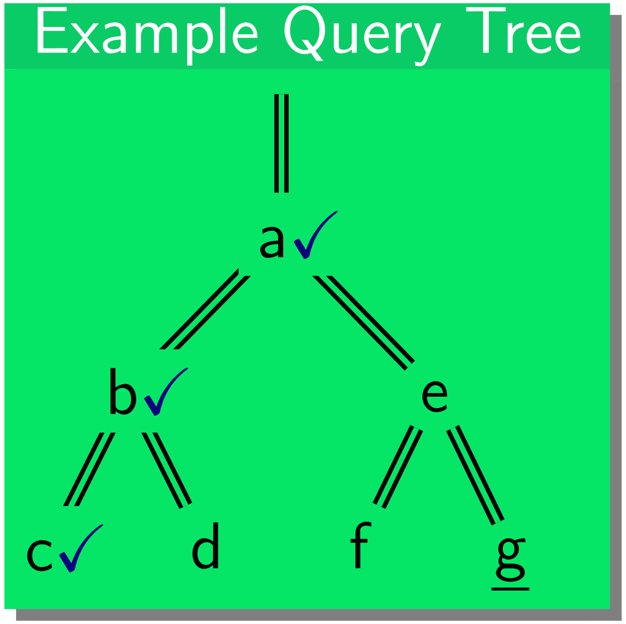
Node-Selecting Q. — XPath — **Pebble Automata: Example**

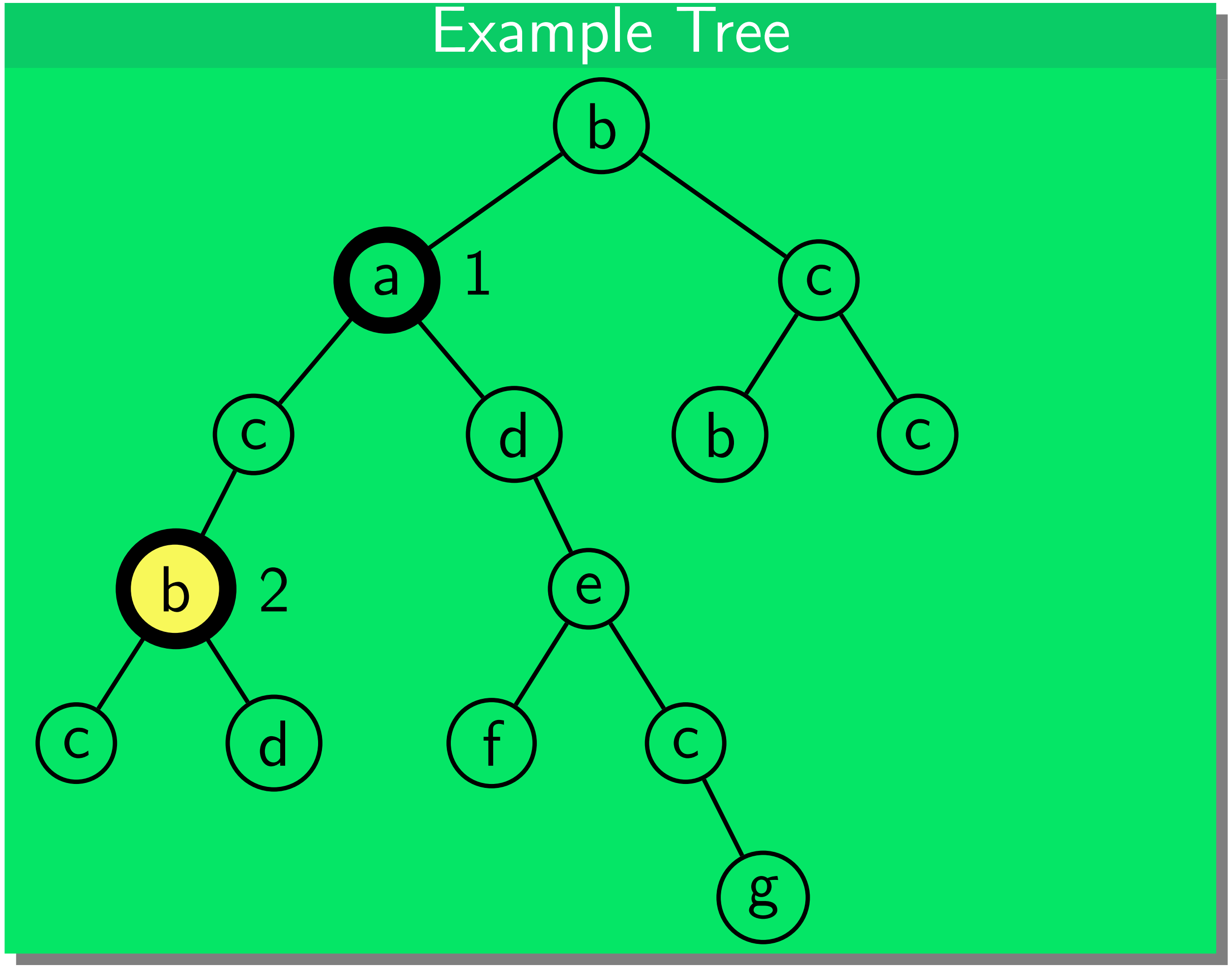
Example Query Tree



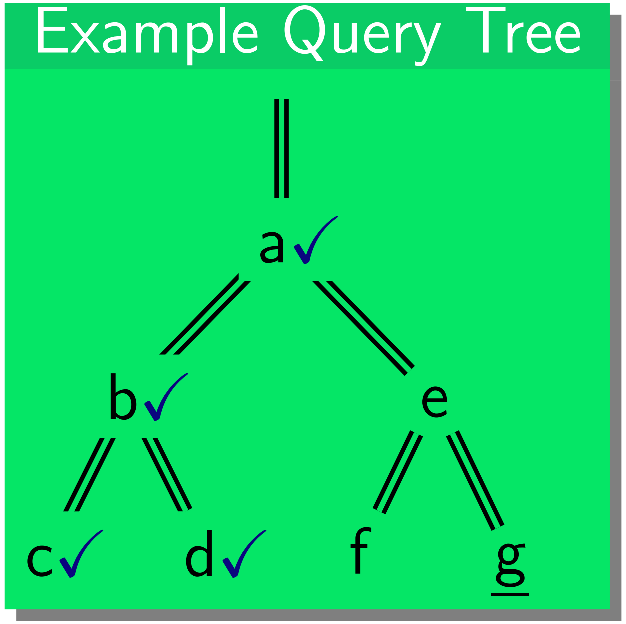
Example Tree

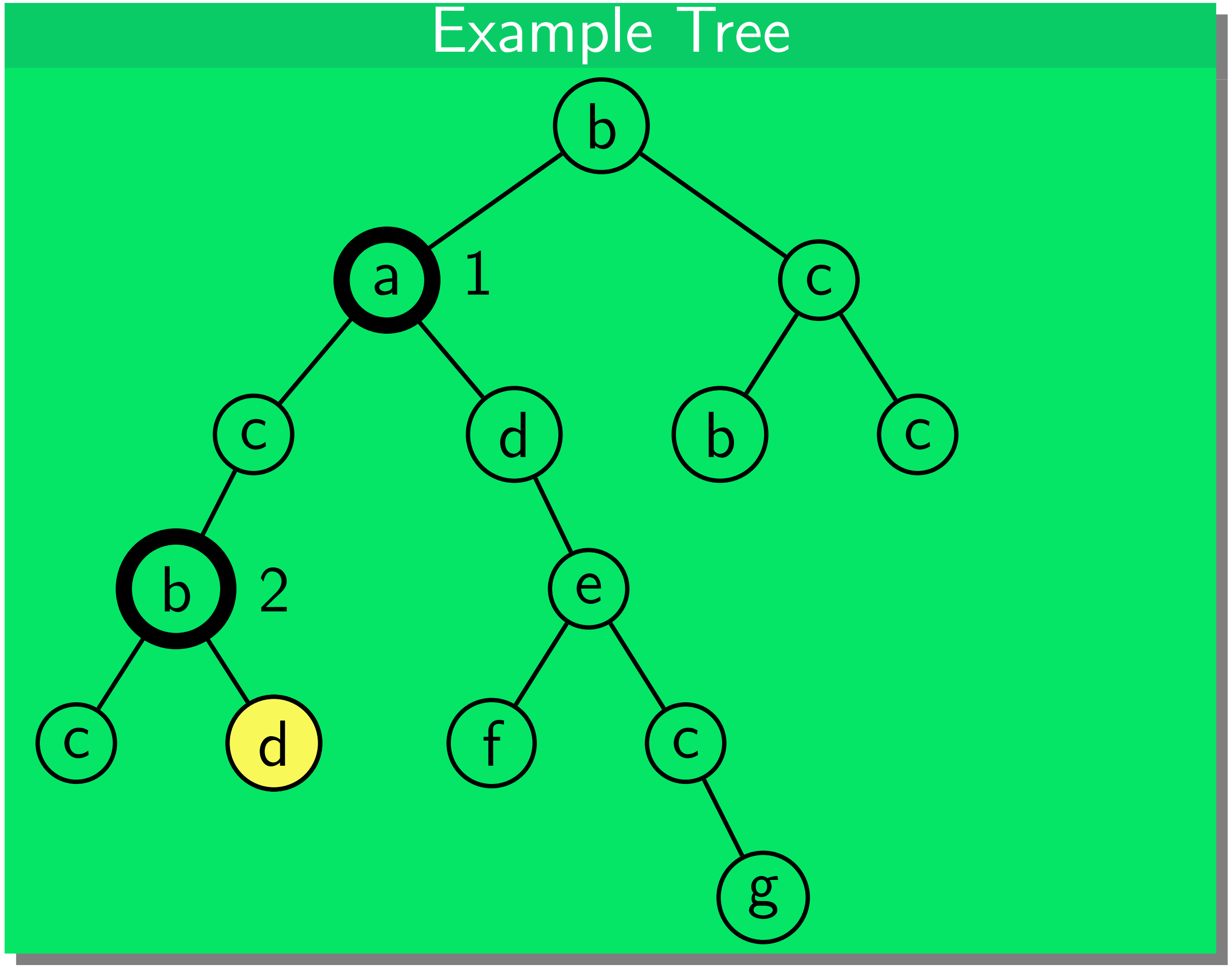



Example Query Tree
a✓
b✓
e
c✓
d
f
g

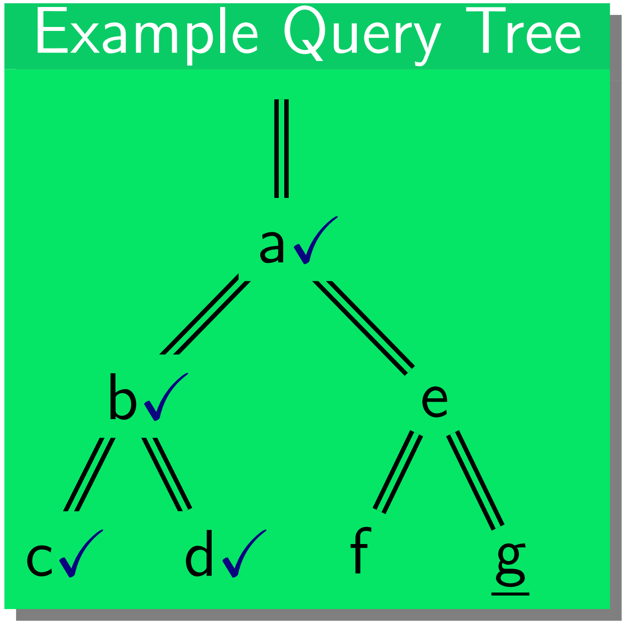

Example Tree
b
a
1
c
c
d
b
c
b
2
e
c
d
f
c
g

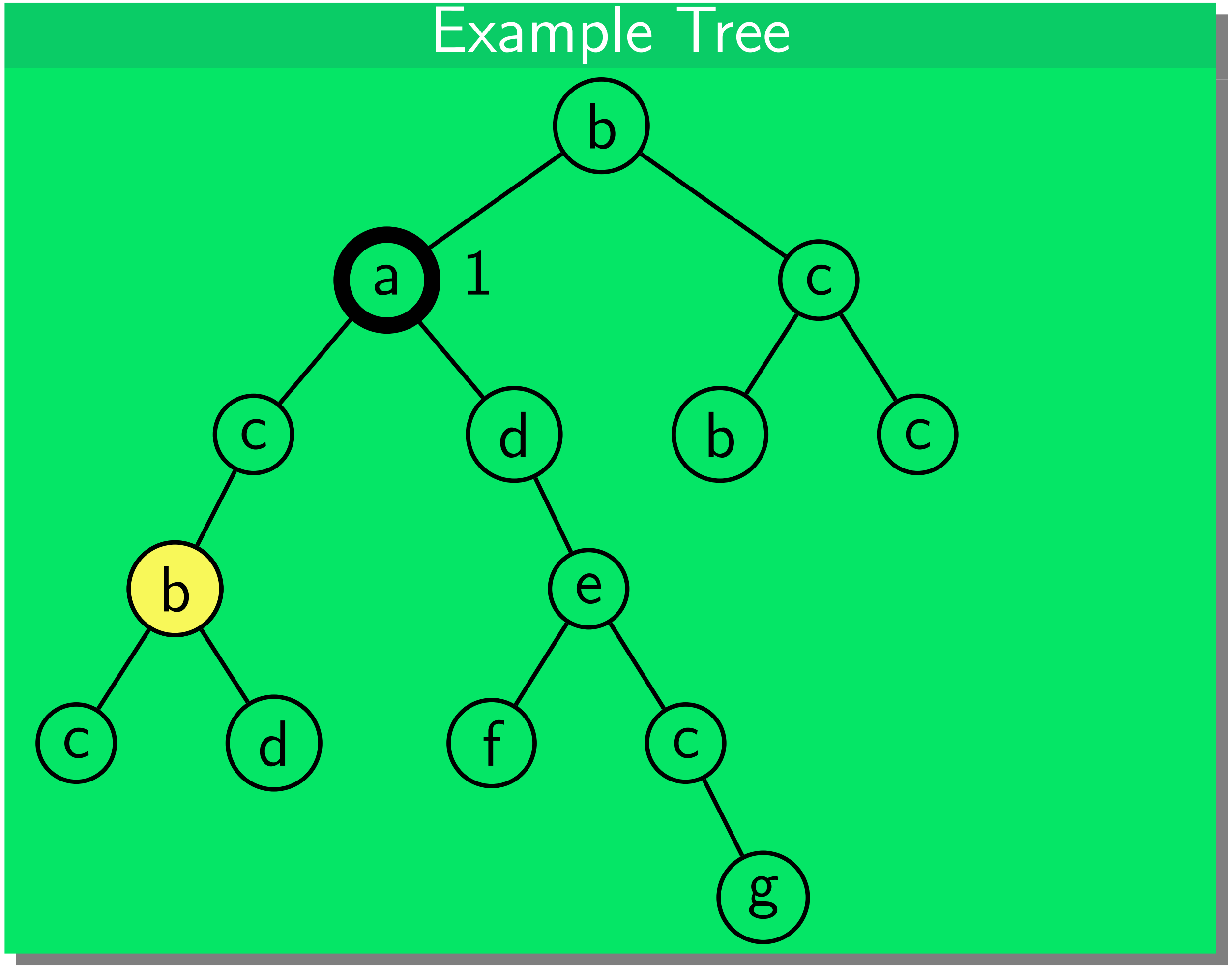
Pebble Automata: Example


Example Query Tree


Example Tree

Pebble Automata: Example

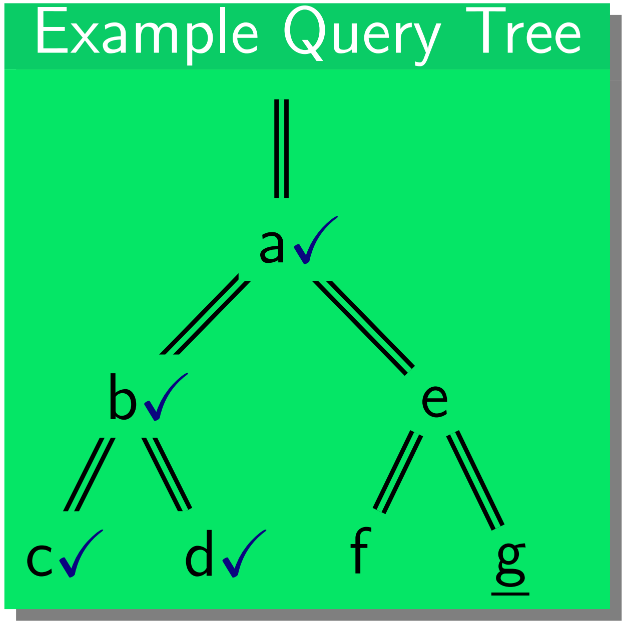

Example Query Tree


Example Tree

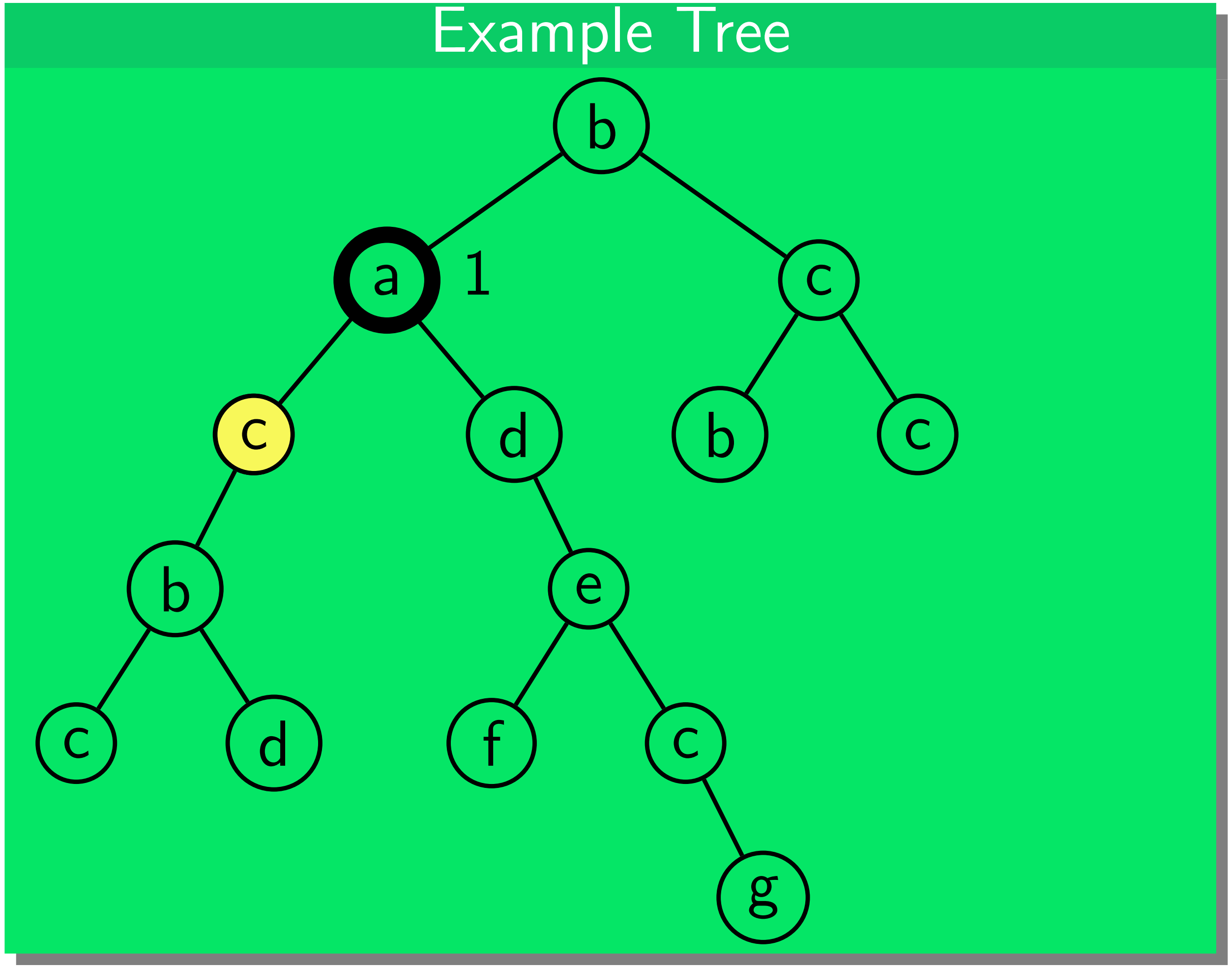
Node-Selecting Q. — XPath —

Pebble Automata: Example

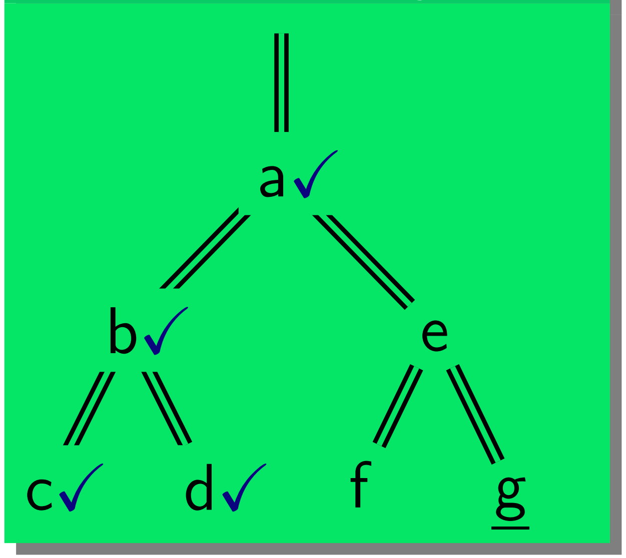
Example Query Tree



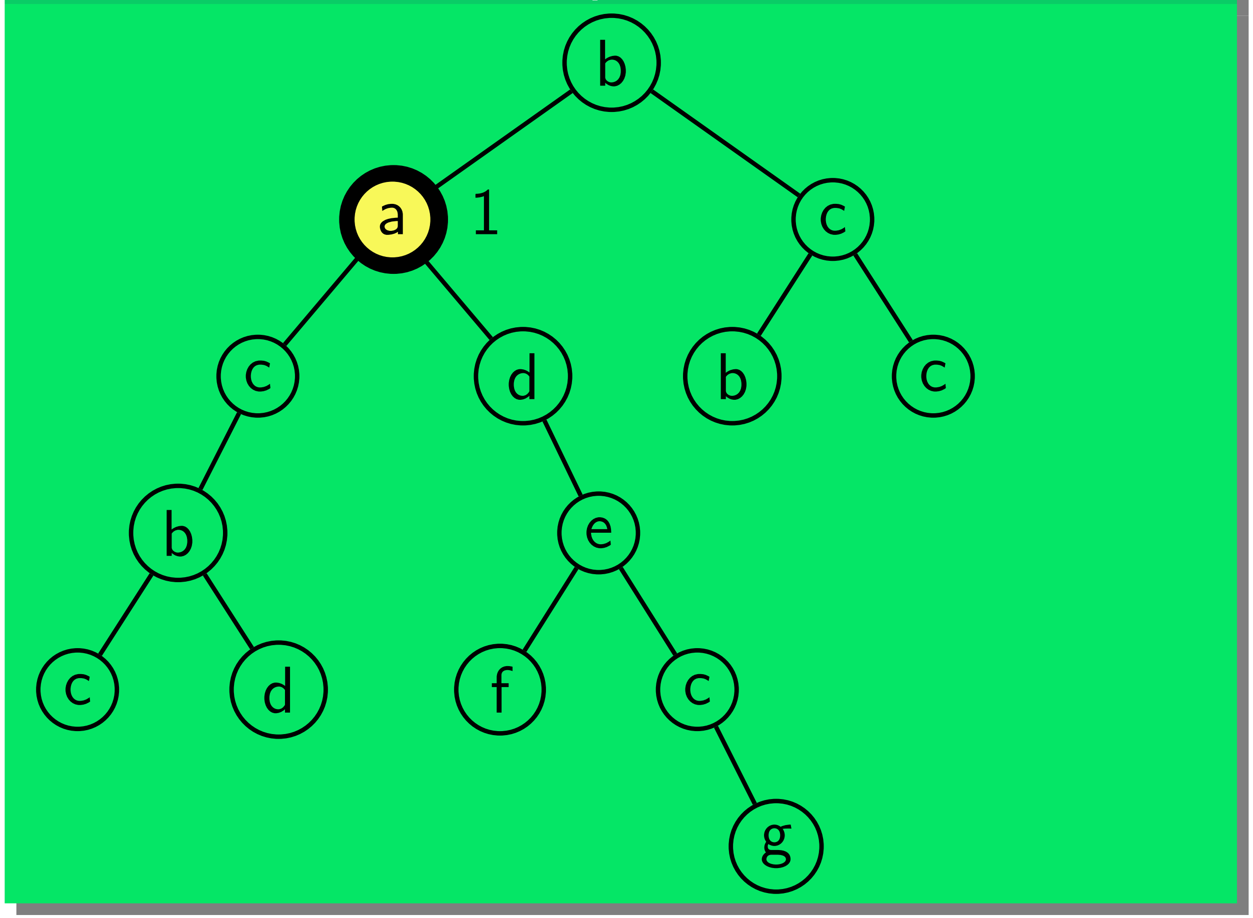
Example Tree



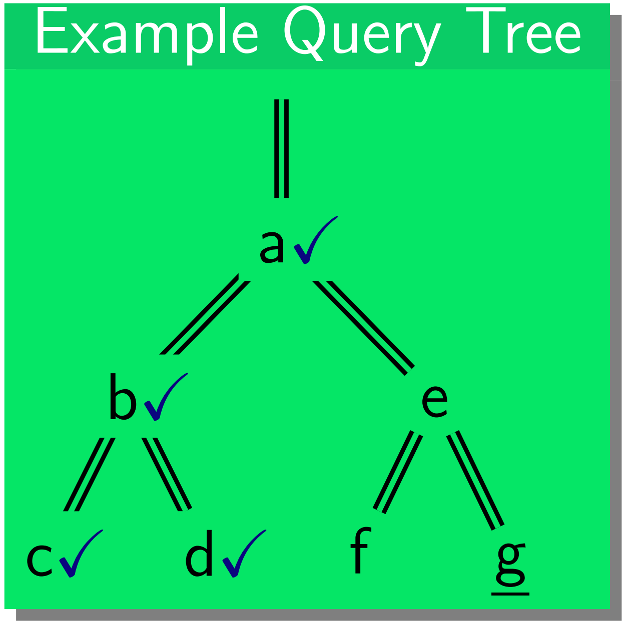
Example Query Tree

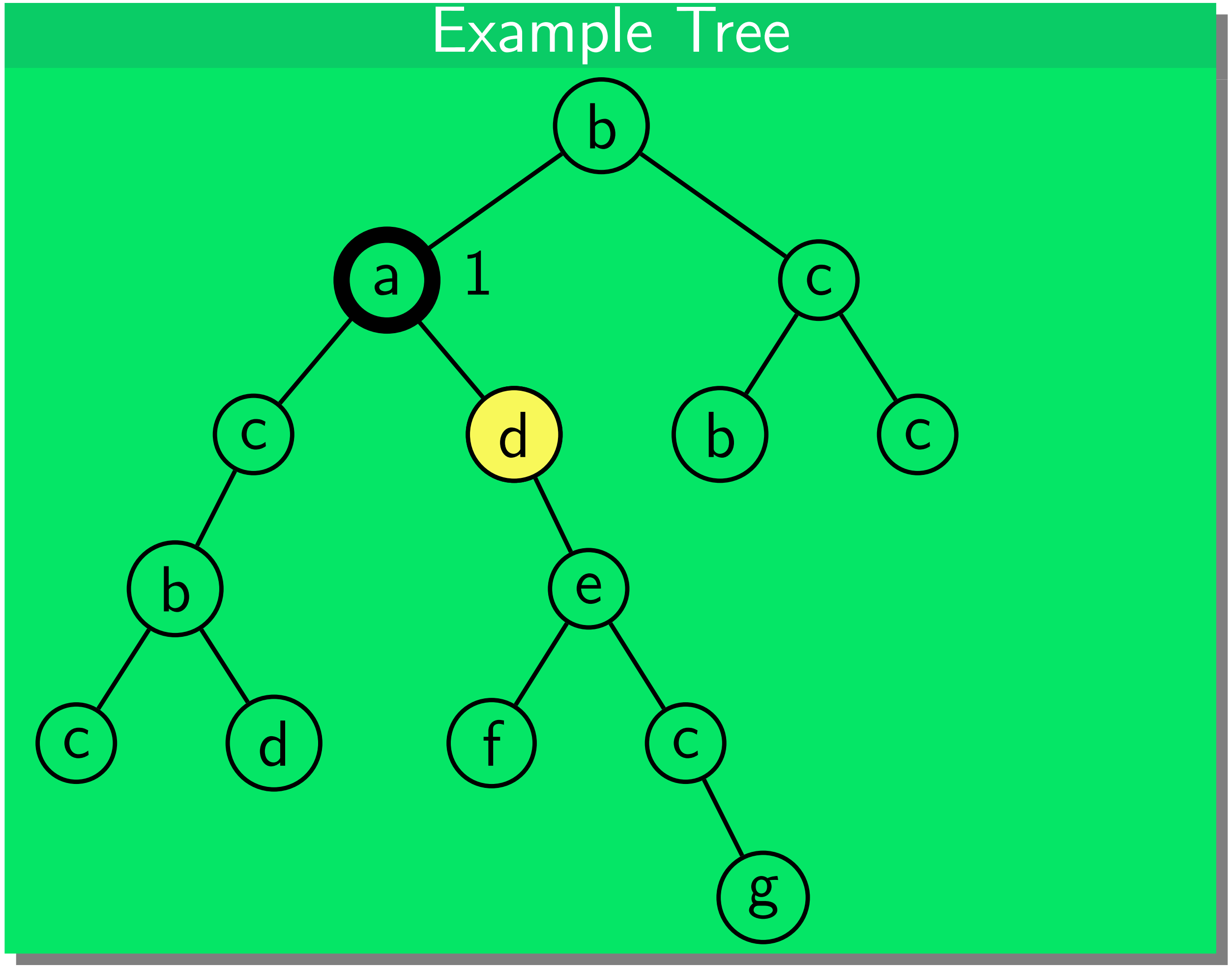


Example Tree

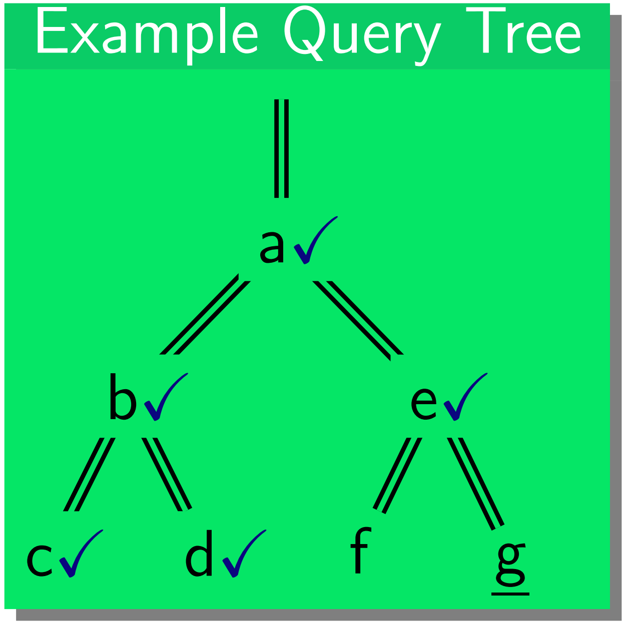


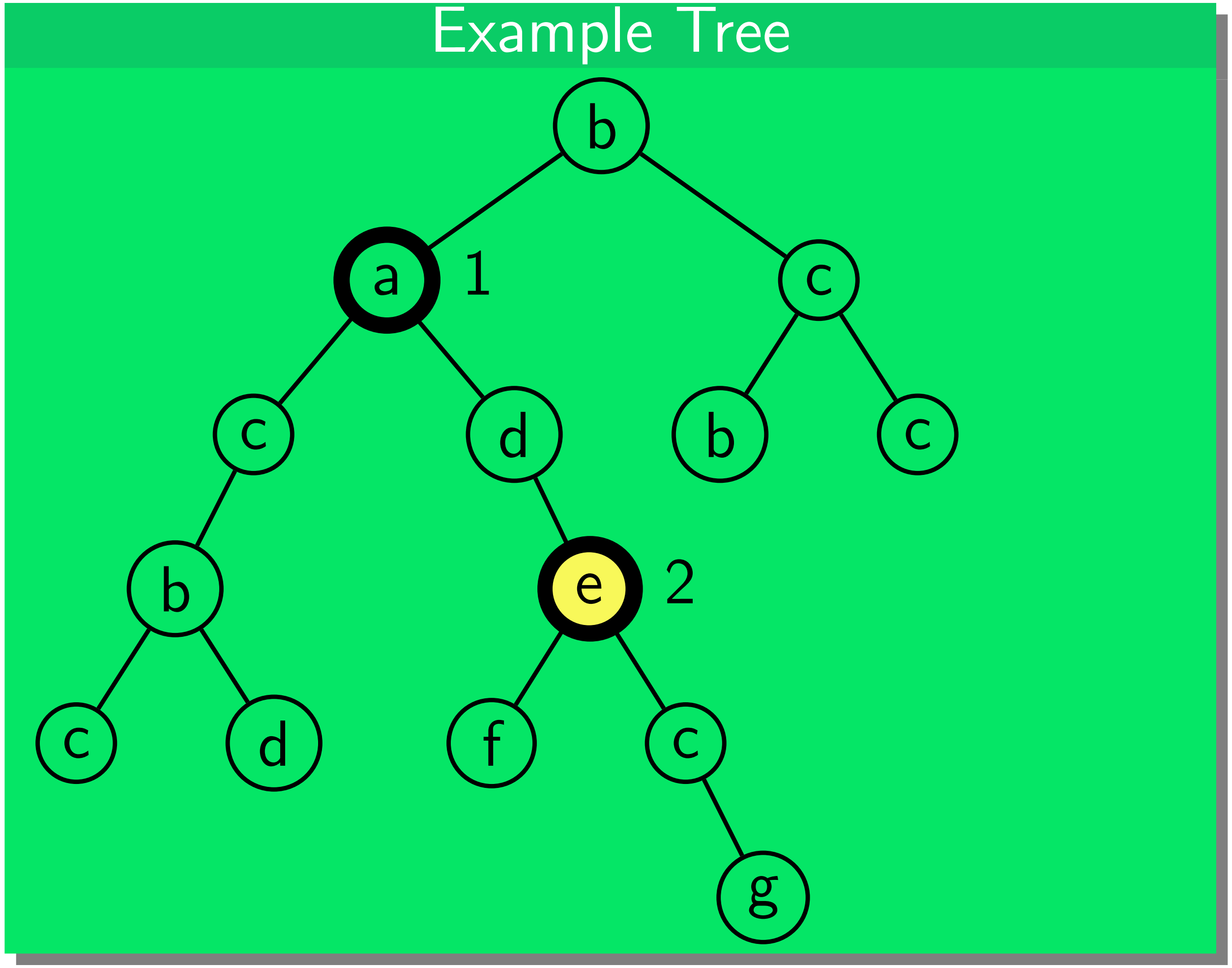
Pebble Automata: Example


Example Query Tree

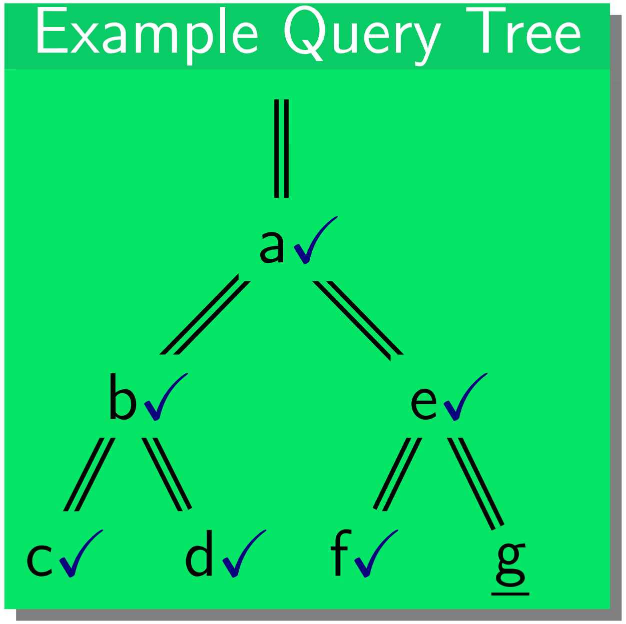

Example Tree

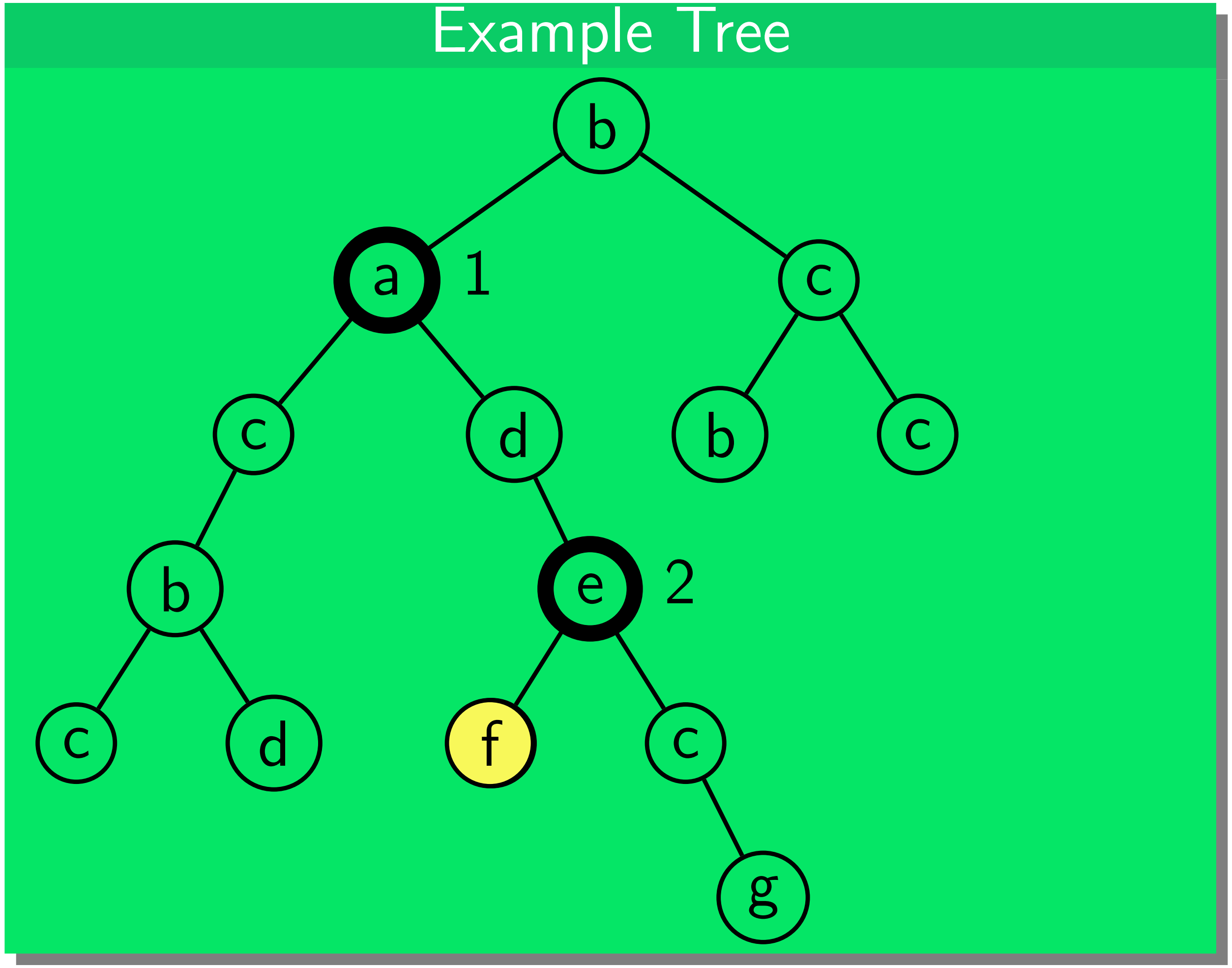
Node-Selecting Q. — XPath — **Pebble Automata: Example**


Example Query Tree


Example Tree

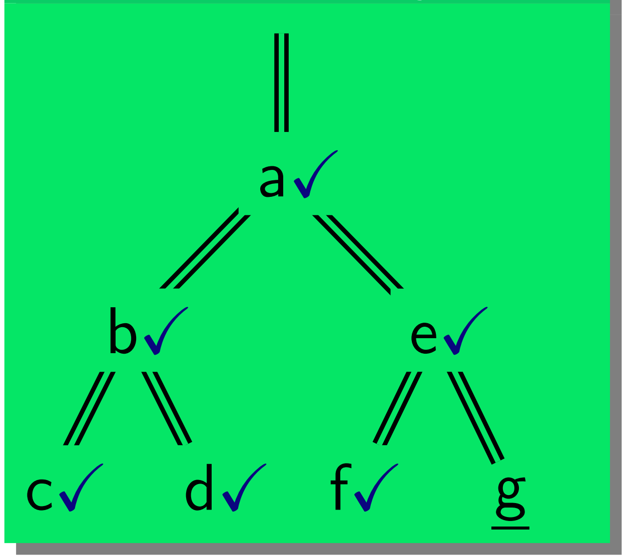
Pebble Automata: Example


Example Query Tree

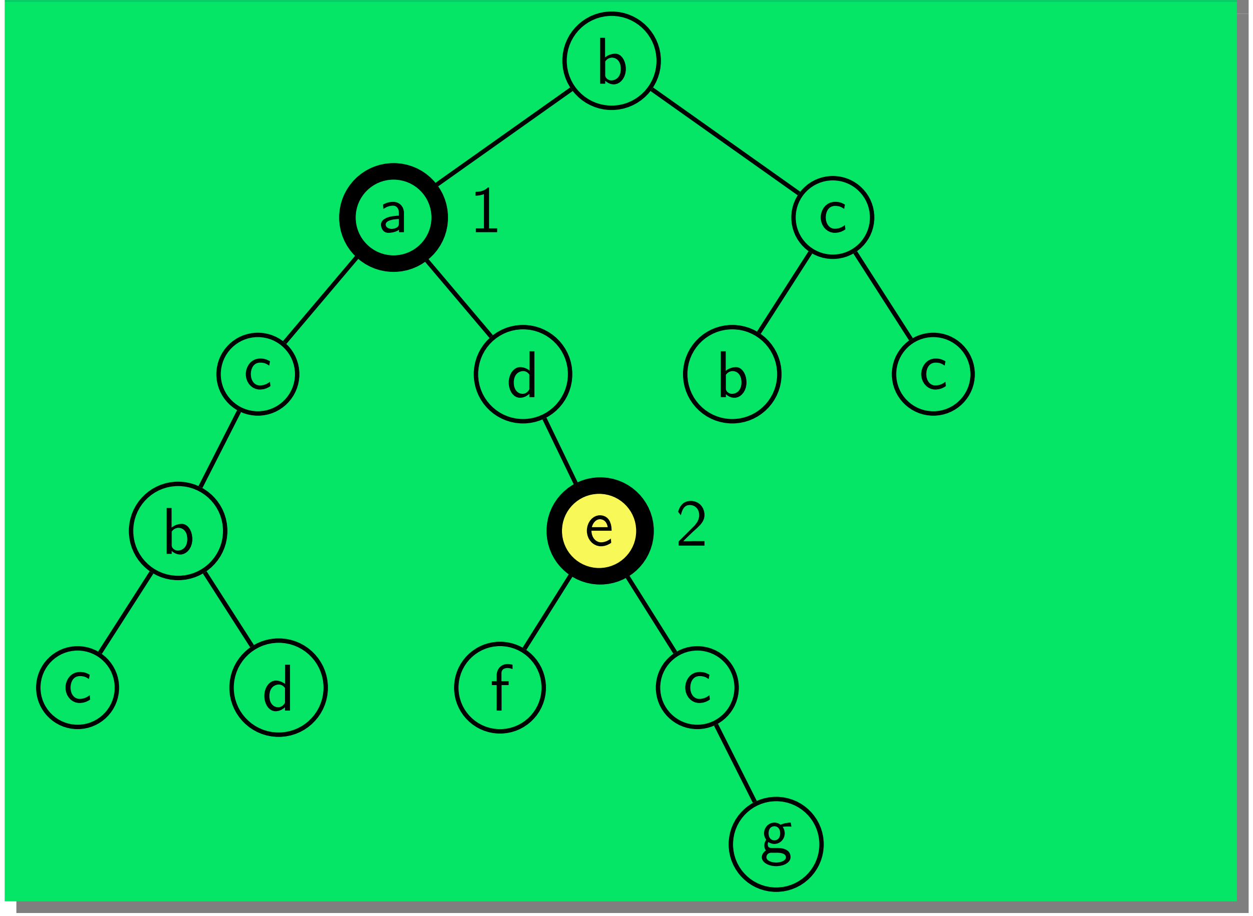

Example Tree

Pebble Automata: Example

Example Query Tree

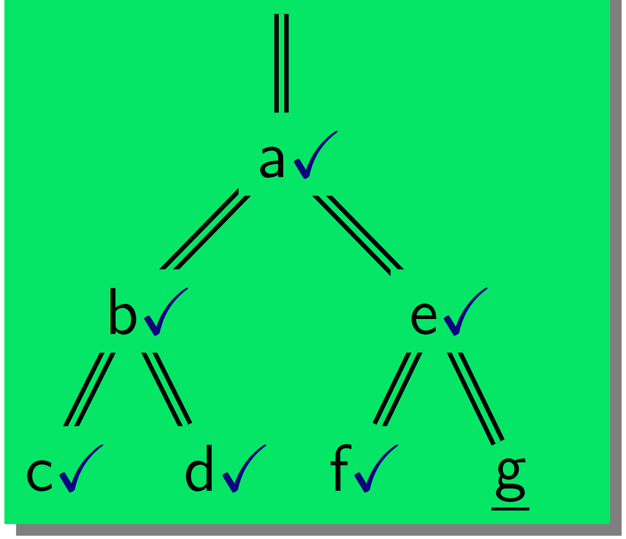


Example Tree

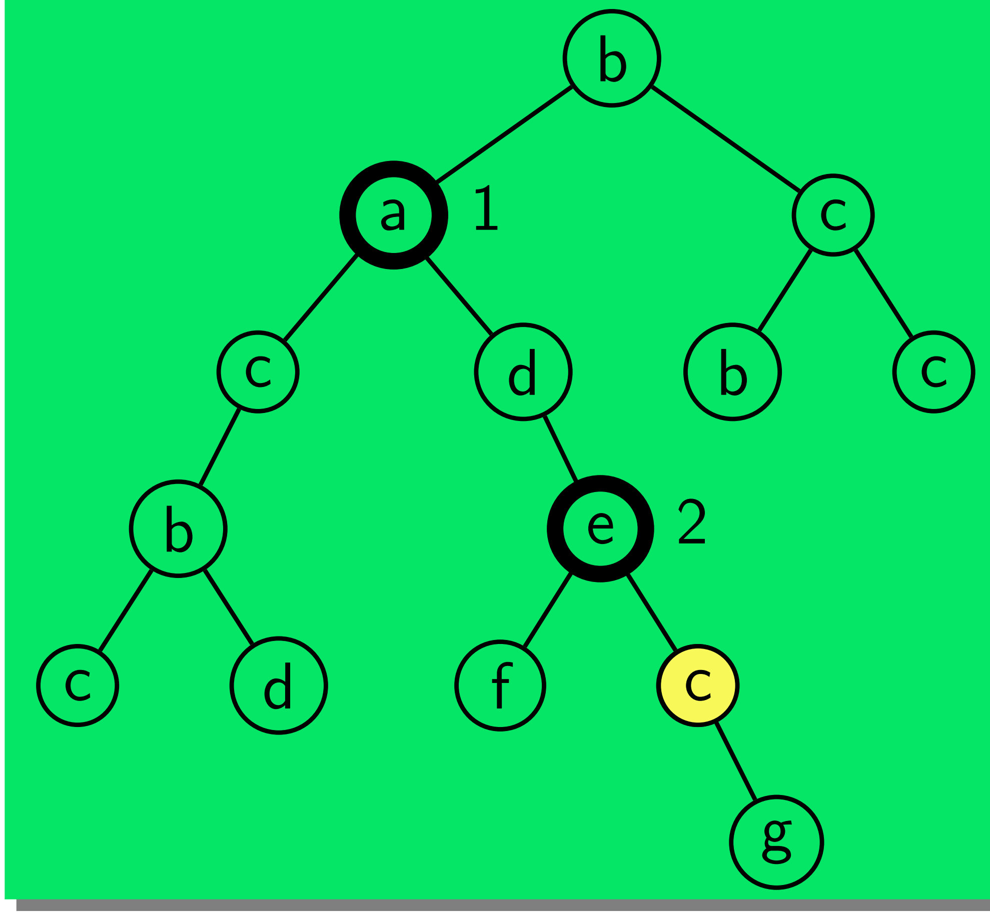


Pebble Automata: Example

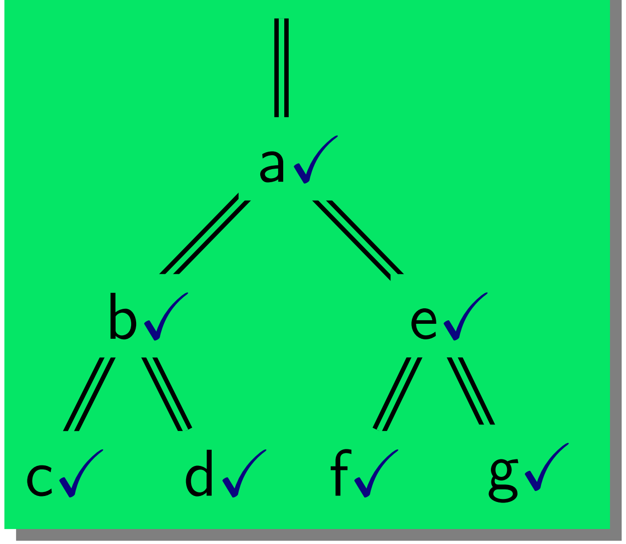
Example Query Tree



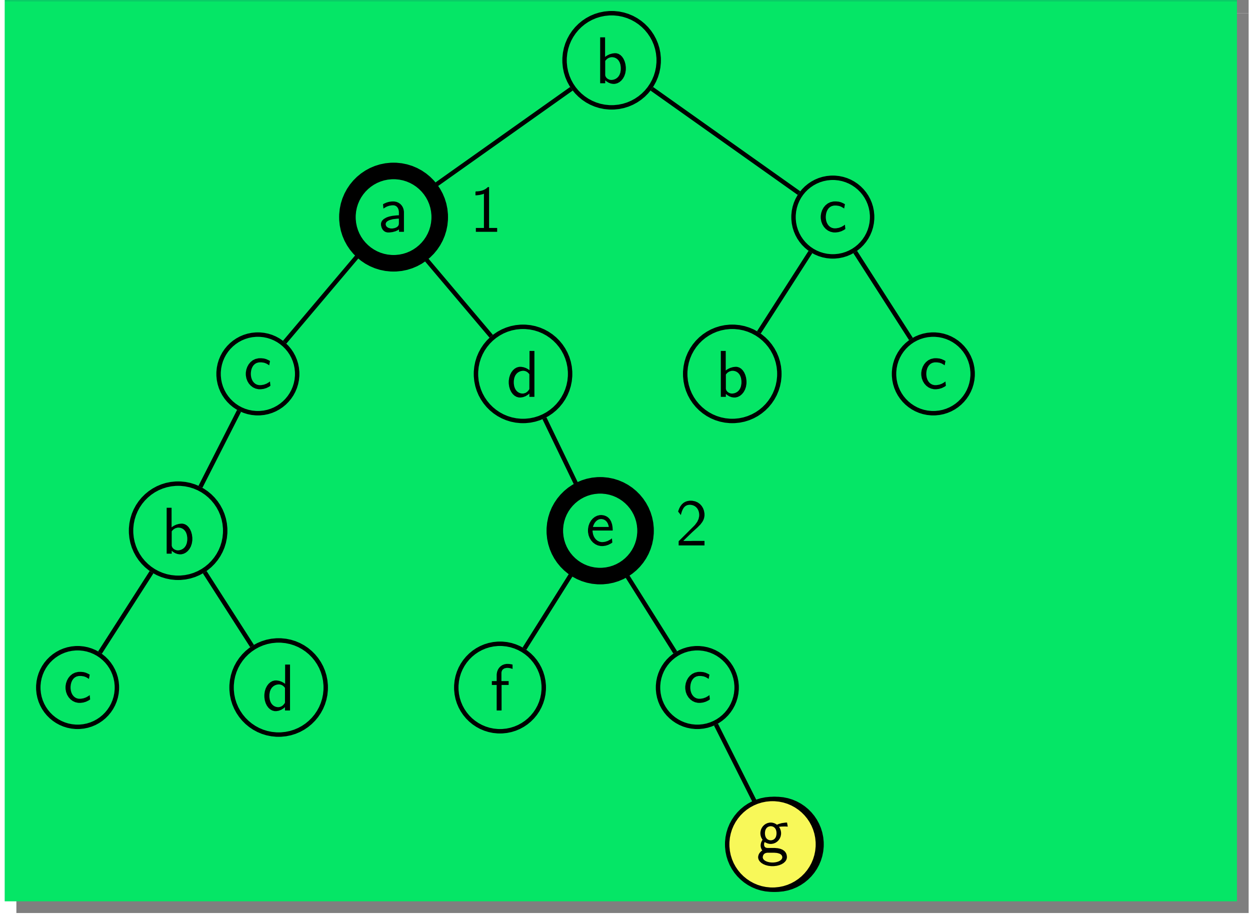
Example Tree



Example Query Tree



Example Tree



Definition (Pebble Automata)

- Extension of tree-walk automata by fixed number k of pebbles
- Only pebble with highest number (current pebble) can move, depending on state, number of pebbles symbols under pebbles and incidence of pebbles
- Possible pebble movements:
  - stay, go to left sibling, go to right sibling, go to parent
  - lift current pebble or place new pebble at current position
- Nondeterminism possible

Facts

- Pebble automata capture navigational `XPath` queries
- Extended by alternation, branching and an output mechanism they even capture a large part of `XSLT` [Papakonstantinou, Vianu 2000]

Some observations

- On strings, MSO logic and (unary) transitive closure logic (TC-logic) coincide
- On trees
  - MSO $\equiv$ parallel automata
  - TC-logic $\equiv$ pebble automata (i.e., strongest sequential automata)
- Whether MSO $\equiv$ TC-logic is open
- First-order logic $\equiv$ `XPath` $+$ conditional axes [Marx 2004]
- The relationship between logics and automata models between FO and TC-logic is largely unexplored:
  - Tree-walk automata,
  - FO-logic $+$ regular expressions
  - Conditional `XPath` $+$ arbitrary star operator
  - ...

Summary

- There is a natural notion of regular node-selecting queries generalizing regular tree languages
- Probably for most practical purposes too strong
- But it offers a useful framework for the study of other classes of queries
- A robust but weaker class of queries is captured by pebble automata

I

- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II

- Schemas
- Node Selecting Queries
- **XPath Query Containment**

III

- Extensions
- Conclusion

Recall our `XPath` Example

Example query

`//Vita/Died/*`

Example document

⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
    ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
    ⟨Movements⟩ 3 ⟨/Movements⟩
    ...
  ⟨/Piece⟩
  ...
⟨/Composer⟩
...

Recall our `XPath` Example

Example query

`//Vita/Died/*`

Example document

⟨Composer⟩
⟨Name⟩ Claude Debussy ⟨/Name⟩
⟨Vita⟩
⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
⟨/Vita⟩
⟨Piece⟩
⟨PTitle⟩ La Mer ⟨/PTitle⟩
⟨PYear⟩ 1905 ⟨/PYear⟩
⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
⟨Movements⟩ 3 ⟨/Movements⟩
...
⟨/Piece⟩
...
⟨/Composer⟩
...

Example document

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Another example query

`(/*[Name]//When) | (//Where)`

Example doc

⟨Composer⟩
  ⟨Name⟩ Claude Debussy ⟨/Name⟩
  ⟨Vita⟩
    ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
    ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
    ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
    ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
  ⟨/Vita⟩
  ⟨Piece⟩
    ⟨PTitle⟩ La Mer ⟨/PTitle⟩
    ⟨PYear⟩ 1905 ⟨/PYear⟩
    ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
    ⟨Movements⟩ 3 ⟨/Movements⟩
    ...
  ⟨/Piece⟩
  ...
⟨/Composer⟩
...

Example doc

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩  ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩⟨/Born⟩
      ⟨Married⟩⟨When⟩ October 1899 ⟨/When⟩⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩⟨When⟩ January 1908 ⟨/When⟩⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩  ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Another example query

`(/*[Name]//When) | (//Where)`

More `XPath` operators

Operator	Meaning
p/q	child
$p//q$	descendant
$p[q]$	filter
$*$	wildcard
$p \mid q$	disjunction

Example doc

```
⟨Composer⟩
   ⟨Name⟩ Claude Debussy ⟨/Name⟩
   ⟨Vita⟩
      ⟨Born⟩ ⟨When⟩ August 22, 1862 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Born⟩
      ⟨Married⟩ ⟨When⟩ October 1899 ⟨/When⟩ ⟨Whom⟩ Rosalie⟨/Whom⟩⟨/Married⟩
      ⟨Married⟩ ⟨When⟩ January 1908 ⟨/When⟩ ⟨Whom⟩ Emma ⟨/Whom⟩⟨/Married⟩
      ⟨Died⟩ ⟨When⟩ March 25, 1918 ⟨/When⟩⟨Where⟩ Paris ⟨/Where⟩ ⟨/Died⟩
   ⟨/Vita⟩
   ⟨Piece⟩
      ⟨PTitle⟩ La Mer ⟨/PTitle⟩
      ⟨PYear⟩ 1905 ⟨/PYear⟩
      ⟨Instruments⟩ Large orchestra ⟨/Instruments⟩
      ⟨Movements⟩ 3 ⟨/Movements⟩
      ...
   ⟨/Piece⟩
   ...
⟨/Composer⟩
...
```

Another example query

`(/*[Name]//When) | (//Where)`

More XPath operators

Operator	Meaning
p/q	child
$p//q$	descendant
$p[q]$	filter
$*$	wildcard
$p \mid q$	disjunction

Question

Does `//Vita/Died/*` always select a subset of positions of `(/*[Name]//When) | (//Where)`?

Question

Does `//Vita/Died/*` always select a subset of positions of `(/*[Name]//When) | (//Where)`?

Answer

No!

Question

Does `//Vita/Died/*` always select a subset of positions of `(/*[Name]//When) | (//Where)`?

Answer

No!

Counter-example

⟨Vita⟩
    ⟨Died⟩
        ⟨How⟩ Heart disease ⟨/How⟩
    ⟨/Died⟩
⟨/Vita⟩

Question

Does `//Vita/Died/*` always select a subset of positions of `(/*[Name]//When) | (//Where)`?

Answer

No!

Counter-example

⟨Vita⟩
  ⟨Died⟩
    ⟨How⟩ Heart disease ⟨/How⟩
  ⟨/Died⟩
⟨/Vita⟩

Further question

But what if the type of documents is constrained?

Fact

For all XML documents of type

```
<!DOCTYPE Composers [
    <!ELEMENT Composers (Composer*)>
    <!ELEMENT Composer (Name, Vita, Piece*)>
    <!ELEMENT Vita (Born, Married*, Died?)>
    <!ELEMENT Born (When, Where)>
    <!ELEMENT Married (When, Whom)>
    <!ELEMENT Died (When, Where)>
    <!ELEMENT Piece (PTitle, PYear,
         Instruments, Movements)>
    ]>
```

the pattern `//Vita/Died/*` always selects a subset of positions of

`(/*[Name]//When) | (//Where)`

XPath Containment: Definition

Definition (Containment for XPath(S))

Let S be a set of XPath-operators. The containment problem for XPath(S) is:

Given: XPath(S)-expression p, q

Question: Is $p(t) \subseteq q(t)$ for all documents t?

Definition (Containment for XPath (S) with DTD)

Let S be a set of XPath-operators. The containment problem for XPath(S) in the presence of DTDs is:

Given: XPath(S)-expression p, q, DTD d

Question: Is $p(t) \subseteq q(t)$ for all documents t satisfying $t \models d$?

Observation

These problems are crucial for static analysis and query optimization

Question

For which fragments S are these problems

- decidable?
- efficiently solvable?

General remarks

- The `XPath` containment problem has been considered for various sets of operators
- Results vary from **PTIME** to “undecidable”
- Various methods have been used:
  - Canonical model technique
  - Homomorphism technique
  - Chase technique
- More about this in [Miklau, Suciu 2002; Deutsch, Tanen 2001; Sch. 2004]
- We will consider automata based techniques

Definition (Relative Containment for `XPath` (S) wrt DTD)

Let S be a set of `XPath`-operators. The containment problem for `XPath`(S) relative to a DTD is:

Given: `XPath`(S)-expression p, q, DTD d

Question: Is $p(D) \subseteq q(D)$ for all documents D satisfying $D \models d$?

A vague plan

- Construct an automaton $\mathcal{A}_p$ for p
- Construct an automaton $\mathcal{A}_q$ for q
- Construct an automaton $\mathcal{A}_d$ for d
- Combine these automata suitably to get an automaton which accepts all counter-example documents

XPath Query Containment. — A Simplification

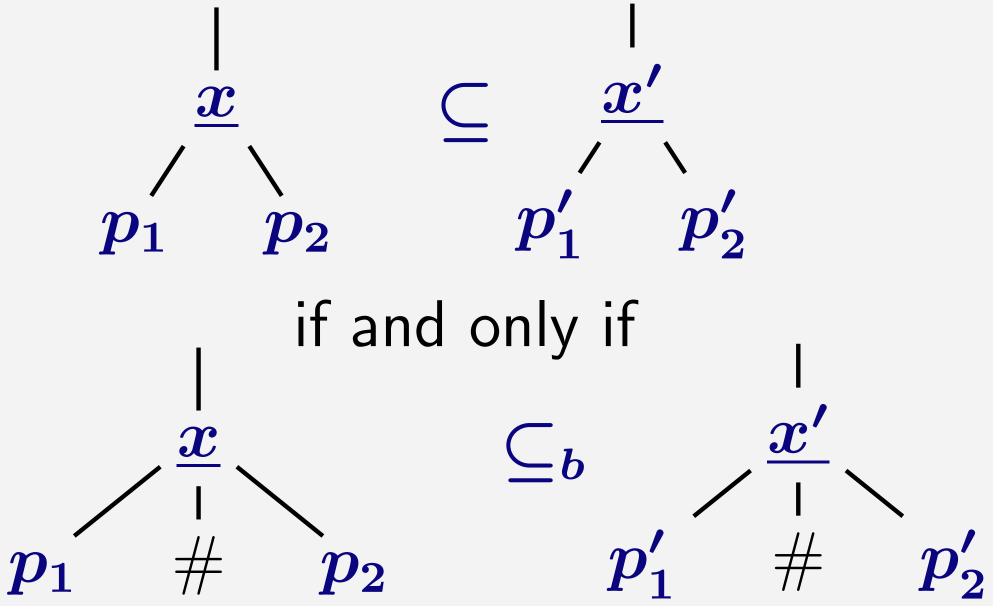
Definition (Boolean containment)

$p \subseteq_b q$ $:\Longleftrightarrow$ whenever p selects *some* node in a tree t then q also selects some node in t.

Useful observation [Miklau, Suciu 2002]

In the presence of [ ], Boolean containment has the same complexity as containment.

Crucial idea



`XPath` Containment: 2 Examples

Result 1 [Neven, Sch. 2003]

The Boolean containment problem for $\mathtt{XPath}(/, //)$ in the presence of DTDs is in **PTIME**

Result 2 [Neven, Sch. 2003]

The Boolean containment problem for $\mathtt{XPath}(/, //, [\,], *, |)$ in the presence of DTDs is in **EXPTIME**

Note

Both results are optimal wrt complexity:
the problems are complete for these classes

Containment for XPath(/, //) and DTDs

Result 1 [Neven, Sch. 2003]

The Boolean containment problem for XPath(/, //) in the presence of DTDs is in **PTIME**

Proof Idea

- XPath(/, //)-expressions can only describe vertical paths in a tree
- Each expression is basically of the form $p_1//p_2//\cdots//p_k$, where each p_i is of the form $l_{i1}/\cdots/l_{im_i}$
- On strings this is a sequence of string matchings corresponding to a regular language L

⇒ Deterministic string automaton of linear size

- Recall: there is a deterministic top-down automaton which checks whether a p-path exists

⇒ Deterministic top-down automaton $\mathcal{A}_p$

⇒ Deterministic top-down automaton $\mathcal{A}_{\overline{q}}$ checking that no q-path exists

Containment for $\texttt{XPath}(/,//)$ and DTDs

Result 1 [Neven, Sch. 2003]

The containment problem for $\texttt{XPath}(/,//)$ in the presence of DTDs is in **PTIME**

Proof idea (cont.)

- Deterministic top-down automaton $\mathcal{A}_p$
- Deterministic top-down automaton $\mathcal{A}_{\overline{q}}$ checking that no q-path exists
- There is a deterministic top-down automaton $\mathcal{A}_d$ checking whether t conforms to d
- $p \subseteq_b q$ in the presence of $d \iff L(\mathcal{A}_p \times \mathcal{A}_{\overline{q}} \times \mathcal{A}_d) = \emptyset$
- The latter can be checked in polynomial time

Containment for $\texttt{XPath}(/, //, [\,], *, |)$ and DTDs

Result 2 [Neven, Sch. 2003]

The containment problem for $\texttt{XPath}(/, //, [\,], *, |)$ in the presence of DTDs is in **EXPTIME**

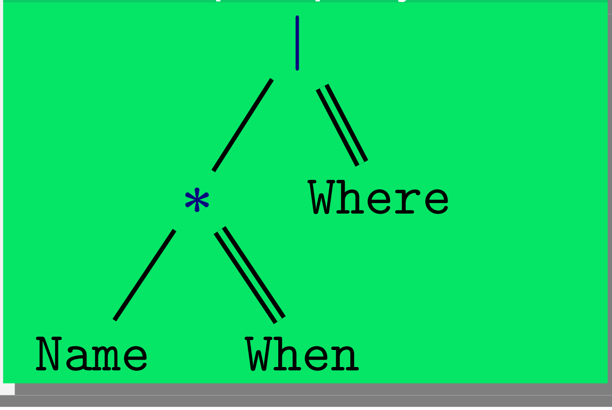
Proof Idea

We again represent patterns like

`(/*[Name]//When) | (//Where)`

as query trees:

Example query tree



Lemma

For each $\texttt{XPath}(/, //, [\,], *, |)$-expression p there is a deterministic bottom-up automaton $\mathcal{A}_p$ of exponential size which checks whether in a tree p holds

Containment for $\texttt{XPath}(/,//,[\,],*,|)$ and DTDs

Lemma

For each $\texttt{XPath}(/,//,[\,],*,|)$-expression p there is a deterministic bottom-up automaton $\mathcal{A}_p$ of exponential size which checks whether in a tree p holds

Proof idea for Lemma

- States of $\mathcal{A}_p$ are of the form $(S_/, S_{//})$
- Both $S_/$ and $S_{//}$ are sets of positions of the query tree:
  - $S_/$: positions matching v
  - $S_{//}$: positions matching some node in the subtree of v

Containment for $\texttt{XPath}(/, //, [\,], *, |)$ and DTDs

Result 2 [Neven, Sch. 2003]

The containment problem for $\texttt{XPath}(/, //, [\,], *, |)$ in the presence of DTDs is in **EXPTIME**

Proof idea (cont.)

- Construct deterministic bottom-up automaton $\mathcal{A}_p$ of exponential size
- Construct deterministic bottom-up automaton $\mathcal{A}_{\overline{q}}$ of exponential size
- Construct deterministic bottom-up automaton $\mathcal{A}_d$ of exponential size
- $p \subseteq_b q$ in the presence of $d \iff L(\mathcal{A}_p \times \mathcal{A}_{\overline{q}} \times \mathcal{A}_d) = \emptyset$

$\Rightarrow$ exponential time

Summary (Automata and `XPath` containment)

- Automata are a useful algorithmic tool
- In particular, if several algorithmic tasks have to be combined
- Complexity depends on type of automata

Summary (`XPath` containment in general)

- Many more results in other papers, e.g., [Miklau, Suciu 2002; Deutsch, Tanen 2001; Sch. 2004]
- The complexity of `XPath` query containment varies strongly with the allowed operators
- Even undecidable in general
- Exact borderline between undecidable and decidable has to be identified

I

- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II

- Schemas
- Node Selecting Queries
- XPath Query Containment

III

- Extensions
- Conclusion

Pebble automata

- As mentioned before: XSLT transormations can be modeled by k-pebble transducers (k-pebble automata $+$ alternation, branching, output)
- Pebbles are mainly used to evaluate `XPath` expressions

XSLT Typechecking problem

Given: Transformation T, Schemas d_1, d_2

Question: Is $T(t)$ valid wrt d_2 whenever t is valid wrt d_1?

Theorem (Milo, Suciu, Vianu 2000)

The typechecking problem for (core) XSLT is decidable

Extensions

XSLT Typechecking problem (cont.)

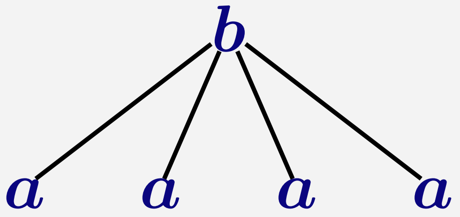
Theorem (Milo, Suciu, Vianu 2000)

The typechecking problem for (core) XSLT is decidable

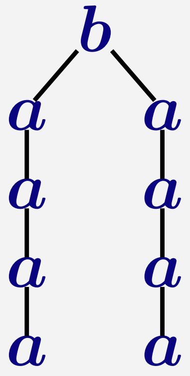
Proof Idea

- Obvious approach:
  - Compute $T(L(d_1))$
  - Check that $T(L(d_1)) \subseteq L(d_2)$
- Problem: $T(L(d_1))$ does not need to be regular:

  Transform



into



- Better approach:

  Compute $T^{-1}(L(d_2))$ and check $L(d_1) \subseteq T^{-1}(L(d_2))$

Extensions — **XSLT Typechecking problem (cont.)**

Proof idea (cont.)

- **k-pebble acceptor**: k-pebble transducer without output
- Prove: $T^{-1}(L)$ is accepted by a k-pebble acceptor if L is regular
- Prove: Behavior of k-pebble acceptors can be described by MSO-formulas

$\Rightarrow$ k-pebble acceptors only accept regular tree languages

$\Rightarrow$ $T^{-1}(\overline{L(d_2)})$ is regular

- Algorithm:
  - Construct automaton for $T^{-1}(\overline{L(d_2)})$
  - Construct an equivalent MSO-formula φ
  - Construct bottom-up tree automaton $\mathcal{A}$ for $\neg\varphi$
  - Check that $L(d_1) \subseteq L(\mathcal{A})$
- Complexity: VERY bad (non-elementary)

So far...

- We have seen that automata are useful for
  - Validation, Typing
  - Navigation
  - Transformation
- What about more general queries?
  - results of higher arity?
  - joins, i.e., comparisons of data values
  - counting
- Are automata useful for `XQuery`?
- ... for tree pattern queries?

General Queries (cont.)

Higher arity

- Nonemptiness and containment questions can be handled by automata: tuples can be encoded by additional labels
- What about query evaluation for higher arity?

Data values

- When data values in XML documents are taken into account, things become more complicated, e.g.:
  - Even First-order logic becomes undecidable
  - Pebble automata become undecidable
  - Automata with data registers become undecidable when they are allowed to move up and down
- What is the right notion for regular (string) languages over infinite alphabets?
- What are sensible decidable restrictions of logics and automata in the context of data values?

Counting

- Automata can be equipped with counting facilities, e.g.:
  Presburger tree automata: $\delta(\sigma, q)$ is Boolean combination of
  - regular expressions and
  - quantifier-free Presburger formulas like
    "number of children in state q_1 $=$ number of children in state q_2"
- Nondet. Presburger automata:
  - $\equiv$ MSO logic
  - Whether automaton accepts all trees is undecidable
- Det. Presburger automata:
  - $\equiv$ Presburger μ-formulas
  - Membership test: $O(|\mathcal{A}||t|)$
  - Non-emptiness: **PSPACE**
  - Containment: **PSPACE**

[Seidl, Sch., Muscholl, Habermehl 2004]

I
- Introduction
- Pre-XML Tree Automata
- Tree Automata for XML

II
- Schemas
- Node Selecting Queries
- XPath Query Containment

III
- Extensions
- Conclusion

We saw...

- A broad variety of automata models which can be used for XML and its theory
- Well-established in the context of validation, typing, navigation, transformation
- Well-established as
  - means to define robust classes
  - proof tools
  - algorithmic tools

Big question

Can automata be employed as a tool for `XQuery` evaluation?