XML and Databases

Lecture 9
Properties of XPath

Sebastian Maneth
NICTA and UNSW

CSE@UNSW -- Semester 1, 2010

Outline

1. XPath Equivalence
2. No Looking Back: How to Remove Backward Axes
3. Containment Test for XPath Expressions

A Note on Equality Test
in XPath

XPath 2.0 has clearer comparison operators!

Useful Functions (on Node Sets)

Careful with equality ("=")

\[//a[b/d] = c/d\]

selects a node!

there is a node in the node set for \[b/d\]
with same string value as a node in node set \[c/d\]

A Note on Equality Test

XPath 2.0

\[//a[b/d] = c/d\]

selects what?

false (on any document)

A Note on Equality Test

XPath 2.0

\[//a[b/d] = c/d\]

selects what?
A Note on Equality Test

Recall

\[ \text{child:}^* \] all child nodes that are elements
\[ \text{child:comment()} \] all child nodes that are comments
\[ \text{child:processing-instruction()} \] all child nodes that are proc. inst.'s
\[ \text{child:nodes()} \] all child nodes that are elements/comments/PIs

(only way to get to an attribute, is via the attribute-axis)

Question

What axes can bring you from an attribute-node back to an element-node?

\[ \text{false} \] (on any document)

\[ //a\[b/d == c/d\] \] selects what?

\[ <a> <b> <d>red</d> <d>green</d> <d>blue</d> </b> <c> <d>yellow</d> <d>orange</d> <d>green</d> </c> </a> false (on any document)

\[ //a[b/d == c/d] \] selects what?

\[ //a//b//c[nodes(): last()] \] selects what?

Recall

\[ \text{child::* all child nodes that are elements} \]
\[ \text{child::comment() all child nodes that are comments} \]
\[ \text{child::processing-instruction() all child nodes that are proc. inst.'s} \]
\[ \text{child::node() all child nodes that are element/comments/PIs} \]

(only way to get to an attribute, is via the attribute-axis)

Question

Which axes can bring you from an attribute-node back to an element-node?

\[ \text{false} \] (on any document)

\[ //a[b/d == c/d] \] selects what?

\[ //a//b//c[nodes(): last()] \] selects what?

Recall

\[ \text{child::* all child nodes that are elements} \]
\[ \text{child::comment() all child nodes that are comments} \]
\[ \text{child::processing-instruction() all child nodes that are proc. inst.'s} \]
\[ \text{child::node() all child nodes that are element/comments/PIs} \]

(only way to get to an attribute, is via the attribute-axis)

Question

Which axes can bring you from an attribute-node back to an element-node?

\[ \text{false} \] (on any document)

\[ //a[b/d == c/d] \] selects what?

\[ //a//b//c[nodes(): last()] \] selects what?

1. XPath Equivalence

EBNF for XPaths that we want to consider now:

path := path | /path | /path / path | path / path | path / qual | node:: | ass := modtest | .
qual := qual && qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual || qual |
1. XPath Equivalence

**Lesson 3.2.** Let \( m \) and \( n \) be node tests, let \( m \) and \( n \) be any names or one of the XPath constructs \( 
\text{node()}, \text{ancestor()}. \) or \( \text{child()}. \)

- Let \( a \) be one of the axis \( \text{parent}, \text{ancestor}, \text{preceding}, \text{preceding-sibling}, \text{self}, \) \( \text{following}, \) or \( \text{following-sibling} \). Then the following holds:
  
  \[
  \text{path}(x) = \begin{cases} 
  \text{if } x = \text{self} \text{ and } m = n \text{ node()} \text{ otherwise} 
  \end{cases}
  \]

- Let \( a \) be the \text{preceding} or \text{ancestor} axis. Then the following equivalences hold:
  
  \[
  \text{dual}(x) = \begin{cases} 
  \text{if } x = \text{ancestor and } m = \text{node()} \text{ otherwise} 
  \end{cases}
  \]
  
  \[
  \text{dual}(x) = \begin{cases} 
  \text{if } x = \text{ancestor and } m = \text{node()} \text{ otherwise} 
  \end{cases}
  \]

  \[
  \text{(same holds for } a = \text{parent)}
  \]

Thus:  dual(parent) = child

\[
\text{dual(following)} = \text{preceding}
\]

etc.

**Rewrite rule #1** \( (p, s: \text{relative paths}, ax: \text{reverse axis}) \)

\[
p[ax::m/s] \Rightarrow p[\text{ancestor}::m] \text{ descendant::node()} = \text{self::node()}
\]

E.g. \( ax = \text{ancestor} \)

\[
p[\text{ancestor::m}] \Rightarrow p[\text{descendant::m}] \text{ descendant::node()} = \text{self::node()}
\]

"any \( m \)-node from which the context node can be reached via descendant, must be an ancestor of the context node."

"any \( m \)-node from which the context node can be reached via \( \text{dual} \)-axis, must reach context node"

Similar for parent and preceding. (Ancestor-or-self not really needed. Why?)

2. No Looking Back

**Dual**

- \text{parent}
- \text{child}
- \text{ancestor}
- \text{ancestor-or-self}
- \text{preceding}
- \text{following}
- \text{following-sibling}

**Forward**

- \text{descendant}
- \text{descendant-or-self}
- \text{preceding-sibling}

Thus: dual(child) = preceding

\[
p[ax::m/s] \Rightarrow p[\text{descendant}::m] \text{ dual}(ax)::node() = \text{self::node()}
\]

**Rewrite rule #1** \( (p, s: \text{relative paths}, \text{ax: reverse axis}) \)

\[
p[ax::m/s] \Rightarrow p[\text{descendant}::m] \text{ dual}(ax)::node() = \text{self::node()}
\]

E.g. \( ax = \text{preceding-sibling} \)

\[
p[\text{preceding-sibling::m}] \Rightarrow p[\text{following-sibling::m}] \text{ following-sibling::node()} = \text{self::node()}
\]

"any \( m \)-node from which the context node can be reached via following-sibling, must be a preceding-sibling of the context node."
Rewrite rule #1
(p,s: relative paths, ax: reverse axis)
\[ p[ax::m/s] \rightarrow p[\text{descendant::m}[s]/\text{dual}(ax)::node()] == \text{self::node()} \]

Removes first reverse axis inside a filter (qualifier).
Use qualifier flattening to replace "any" reverse axis from inside a filter.

Qualifier Flattening
\[ p[p1/p2] = p[p1[p2]] \]

Similar rules for absolute paths:
\[ /p/ax::n/\text{dual}(ax)::node() == /p/ax::n \]
\[ /ax::n/\text{dual}(ax)::node() == /ax::n \]

Rewrite rules #2 and #2a

E.g.
\[ /\text{descendant::price}/\text{preceding::name} \]
is rewritten via Rule #2a into:
\[ /\text{descendant::name}[\text{following::price}==/\text{descendant::price}] \]

Similar rules for absolute paths:
\[ /p/ax::n/\text{dual}(ax)::node() == /p/ax::n \]
\[ /ax::n/\text{dual}(ax)::node() == /ax::n \]

Rewrite rules #2 and #2a

E.g.
\[ /\text{descendant::journal}/\text{child::title}/\text{descendant::price}/\text{preceding::name} \]
becomes
\[ /\text{descendant::name}[\text{following::price}==/\text{descendant::journal}/\text{child::title}/\text{descendant::price}] \]

Of course, the "join" can be removed in this example:
\[ /\text{descendant::name}[\text{following::price}] \]

Can you avoid the join, also for this example?

Similar rules for absolute paths:
\[ /p/ax::n/\text{dual}(ax)::node() == /p/ax::n \]
\[ /ax::n/\text{dual}(ax)::node() == /ax::n \]

Rewrite rules #2 and #2a

2. No Looking Back

<table>
<thead>
<tr>
<th>Dual</th>
<th>backward</th>
<th>forward</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent</td>
<td>child</td>
<td>ancestor</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>descendant</td>
<td>parent-or-self</td>
</tr>
<tr>
<td>preceding</td>
<td>following</td>
<td>preceding-or-self</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td>following-sibling</td>
<td>not needed</td>
</tr>
</tbody>
</table>

Joins (\(==\)) are expensive! (typically quadratic wrt data)
To obtain queries with fewer joins consider the forward-axis left of the reverse-axis to be removed!

Rules (1),(2),(2a) suffice to remove ALL backward axes from above queries!
Why?
- Size increase?
- How many joins?

New rules will be of the form
\[ p/\text{forw}/\text{back} \rightarrow p/\text{new} \]
Interaction of `back=parent` with forward axes:

```
| descendant::/parent::m = descendant-or-self::m[child::m] | (3) |
| child::/parent::m = self::m[child::m] | (4) |
| p/self::/parent::m = p/self::m/parent::m | (5) |
| p//following-sibling::/parent::m = p//following-sibling::m/parent::m | (6) |
```

```
| descendant::/parent::m = descendant-or-self::m[child::m] | (3) |
| child::/parent::m = self::m[child::m] | (4) |
| p/self::/parent::m = p/self::m/parent::m | (5) |
| p//following-sibling::/parent::m = p//following-sibling::m/parent::m | (6) |
```

2. No Looking Back

Interaction of \texttt{back=ancestor} with forward axes:

\begin{align*}
\text{\texttt{descendant}} \rightarrow \text{\texttt{ancestor}} \\
\text{\texttt{descendant-or-self}} \rightarrow \text{\texttt{ancestor}} \\
\text{\texttt{descendant-or-self}} \rightarrow \text{\texttt{ancestor-or-self}}
\end{align*}

Similar rules for \texttt{ancestor} in a filters.
2. No Looking Back

Interaction of \textit{back=ancestor} with forward axes:

\[
\begin{align*}
&/\text{descendant::} m \text{ \backslash descendant::} n /\text{ancestor::} m \\
&/\text{descendant::} m \text{ \backslash descendant-or-self::} n /\text{ancestor::} m \\
&/\text{child::} m \text{ \backslash descendant-or-self::} n /\text{ancestor::} m
\end{align*}
\]

Similar rules for \textit{ancestor} in a filters.

E.g., what is the forward query for: \texttt{//*[:,\text{ancestor::} a]}

\textbf{Rule 33}

Wrong. Should be \textit{descendant} instead!

Rule (33):

\[
/\text{descendant::} m /\text{preceding::} n /\text{following::} m
\]

\texttt{p/child::*[following-sibling::*[descendant::n]]/descendant-or-self::m}

\texttt{p/child::*[following-sibling::*[descendant-or-self::m]]}

We obtain:

\[
/\text{descendant::name}[\text{following::price}]
\]

Rule (33a)

\[
/\text{descendant::} m /\text{preceding::} n \rightarrow /\text{descendant::} n /\text{preceding::} m
\]

\[
/\text{child::*}[\text{following-sibling::*[descendant-or-self::m]]/\text{descendant-or-self::m}]
\]

We obtain:

\[
/\text{descendant::price}[\text{preceding::na}\emptyset]
\]

\[
/\text{child::*[following-sibling::*[descendant-or-self::m]]/\text{descendant-or-self::m}]
\]
Given an XPath expression \( p \) that has no joins of the form \((p_1 \equiv p_2)\) with both \( p_1, p_2 \) relative, an equivalent expression \( u \) without reverse axes can be computed.

**Theorem**

- **Time needed:** at most exponential in length of \( p \)
- **Length of \( u \):** at most exponential in length of \( p \)

No joins are introduced when computing \( u \).

**More Questions**

- Why rewriting takes exponential time?
- Can you find a subclass for which \( \text{Time} \) to compute \( u \) is linear or polynomial?
- What is the problem with joins \((p_1 \equiv p_2)\) for removal of reverse axes?

### 3. XPath Containment Test

Given two XPath expressions \( p, q \):

- Are all nodes selected by \( p \), also selected by \( q \) (on any document)?

**Boolean query**

- Has many applications!

Want to select documents that "match \( p \)"
- \( \text{if a document matches } p, \text{ and } p \text{ contained in } q, \text{ then we know the document also matches } q \)
- \( \text{if a document does not match } q, \text{ and } p \text{ contained in } q, \text{ then we know the document does not match } p \)

**Applications**

- Decrease online-time of publish/subscribe systems based on XPath
- Decrease query-time by making use of materialized intermediate results
- Optimization by ruling out queries with empty result set etc, etc
3. XPath Containment Test

Given two XPath expressions \( p, q \)

- **0-containment** For every tree, if \( p \) selects a node then so does \( q \).
  \[ p \subseteq_0 q \]

- **1-containment** For every tree, all nodes selected by \( p \) are also selected by \( q \).
  \[ p \subseteq_1 q \]

- **2-containment** For every tree, and every context node \( N \), all nodes selected by \( p \) starting from \( N \), are also selected by \( q \) starting from \( N \).
  \[ p \subseteq_2 q \]

1. Inclusion on Booleans
2. Inclusion on Node Sets
3. Inclusion on Node Relations

(If only child and descendant axes are allowed then \( \subseteq_1 \) and \( \subseteq_2 \) are the same! — Why?)

---

3. XPath Containment Test

Given two XPath expressions \( p, q \)

Sometimes we want to test containment wrt a given DTD:

\[ p = /a/b//d \]
\[ q = /a//c \]

Want to check if \( p \subseteq_0 q \).

NO! a b d
But, what if documents are valid wrt this DTD?

\[ \text{root} \rightarrow a^* \]
\[ a \rightarrow b^* | c^* \]
\[ b \rightarrow d^*+ \]
\[ c \rightarrow b?c? \]

3. XPath Containment Test

4 techniques of testing XPath (Boolean) containment:

1. The Canonical Model Technique
2. The Homomorphism Technique
3. The Automaton Technique
4. The Chase Technique
3. XPath Containment Test

Canonical Model - XPath(//, //, [], *)

Idea: if there exists a tree that matches p but not q, then such a tree exists of size polynomial in the size of p and q.

Simple: remember, if you know that the XML document is only of height 5, then //a/b/*/c could be enumerated by /a/b/*/c | /*a/b/*/c | /*/*/a/b/*/c | /*/*//*/a/b/*/c | /*/*//*//*/a/b/*/c | /*/*//*//*//*/a/b/*/c | ... | /*/*/.../*/a/b/*/c

Similarly, we try to construct a counter example tree, by replacing in p
- every * by some new symbol "z"
- every // by z/, z/z/, z/z/z/, ...

N = length of longest "..." chain in q

N+1 many z's

Example

Test for q-match:

Formally, must test 1 and 2 more z's at right branch of each of the trees.

Homomorphism h maps each node of q's query tree Q to a node of p's query tree P such that

(1) root of Q is mapped to root of P
(2) if (u,v) is child-edge of Q then (h(u),h(v)) is child-edge of P
(3) if (u,v) is descendant-edge of Q, then h(v) is a "below" h(u) in P
(4) if u is labeled by "e" (not *), then h(u) is also labeled by "e".

Theorem:

p ⊆ q if and only if there is a homomorphism from Q to P.

Homomorphism h maps each node of q's query tree Q to a node of p's query tree P such that

(1) root of Q is mapped to root of P
(2) if (u,v) is child-edge of Q then (h(u),h(v)) is child-edge of P
(3) if (u,v) is descendant-edge of Q, then h(v) is a "below" h(u) in P
(4) if u is labeled by "e" (not *), then h(u) is also labeled by "e".
### 3. XPath Containment Test

**Homomorphism** $h$ maps each node of q's query tree $Q$ to a node of p's query tree $P$ such that:

1. root of Q is mapped to root of P
2. if (u,v) is child-edge of Q then $(h(u),h(v))$ is child-edge of P
3. if (u,v) is descendant-edge of Q, then $h(v)$ is a "below" $h(u)$ in P
4. if u is labeled by "e" (not *), then $h(u)$ is also labeled by "e".

**Homomorphism $h$ exists from Q to P, thus $p \subseteq q$ must hold!**

Cave: If we add the star (\( * \)) then homomorphism need not exist!

**Theorem**

$p \subseteq q$ if and only if there is a homomorphism from Q to P.

Cave: if we add the star (\( * \)) then homomorphism need not exist!

**Question**

Given $p$, $q$ and the fact $p \subseteq q$, how can you determine from a result set of nodes for $q$, the correct result set of nodes for $p$?

**With homomorphism technique:**

Use a result node of $q$ together with run-time info on pattern nodes. Enables to search "inside", only on paths between pattern nodes.

**Query nodes of a q result**

**Theorem**

$p \subseteq q$ if and only if there is a homomorphism from Q to P.

**Cave**

If we add the star (\( * \)) then homomorphism need not exist!

**Question**

Given $p$, $q$ and the fact $p \subseteq q$, how can you determine from a result set of nodes for $q$, the correct result set of nodes for $p$?

**With homomorphism technique:**

Use a result node of $q$ together with run-time info on pattern nodes. Enables to search "inside", only on paths between pattern nodes.

**Query nodes of a q result**

**Theorem**

$p \subseteq q$ if and only if there is a homomorphism from Q to P.

**Cave**

If we add the star (\( * \)) then homomorphism need not exist!

**Question**

Given $p$, $q$ and the fact $p \subseteq q$, how can you determine from a result set of nodes for $q$, the correct result set of nodes for $p$?

**With homomorphism technique:**

Use a result node of $q$ together with run-time info on pattern nodes. Enables to search "inside", only on paths between pattern nodes.

**Query nodes of a q result**
If we add the star (\*), then homomorphism need not exist!

\[ p \subseteq q \]

where to map??

\[ p = \langle a, \{ b, c \} \rangle \]
\[ q = \langle a, \{ b, c \} \rangle \]

Let's check the web...

\[ \text{YES} \]

p contained in q!

Where is p contained in q?

\[ p = \langle a, \{ b, c \} \rangle \]
\[ q = \langle a, \{ b, c \} \rangle \]

Cave: If we add the star (\*), then homomorphism need not exist!

\[ \exists p, q \in \text{XPath}(\langle a \rangle, \langle b \rangle) \text{ such that } p \subseteq q \text{ and } \text{there is no homomorphism from } Q \text{ to } P \]

\[ p = \langle a, \{ b, c \} \rangle \]
\[ q = \langle a, \{ b, c \} \rangle \]

Cave: If we add the star (\*), then homomorphism need not exist!

\[ \exists p, q \in \text{XPath}(\langle a \rangle, \langle b \rangle) \text{ such that } p \subseteq q \text{ and } \text{there is no homomorphism from } Q \text{ to } P \]

3. XPath Containment Test

Automaton Technique

Recall: for any DTD there is a tree automaton which recognized the corresponding trees.

Similarly, for any \( \text{XPath}(\langle a \rangle, \langle b \rangle, \langle c \rangle) \) expression \( \text{ex} \) we can construct a (non-deterministic bottom-up) tree automaton \( A \) which accepts a tree if and only if \( \text{ex} \) matches the tree.

Theorem

Containment test of XPath(\langle a \rangle, \langle b \rangle, \langle c \rangle) in the presence of DTDs can be solved in \( \text{EXPTIME} \).

Proof Idea

construct automaton for all possible counter example trees. Test if this automaton accepts any tree.

Emptiness test for automata

\( p \subseteq q \), for all trees but finitely many exceptions?

solvable!
3. XPath Containment Test

Chase Technique -- 1979 relational DB's to check query containment in the presence of integrity constraints.

Example

DTD E =

\[
\begin{align*}
\text{root} & \rightarrow a^+ \\
b & \rightarrow b^+ | c^+ \\
c & \rightarrow b?c?
\end{align*}
\]

Is \( p \) contained in \( q \) for E-conform documents?

First Possibility: use tree automata

\[\text{Construct automata } A_p, A_q, A_{E} \]

\[\text{Construct } B_q \text{ for the complement of } A_q \]

\[\text{Intersect } B_q \text{ with } A_p \text{ and } A_{E} \text{ (gives automation } A) \]

\[\text{Check if } A \text{ accepts any tree.}\]

Example

\[ p = /a/b//d \]

\[ q = /a/c \]

DTD E =

Is \( p \) contained in \( q \) for E-conform documents?

("the chase" extends the relational homomorphism technique)

Each b-element has a d-child and a c-child

\[ c1: b \rightarrow d \]

\[ c2: b \rightarrow c \]

p's pattern tree after chasing with c1,c2

\[ p \text{ is contained in } q \text{ in the presence of the DTD } E \]

p's pattern tree after chasing with c1,c2

END

Lecture 8