

Programming Languages for XML

- XPath
- XQuery
- Extensible StyleSheets Language (XSLT)

XPath

XPath is a simple language for describing sets of similar paths in a graph of semistructured data.

The XPath Data Model

Sequence of items corresponds to a set of tuples in the relational algebra.

An item is either:

1. A value of primitive type: integer, real, boolean, or string.
2. A node (three kinds of nodes)

Three kinds of nodes:

- (a) Documents. These are files containing an XML document, perhaps denoted by their local path name or URL.
- (b) Elements. These are XML elements, including their opening tags, their matching closing tags if there is one, and everything in between (i.e., below them in the tree of semistructured data that an XML document represents).
- (c) Attributes. These are found inside opening tags.

The items in a sequence needn't be all of the same type although often they will be.

A sequence of five items:

10

“ten”

10.0

<Number base = “8”>

 <Digit>1</Digit>

 <Digit>2</Digit>

</Number>

@val=“10”

Document Nodes

It is common to apply XPath to documents that are files. We can make a document node from a file by applying the function:

```
doc(file name)
```

The named file should be an XML document. We can name a file either by giving its local name or a URL if it is remote.

```
doc("movie.xml")
```

```
doc("/usr/slly/data/movies.xml")
```

```
doc("infolab.stanford.edu/~hector/movies.xml")
```

Path Expressions

An XPath expression starts at the root of a document and gives a sequence of tags and slashes (/).

```
doc(file name)/ $T_1$ / $T_2$ /.../ $T_n$ 
```

```
doc("movie.xml")/StarMoviedata/Star/Name
```

Evaluation of XPath expressions:

1. Start with a sequence of items consisting of one node: the document node.
2. Then, process each of T_1, T_2, \dots, T_n in turn.
3. To process T_i , consider the sequence of items that results from processing the previous tag, if any. Examine those items, in order, and find each of all its subelements whose tag is T_i .

Programming Language for XML

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
```

```
<StarMovieData>
```

```
  <Star starID = "cf" starredIn = "sw">
```

```
    <Name>Carrie Fishes</Name>
```

```
    <Address>
```

```
      <Street>123 Maple St.</Street><City>Hollywood</City>
```

```
    </Address>
```

```
    <Address>
```

```
      <Street>5 Locust Ln.</Street><City>Malibu</City>
```

```
    </Address>
```

```
  </Star>
```

```
  <Star starID = "mh" starredIn = "sw">
```

```
    <Name>Mark Hamill</Name><Street>456 Oak Rd.</Street>
```

```
    <City>Brentwood</City>
```

```
  </Star>
```

```
  <Movie movieID = "sw" starOf = "cf mh">
```

```
    <Title>Star Wars</title><Year>1977</Year>
```

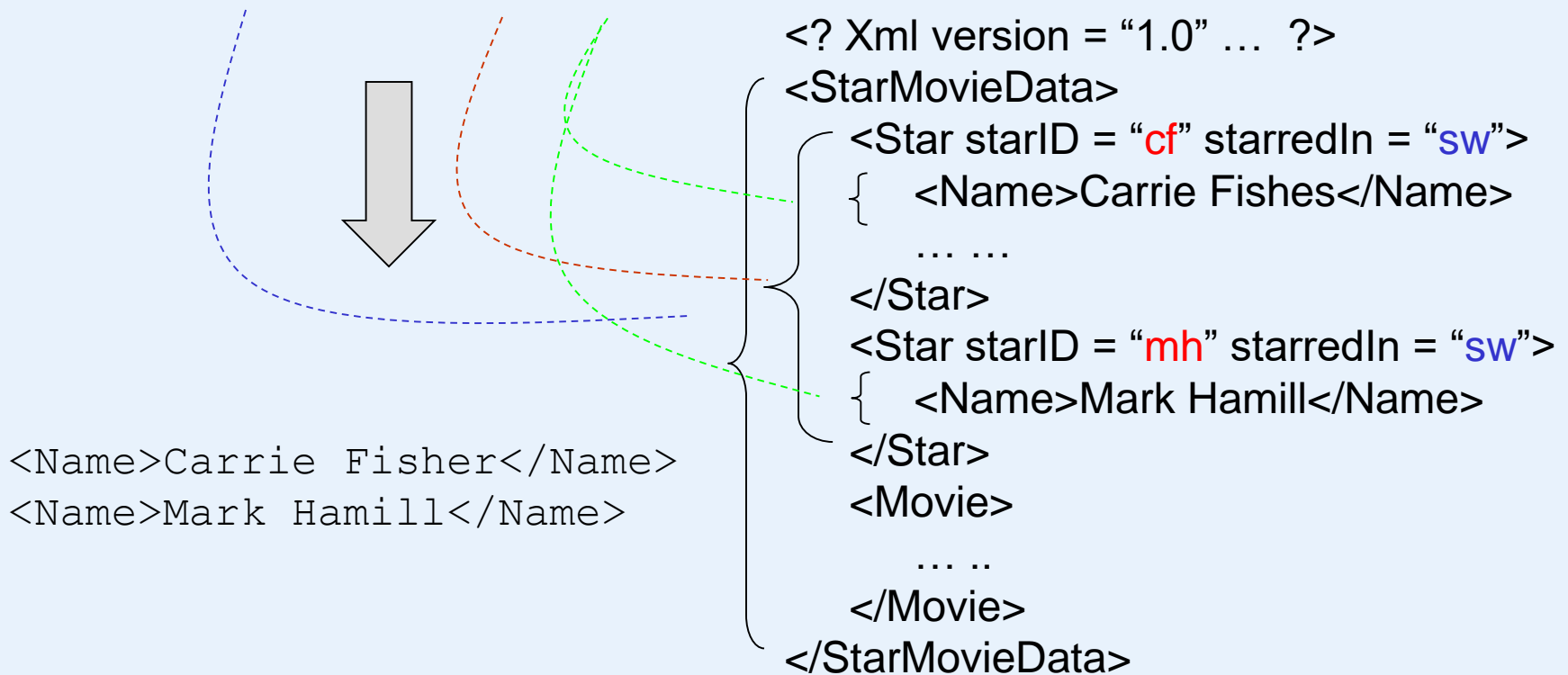
```
  </Movie>
```

```
</StarMovieData>
```

```
doc("movie.xml")/StarMoviedata/Star/Name
```

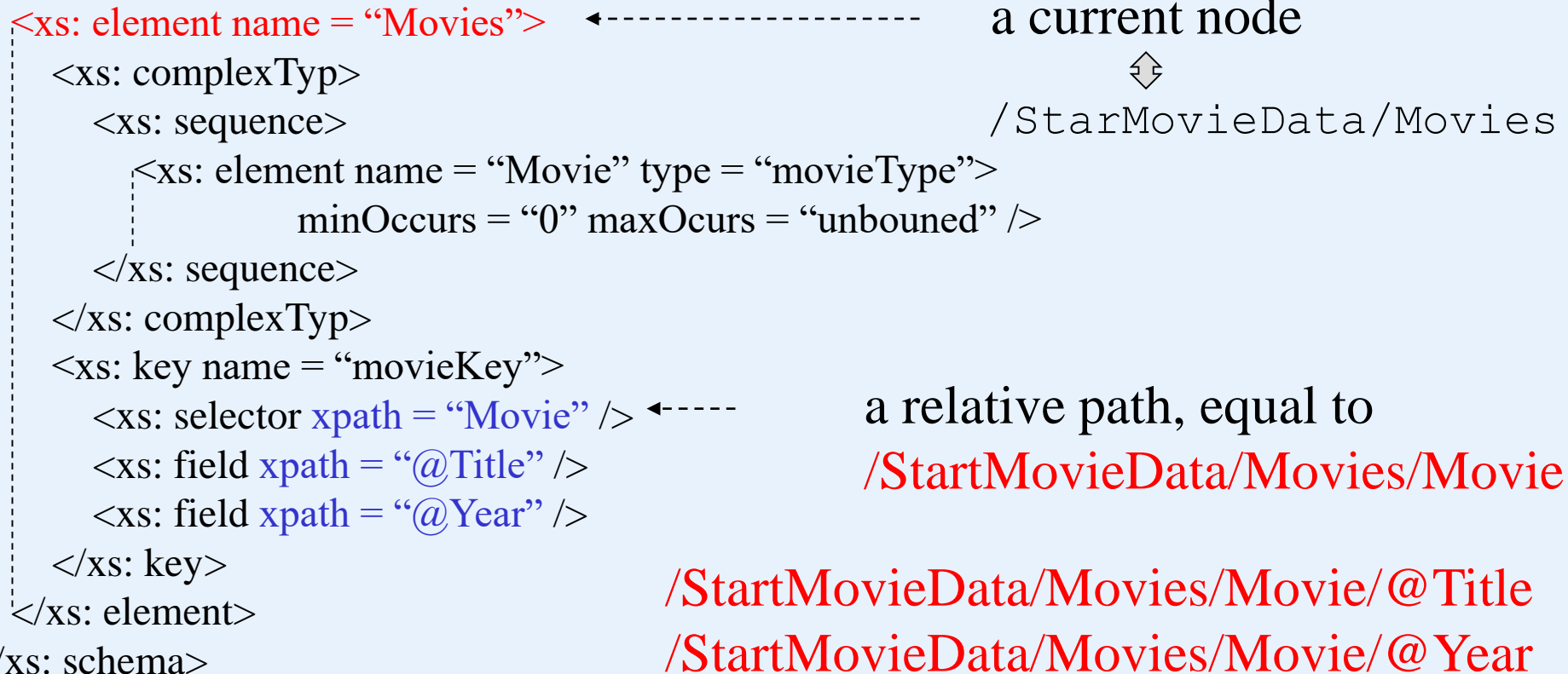
In the following discussion, the document node is not included in an XPath for simplicity.

/StarMoviedata/Star/Name



Relative Path Expressions

In several contexts, we shall use XPath expressions that are relative to the *current node* or sequence of nodes.



Attribute in Path Expressions

- Path expressions allow us to find all the elements within a document that are reached from the root along a particular path.

$$/T_1/T_2/.../T_n$$

- We can also end a path by an attribute name preceded by an *at-sign*.

$$/T_1/T_2/.../T_n/@A$$

`/StarMovieData/Star/@starID`

Axes

So far, we have only navigated through semistructured-data graphs in two ways: from a node to its children or to an attribute. In fact, XPath provides several axes to navigate a graph in different ways. Two of these axes are *child* (the default axis) and *attribute*, for which @ is really a shorthand.

Axes used in Xpath expressions: */axis::*

Self	<i>/self::</i>	<i>/next-sibling::</i>
Parent	<i>/parent::</i>	<i>/following::</i>
descendant	<i>/descendant::</i>	<i>/preceding::</i>
Ancestor	<i>/ancestor::</i>	<i>/child::</i>
Next-sibling		<i>/attribute::</i>
Following		
Preceding	<i>/child::StarMovieData/descendant::Star/attribute::starID</i>	

Axes

self	Selects the current node
parent	Selects the parent of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
next-sibling	Select the next sibling
following	Selects everything in the document after the closing tag of the current node
preceding	Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes
child	Selects all children of the current node
attribute	Selects all attributes of the current node

- All the children of the current node are referred to as siblings.
- All those nodes visited after the current node during a DFS search are referred as the following nodes.
- All those nodes visited before the current node during a DFS search are referred as the preceding nodes.

Abbreviated axes

/ - stands for *child*

@ – stands for *attribute*

. - stands for *self*

.. – stands for *parent*

// - stands for *descendant*

/child::StarMovieData/descendent::Star/attribute::starID



/StarMovieData//Star/@starID

/descendant::City



//City

/StarMovieData//Star//City

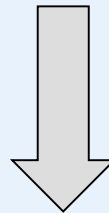
produces the same results as //City.

Context of Expression

- By “context”, we mean an element in a document, working as a reference point (current node).
- So it makes sense to apply axes like *parent*, *ancestor*, or *next-sibling* to a current node.

- Two functions: `text()`, `node()`
 - `/child::text()` – select all those children of the current node, which are text nodes
 - `/child::node()` – select the all the children of the current node, whatever their node type
 - `/self::node()` – select the current node

`/StarMovieData//Star/self::node()`



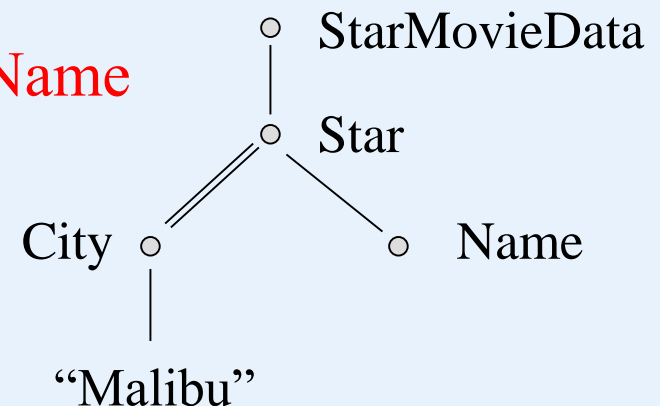
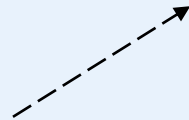
`/StarMovieData//Star`

Conditions in Path Expressions

As we evaluate a path expression, we can restrict ourselves to follow only a subset of the paths whose tags match the tags in the expression. To do so, we follow a tag by a condition, surrounded by square brackets. Such a condition can be anything that has a boolean value. Values can be compared by comparison operators: = , >=, !=. A compound condition can be constructed by connecting comparisons with logic operations: \vee , \wedge .

`/StarMovieData/Star[.//City = "Malibu"]/Name`

a simple predicate



Programming Language for XML

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
```

```
<StarMovieData>
```

```
  <Star starID = "cf" starredIn = "sw">
```

```
    <Name>Carrie Fishes</Name>
```

```
    <Address>
```

```
      <Street>123 Maple St.</Street><City>Hollywood</City>
```

```
    </Address>
```

```
    <Address>
```

```
      <Street>5 Locust Ln.</Street><City>Malibu</City>
```

```
    <Address>
```

```
      <Name>Carrie Fisher</Name>
```

```
  </Star>
```

```
  <Star starID = "mh" starredIn = "sw">
```

```
    <Name>Mark Hamill</Name><Street>456 Oak Rd.</Street>
```

```
    <City>Brentwood</City>
```

```
  </Star>
```

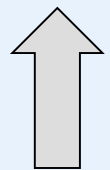
```
  <Movie movieID = "sw" starOf = "cf mh">
```

```
    <Title>Star Wars</title><Year>1977</Year>
```

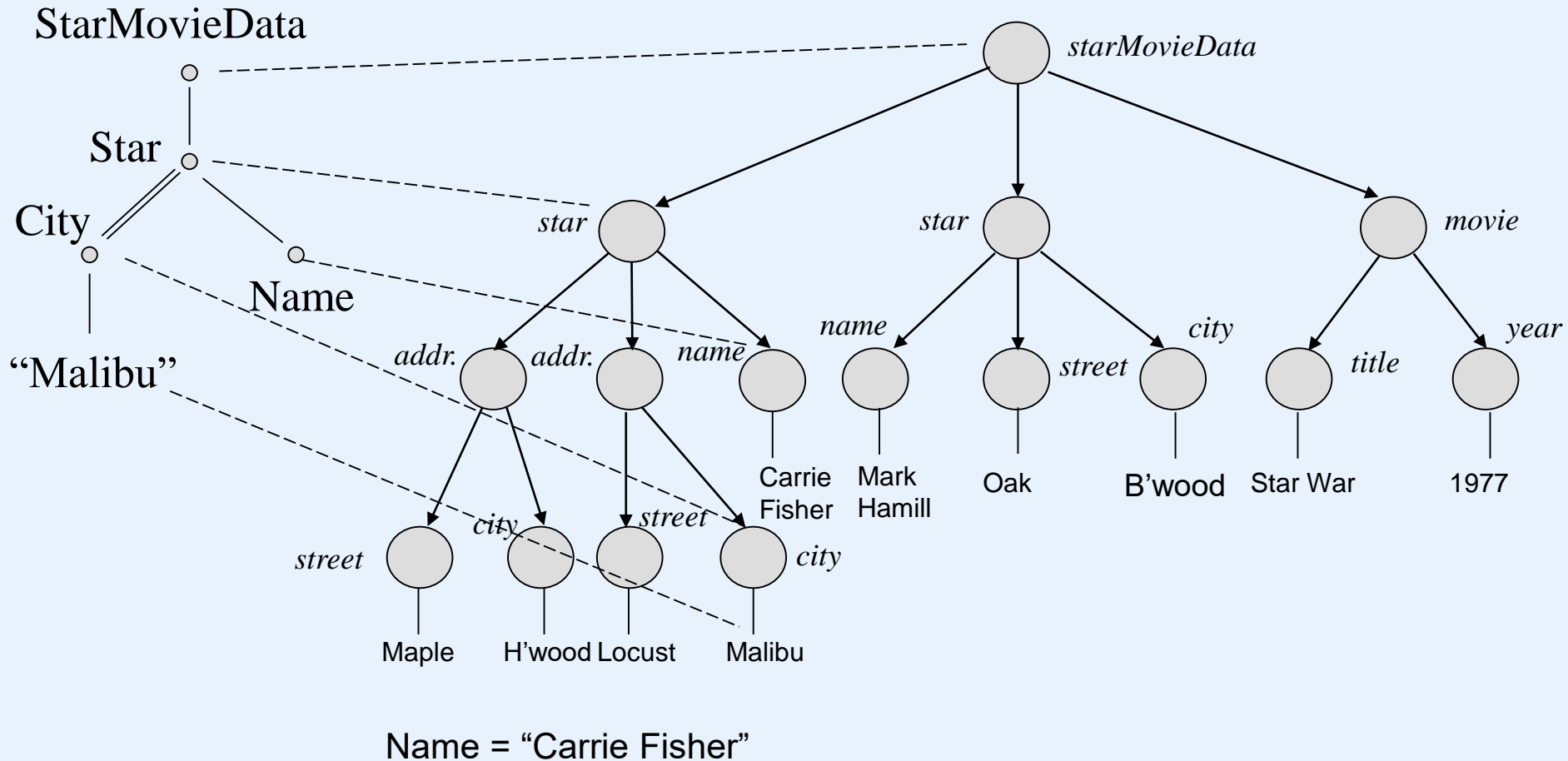
```
  </Movie>
```

```
  /StarMovieData/Star[.//City = "Malibu"]/Name
```

```
</StarMovieData>
```



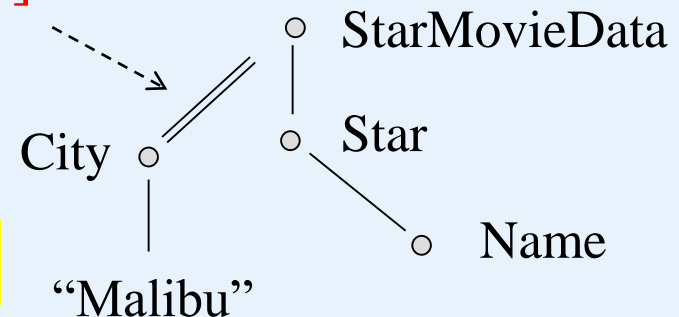
`/StarMovieData/Star[.//City = "Malibu"]/Name`



Conditions in Path Expressions

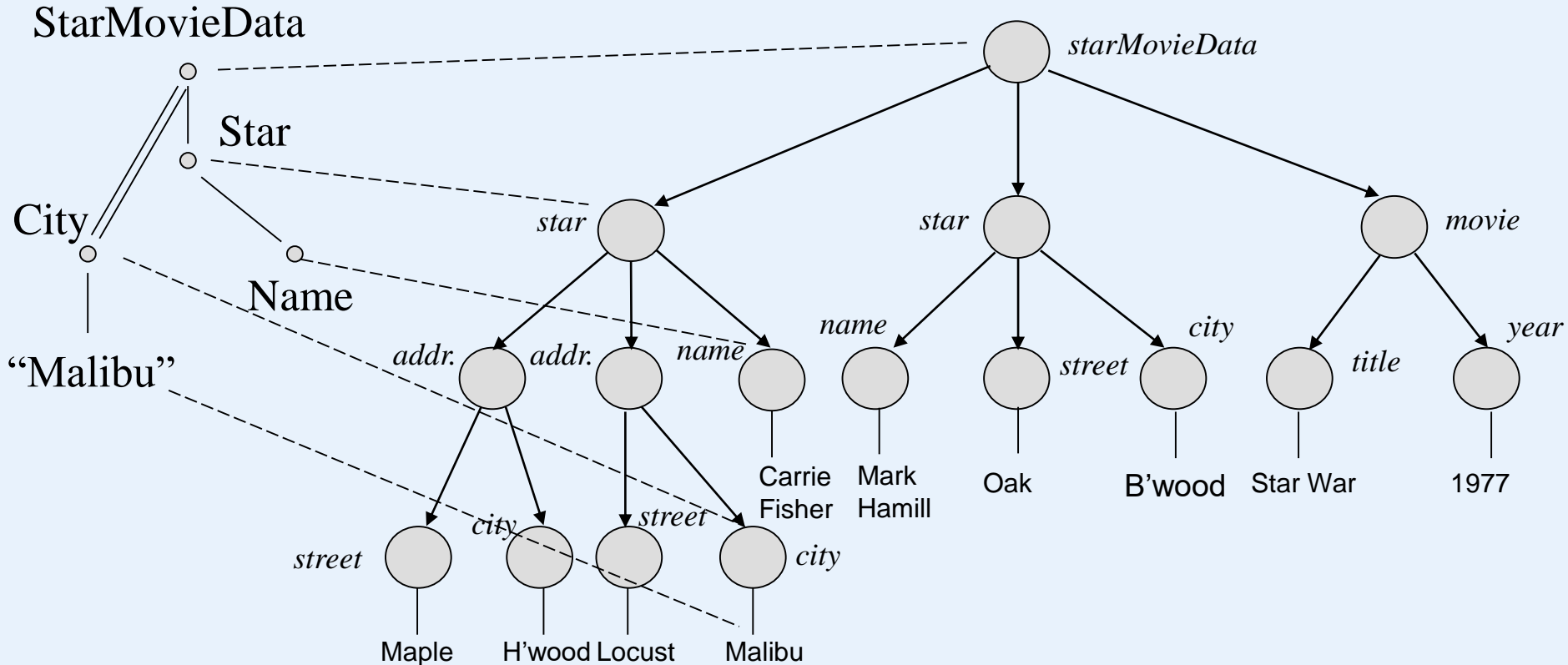
As we evaluate a path expression, we can restrict ourselves to follow only a subset of the paths whose tags match the tags in the expression. To do so, we follow a tag by a condition, surrounded by square brackets. Such a condition can be anything that has a boolean value. Values can be compared by comparison operators: = , >=, !=. A compound condition can be constructed by connecting comparisons with operations: \vee , \wedge .

`/StarMovieData/Star[../City = "Malibu"]/Name`



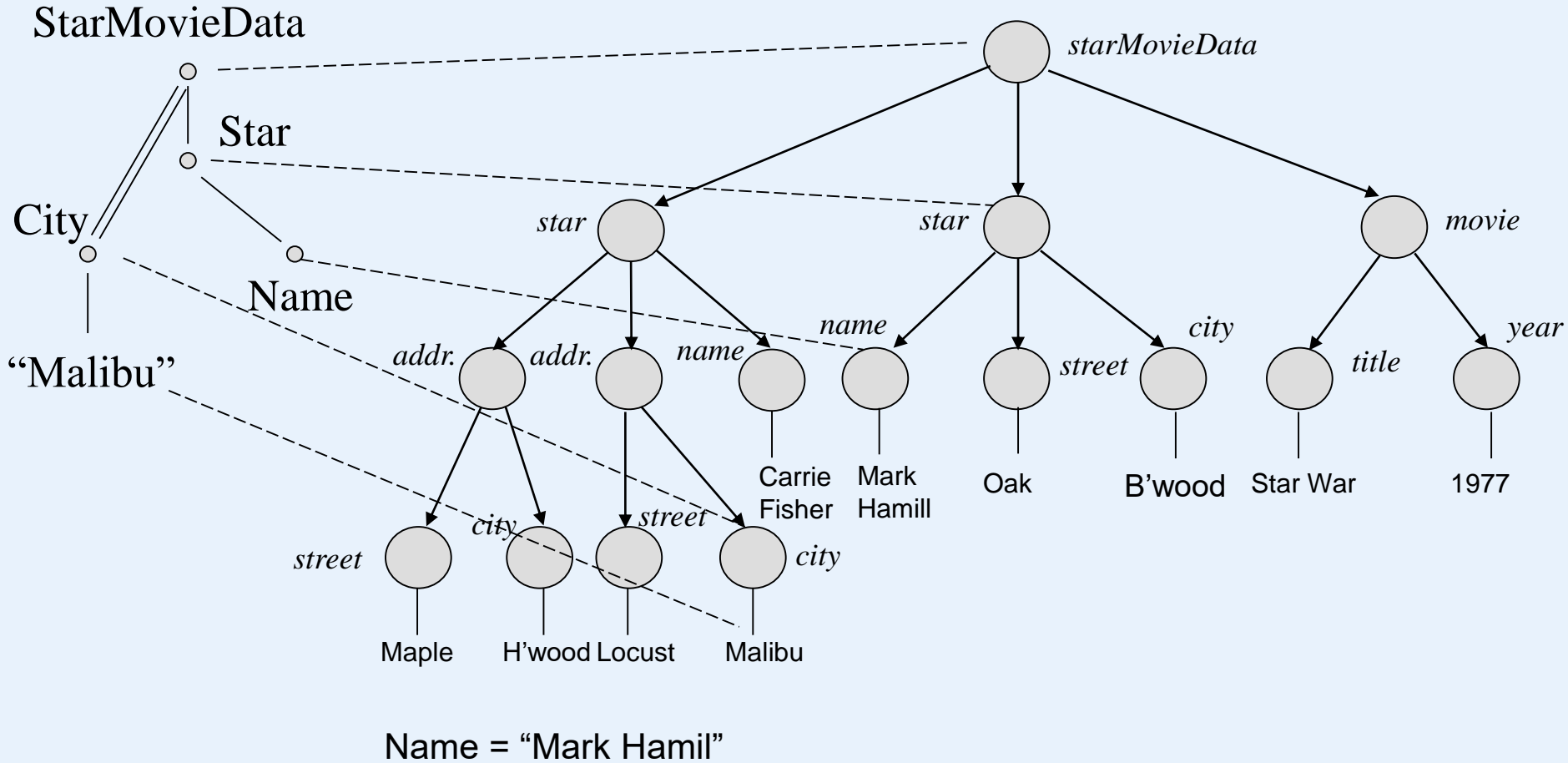
`/StarMovieData/Star[../City = "Malibu"]/Name`

/StarMovieData/Star[.//City = "Malibu"]/Name



Name = "Carrie Fisher"

`/StarMovieData/Star[../City = "Malibu"]/Name`



Conditions in Path Expressions

Several other useful forms of condition are:

- An integer [i] by itself is true only when applied the i th child of its parent.

`/StarMovieData/Stars/Star[2]`

- A tag [T] by itself is true only for elements that have one or more subelements with tag T .

`/StarMovieData/Stars/Star[Address]`

- An attribute [A] by itself is true only for elements that have an attribute A .

`/StarMovieData/Stars/Star[@startID]`

Programming Language for XML

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
```

```
<Movies>
```

```
  <Movie title = "King Kong" >
```

```
    <Version year = "1933">
```

```
      <Star>Fay Wray</Star>
```

```
    </Version>
```

```
    <Version year = "1976">
```

```
      <Star>Jeff Bridegs</Star>
```

```
      <Star>Jessica Lange</Star>
```

```
    </Version>
```

```
  </Movie>
```

```
  <Movie title = "Footloose">
```

```
    <Version year = "1984">
```

```
      <Star>Kevin Bacon</Star>
```

```
      <Star>John Lithgow</Star>
```

```
      <Star>Sarah Jessica Parkr</Star>
```

```
    </Version>
```

```
  </Movie>
```

```
</Movies>
```

← /Movies/Movie/Version[1]/@year

← /Movies/Movie/Version[Star] ?

/Movies/Movie/Version/Star?

Wildcards

In an XPath expression, we can use `*` to say “any tag”. Likewise, `@*` says “any attribute.”

```
/StarMovieData/*/@*
```

Results: “cf”, “sw”, “mh”, “sw”, “sw”, “cf mh”

```
<StarMovieData>
  <Star starID = “cf” starredIn = “sw”>
    ... ..
  </Star>
  <Star starID = “mh” starredIn = “sw”>
    ... ..
  </Star>
  <Movie movieID = “sw” starOf = “cf mh”>
    ... ..
  </Movie>
</StarMovieData>
```

The XPath expressions are mainly used in HTML, XQuery and XSLT languages.

Example:

```
<ul>
```

```
    {doc(starMovie.xml)/StarMovieData/*/@*}
```

```
</ul>
```



- Cf
- Sw
- Mh
- Sw
- Sw
- cf mh

XQuery

- XQuery is an extension of XPath that has become a standard for high-level querying of databases containing XML data.
- XQuery is designed to take data from multiple databases, from XML files, from remote Web documents, even from CGI (common gate interface) scripts, and to produce XML results that you can process with XSLT.

XQuery Basics

All values produced by XQuery expressions are sequences of items.

Items:

- primitive values

- nodes: document, element, attribute nodes

XQuery is a *functional language*, which implies that any XQuery expression can be used in any place that an expression is expected.

FLWR Expressions

FLWR (pronounced “flower”) expressions are in some sense analogous to SQL select-from-where expressions.

An XQuery expression may involve clauses of four types, called for-, let-, where-, and return-clauses (FLWR).

1. The query begins with zero or more for- and let-clauses. There can be more than one of each kind, and they can be interlaced in any order, e.g., for, for, let, for, let.
2. Then comes an optional where-clause.
3. Finally, there is exactly one return-clause.

```
Return <Greeting>“Hello World”</Greeting>
```

Let Clause

let *variable* := *expression*

- The intent of this clause is that the expression is evaluated and assigned to the variable for the remainder of the FLWR expression.
- Variables in XQuery must begin with a dollar-sign.
- More generally, a comma-separated list of assignments to variables can appear.

```
let $stars := doc("stars.xml")
```

```
let $movies := doc("movies.xml")  
    $stars := doc("stars.xml")
```

for Clause

for *variable* in *expression*

```
let $movies := doc("movies.xml")  
for $m in $movies/Movies/Movie
```

Stars.xml

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<Stars>
  <Star>
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maples St.</street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ave.</Street>
      <City>Malibu</City>
    </Address>
  </Star>
  ... more stars
</Stars>
```

Movies.xml

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
<Movies>
  <Movie title = "King Kong">
    <Version year = "1993">
      <Star>Fay Wray</Star>
    </Version>
    <Version year = "1976">
      <Star>Jeff Brideges</Star>
      <Star>Jessica Lange</Star>
    </version>
  </Movie>
  <Movie title = "Footloose">
    <Version year = "1984">
      <Star>Kevin Bacon</Star>
      <Star>John Lithgow</Star>
      <Star>Sarah Jessica Parkr</Star>
    </Version>
  </Movie>
</Movies>
```


Where Clause

where *condition*

where $\$s/\text{Address}/\text{Street} = \text{“123 Maple St.”}$ and
 $\$s/\text{Address}/\text{City} = \text{“Malibu”}$

This clause is applied to an item, and the condition, which is an expression, evaluates to true or false.

return Clause

return *expression*

This clause returns the values obtained by evaluating *expression*.

```
let $movies := doc(“movies.xml”)
for $m in $movies/Movies/Movie
return $m/Version/Star
```

```
<Star>Fay Wray</Star>
<Star>Jeff Brideges</Star>
<Star>Jessica Lange</Star>
<Star>Kevin Bacon</Star>
<Star>John Lithgow</Star>
<Star>Sarah Jessica Parker</Star>
```

Programming Language for XML

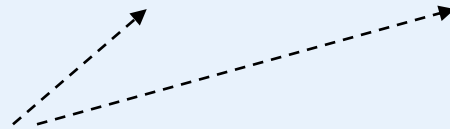
```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return $m/Version/Star
```

```
<Star>Fay Wray</Star>
<Star>Jeff Brideges</Star>
<Star>Jessica Lange</Star>
<Star>Kevin Bacon</Star>
<Star>John Lithgow</Star>
<Star>Sarah Jessica Parker</Star>
```

```
<? Xml version = "1.0" encoding = "utf-8" ... ?>
<Movies>
  <Movie title = "King Kong">
    <Version year = "1993">
      <Star>Fay Wray</Star>
    </Version>
    <Version year = "1976">
      <Star>Jeff Brideges</Star>
      <Star>Jessica Lange</Star>
    </version>
  </Movie>
  <Movie title = "Footloose">
    <Version year = "1984">
      <Star>Kevin Bacon</Star>
      <Star>John Lithgow</Star>
      <Star>Sarah Jessica Parkr</Star>
    </Version>
  </Movie>
</Movies>
```

Replacement of variables by their Values

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title = $m/@title>$m/Version/Star</Movie>
```



Not correct! The variable will not be replaced by its values.

```
<Movie title = $m/@title>$m/Version/Star</Movie>
<Movie title = $m/@title>$m/Version/Star</Movie>
<Movie title = $m/@title>$m/Version/Star</Movie>
... ..
```

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
return <Movie title = {$m/@title}>{$m/Version/Star}</Movie>
```

```
<Movie title = "King Kong"><Star>Fay Wray</Star></Movie>
<Movie title = "King Kong"><Star>Jeff Brideges</Star></Movie>
<Movie title = "King Kong"><Star>Jessica Lange</Star></Movie>
<Movie title = "Footloose"><Star>Kevin Bacon</Star></Movie>
<Movie title = "Footloose"><Star>John Lithgow</Star></Movie>
<Movie title = "Footloose"><Star>Sarah Jessica Parker</Star></Movie>
... ..
```

Joins in XQuery

We can join two or more documents in XQuery in much the same way as in SQL. In each case, we need variables, each of which ranges over elements of one of the documents or tuples of one of the relations, respectively.

1. In SQL, we use a from-clause to introduce the needed tuple variables
2. In XQuery, we use a for-clause.

```
let $movies := doc("movies.xml")
    $stars := doc("stars.xml")
for $s1 in $movies/Movies/Movie/Version/Star
    $s2 in $Stars/Stars/Star
where data($s1) = data($s2/Name)
return $s2/Address/City
```

```
Select ssn, lname, Dname
From employees s1, departments s2
Where s1.dno = s2. Dnumber
```

Programming Language for XML

```
let $movies := doc("movies.xml")
    $stars := doc("stars.xml")
for $s1 in $movies/Movies/Movie/Version/Star
    $s2 in $Stars/Stars/Star
where data($s1) = data($s2/Name)
return $s2/Address/City
```

```
<? Xml version = "1.0" .... .. ?>
<Movies>
  <Movie title = "King Kong">
    <Version year = "1993">
      <Star>Fay Wray</Star>
    </Version>
  <Version year = "1976">
    <Star>Jeff Brideges</Star>
    <Star>Jessica Lange</Star>
  </version>
</Movie>
<Movie title = "Footloose">
  <Version year = "1984">
    <Star>Kevin Bacon</Star>
    <Star>John Lithgow</Star>
    <Star>Sarah Jessica Parkr</Star>
  </Version>
</Movie>
</Movies>
```

```
<? Xml version = "1.0" encoding = "utf-8" ... ?>
<Stars>
  <Star>
    <Name>Fay Wray</Name>
    <Address>
      <Street>123 Maples St.</street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ln.</Street>
      <City>Mallibu</City>
    </Address>
  </Star>
  ... more stars
</Stars>
```

XQuery Comparison Operators

A query: find all the stars that live at 123 Maple St., Malibu.

The following FLWR seems correct. But it does not work.

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $s/Address/Street = "123 Maple St."
    and $s/Address/City = "Malibu"
return $s/Name
```

Correct query:

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star,
    $s1 in $s/Address
where $s1/Street = "123 Maple St." and
    $s1//City = "Malibu"
return $s/Name
```

```
<? Xml version = "1.0" encoding = "utf-8" ... ?>
<Stars>
  <Star>
    <Name>Fay Wray</Name>
    <Address>
      <Street>123 Maples St.</street>
      <City>Hollywood</City>
    </Address>
  <Address>
    <Street>5 Locust Ave.</Street>
    <City>Mallibu</City>
  </Address>
</Star>
  ... more stars
</Stars>
```

Elimination of Duplicates

XQuery allows us to eliminate duplicates in sequences of any kind, by applying the built-in `distinct values`.

Example. The result obtained by executing the following first query may contain duplicates. But the second not.

```
let $starsSeq := (  
  let $movies := doc("movies.xml")  
  for $m in $movies/Movies/Movie  
  return $m/Version/Star  
)  
return <Stars>{$starsSeq}</Stars>
```

```
let $starsSeq := distinct-values(  
  let $movies := doc("movies.xml")  
  for $m in $movies/Movies/Movie  
  return $m/Version/Star  
)  
return <Stars>{$starsSeq}</Stars>
```

Select average(*distinct salary*) from *employee*;

Quantification in XQuery

There are expressions that say, in effect, *for all* (\forall), and *there exists* (\exists):

every *variable* in *expression1* satisfies *expression2*
some *variable* in *expression1* satisfies *expression2*

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where every $c in $s/Address/City
      satisfies $c = "Hollywood"
return $s/Name
```

Find the stars who have houses only in Hollywood.

```
let $stars := doc("stars.xml")
for $s in $stars/Stars/Star
where $c in $s/Address/City satisfies
      $c = "Hollywood"
return $s/Name
```

Find the stars with a home in Hollywood.
(Key word *some* is not used.)

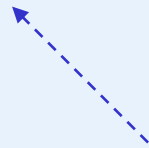
```
Select ssn, fname, salary from employee where salary  
> all (select salary from employee where dno = 4);
```

```
Select fname, lname  
from employee  
where  
    exists (select *  
            from dependent  
            where essn = ssn);
```

Aggregation

XQuery provides built-in functions to compute the usual aggregations such as count, average, sum, min, or max. They take any sequence as argument. That is, they can be applied to the result of any XPath expression.

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie
where count($m/Version) > 1
return $m
```



Find the movies with multiple versions.

```
Select s.ssn, s.lname, count(r.lname)
from employee s, employee r
where s.ssn = r.superssn
group by s.ssn, s.lname;
having count(s.name) < 3;
```

Branching in XQuery Expressions

There is an *if-then* expression in Xquery of the form:

if (*expression1*) **then** (*expression2*)

```
let $kk := doc("movies.xml")/Movies/Movie/
Movie[@title = "King Kong"]

for $v in $kk/Version
return if ($v/@year = max($kk/Version/@year))
then <Latest>{$v}</Latest>
else <Old>{$v}</Old>
```

Tag the version of *King Kong*.

```
<? Xml version = "1.0" .... . ?>
<Movies>
  <Movie title = "King Kong">
    <Version year = "1993">
      <Star>Fay Wray</Star>
    </Version>
    <Version year = "1976">
      <Star>Jeff Bridges</Star>
      <Star>Jessica Lange</Star>
    </version>
  </Movie>
  <Movie title = "Footloose">
    <Version year = "1984">
      <Star>Kevin Bacon</Star>
      <Star>John Lithgow</Star>
      <Star>Sarah Jessica
Parkr</Star>
    </Version>
  </Movie>
</Movies>
```

Movies.xml

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
```

```
<Movies>
```

```
  <Movie title = "King Kong">
```

```
    <Version year = "1993">
```

```
      <Star>Fay Wray</Star>
```

```
    </Version>
```

```
    <Version year = "1976">
```

```
      <Star>Jeff Brideges</Star>
```

```
      <Star>Jessica Lange</Star>
```

```
    </version>
```

```
  </Movie>
```

```
  <Movie title = "Footloose">
```

```
    <Version year = "1984">
```

```
      <Star>Kevin Bacon</Star>
```

```
      <Star>John Lithgow</Star>
```

```
      <Star>Sarah Jessica Parkr</Star>
```

```
    </Version>
```

```
  </Movie>
```

```
</Movies>
```

```
Let $kk :=
```

```
  doc("movies.xml")/Movies/Movie/Movie  
  [@title = "King Kong"]
```

```
For $v in $kk/Version
```

```
Return if ($v/@year =
```

```
  max($kk/Version/@year))
```

```
  then <Latest>{$v}</Latest>
```

```
  else <Old>{$v}</Old>
```



```
<Latest><Version year = "1993"> ... </Latest>
```

```
<Old><Version year = "1976"> ... </Old>
```

Ordering the Result of a Query

It is possible to sort the result as part of a FLWR query

order *list of expressions*

```
let $movies := doc("movies.xml")
for $m in $movies/Movies/Movie,
    $v in $m/Version
order $v/@year
return <Movie title = "{$m/@title}" year = "{$v/@year}" />
```

```
Select *
From employees
order by ssn
```

Construct the sequence of *title-year* pairs, ordered by *year*.

Movies.xml

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
```

```
<Movies>
```

```
  <Movie title = "King Kong">  
    <Version year = "1993">  
      <Star>Fay Wray</Star>  
    </Version>  
    <Version year = "1976">  
      <Star>Jeff Brideges</Star>  
      <Star>Jessica Lange</Star>  
    </version>
```

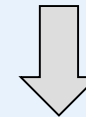
```
</Movie>
```

```
<Movie title = "Footloose">  
  <Version year = "1984">  
    <Star>Kevin Bacon</Star>  
    <Star>John Lithgow</Star>  
    <Star>Sarah Jessica Parkr</Star>  
  </Version>
```

```
</Movie>
```

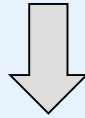
```
</Movies>
```

```
let  $movies := doc("movies.xml")  
for  $m in $movies/Movies/Movie,  
     $v in $m/Version  
order $v/@year  
return <Movie title = "{$m/@title}"  
       year = "{$v/@year}" />
```



```
<Movie title = "King Kong" year = "1976" />  
<Movie title = "Footloose" year = "1984" />  
<Movie title = "King Kong" year = "1993" />
```

```
let  $movies := doc("movies.xml")
for  $m in $movies/Movies/Movie,
     $v in $m/Version
order $m/@title, $v/@year
return <Movie title = "{$m/@title}" year = "{$v/@year}" />
```



```
<Movie title = "Footloose" year = "1984" />
<Movie title = "King Kong" year = "1976" />
<Movie title = "King Kong" year = "1993" />
```


About usage of XQuery

An XQuery expression can be embedded in an HTML file.

```
⋮  
<ul>  
  {  
    for $x in doc("books.xml")/bookstore/book/title  
    order by $x  
    return <li>{$x}</li>  
  }  
</ul>  
⋮
```

Extensible Stylesheet Language

XSLT (Extensible Stylesheet Language for Transformation) is a standard of the World-Wide-Web Consortium.

- Its original purpose was to allow XML documents to be transformed into HTML or similar forms that allowed the document to be viewed or printed.
- In practice, XSLT is another query language for XML to extract data from documents or turn one document form into another form.

XSLT Basics

Like XML schema, XSLT specifications are XML documents, called *stylesheet*. The tag used in XSLT are found in a name-space: <http://www.w3.org/1999/XSL/Transform>.

At the highest level, a stylesheet looks like:

```
<? Xml version = '1.0' encoding = "utf-8" ?>  
<xsl:stylesheet xmlns:xsl =  
    http://www.w3.org/1999/XSL/Transform>  
    ... ..  
</xsl:stylesheet>
```

Templates

A stylesheet will have one or more templates. To apply a stylesheet to an XML document, we go down the list of templates until we find one that matches the root.

```
<xsl:template match = "XPath expression">
```

Templates

```
<xsl:template match = "XPath expression">
```

XPath expression can be either rooted (beginning with a slash) or relative. It describes the elements of XML documents to which this template is applied.

Rooted expression – the template is applied to every element of the document that matches the path (absolute path).

Relative expression – part of an Xpath, evaluated relative to a reference point (the current node).

```
<? Xml version = "1.0" encoding = "utf-8" ?>
<xsl:stylesheet xmlns:xsl =
    http://www.w3.org/1999/XSL/Transform>
  <xsl:template match = "/">
    <HTML>
      <BODY>
        <B>This is a document</B>
      </BODY>
    </HTML>
  </xsl:template >
</xsl:stylesheet>
```

Applying the template, an XML document is transformed to a HTML file:

```
<HTML>
  <BODY>
    <B>This is a document</B>
  </BODY>
</HTML>
```

Obtaining Values from XML Data

```
<xsl:value-of select = "expression" />
```

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
```

```
<Movies>
```

```
<Movie title = "King Kong">  
  <Version year = "1993">  
    <Star>Fay Wray</Star>  
  </Version>  
  <Version year = "1976">  
    <Star>Jeff Brideges</Star>  
    <Star>Jessica Lange</Star>  
  </Version year = "2005" />  
</Movie>
```

```
<Movie title = "Footloose">  
  <Version year = "1984">  
    <Star>Kevin Bacon</Star>  
    <Star>John Lithgow</Star>  
    <Star>Sarah Jessica Parkr</Star>  
  </Version>  
</Movie>
```

```
</Movies>
```

```
<? Xml version = "1.0" encoding = "utf-8" ?>  
<xsl:stylesheet xmlns:xsl =  
  http://www.w3.org/1999/XSL/Transform>  
  <xsl:template match = "/Movies/Movie">  
    <xsl:value-of select = "@title" />  
    <BR/>  
  </xsl:template >  
</xsl:stylesheet>
```

"King Kong"

"Footloose"

This ability makes XSTL a query language.

Recursive Use of Templates

Powerful transformations require recursive application of templates at various elements of the input.

```
<xsl:apply-template select = "expression" />
```

Programming Language for XML

```
<? Xml version = "1.0" encoding = "utf-8" ?>
<xsl:stylesheet xmlns:xsl =
http://www.w3.org/1999/XSL/Transform>

  <xsl:template match = "/Movies">
    <Movies>
      <xsl:apply-templates />
    </Movies>
  </xsl:template >
  <xsl:template match = "Movie">
    <Movie title = "<xsl:value-of select = "@title" />">
      <xsl:apply-templates />
    </Movie>
  </xsl:template>
  <xsl:template match = "Version">
    <xsl:apply-template />
  </xsl:template>
  <xsl:template match = "Star">
    <Star name = "<xsl:value-of select = "." />" />
  </xsl:template>
</xsl:stylesheet>
```

use this template

use this template

use this template

```
<? Xml version = "1.0" encoding = "utf-8"
  standalone = "yes" ?>
<Movies>
  <Movie title = "King Kong">
    <Version year = "1993">
      <Star>Fay Wray</Star>
    </Version>
    <Version year = "1976">
      <Star>Jeff Bridges</Star>
      <Star>Jessica Lange</Star>
    </version>
  </Movie>
  <Movie title = "Footloose">
    <Version year = "1984">
      <Star>Kevin Bacon</Star>
      <Star>John Lithgow</Star>
      <Star>Sarah Jessica Park
    </Star>
    </Version>
  </Movie>
</Movies>
```


Programming Language for XML

```
<? Xml version = "1.0" encoding = "utf-8"
  standalone = "yes" ?>
<Movies>
  <Movie title = "King Kong">
    <Version year = "1993">
      <Star>Fay Wray</Star>
    </Version>
    <Version year = "1976">
      <Star>Jeff Bridges</Star>
      <Star>Jessica Lange</Star>
    </version>
  </Movie>
  <Movie title = "Footloose">
    <Version year = "1984">
      <Star>Kevin Bacon</Star>
      <Star>John Lithgow</Star>
      <Star>Sarah Jessica Park</Star>
    </Version>
  </Movie>
</Movies>
```

```
<? Xml version = "1.0" encoding = "utf-8"
  standalone = "yes" ?>
<Movies>
  <Movie title = "King Kong">
    <Star name = "Fay Wray" />
    <Star name = "Jeff Brideges" />
    <Star name = "Jessica Lange" />
  </Movie>
  <Movie title = "Footloose">
    <Star name = "Kevin Bacon" />
    <Star name = "John Lithgow" />
    <Star name = "Sarah Jessica Parkr" />
  </Movie>
</Movies>
```

Iteration in XSLT

We can put a loop within a template that gives us freedom over the order in which we visit certain subelements of the element to which the template is being applied.

```
<xsl:for-each select = "expression" >
```

The expression is an XPath expression whose value is a sequence of items. Whatever is between the opening <for-each> tag and its matching closing tag is executed for each item, in turn.

Programming Language for XML

```
<? Xml version = "1.0" encoding = "utf-8" standalone = "yes" ?>
```

```
<Stars>
```

```
<Star>
```

```
<Name>Carrie Fisher</Name>
```

```
<Address>
```

```
<Street>123 Maples St.</street>
```

```
<City>Hollywood</City>
```

```
</Address>
```

```
<Address>
```

```
<Street>5 Locust Ln.</Street>
```

```
<City>Malibu</City>
```

```
</Address>
```

```
</Star>
```

... *more stars*

```
</Stars>
```

1. Carrie Fishes

2. Mark Hamill

... ..

1. Hollywood

2. Malibu

... ..

```
<? Xml version = "1.0" encoding = "utf-8" ?>
```

```
<xsl:stylesheet xmlns:xsl =
```

```
http://www.w3.org/1999/XSL/Transform >
```

```
<xsl:template match = "/">
```

```
<OL>
```

```
<xsl:for-each select = "Stars/Star" >
```

```
<LI>
```

```
<xsl:value-of select = "Name">
```

```
</LI>
```

```
</xsl:for-each>
```

```
</OL><P />
```

```
<OL>
```

```
<xsl:for-each select =
```

```
"Stars/Star/Address">
```

```
<LI>
```

```
<xsl:value-of select = "City">
```

```
</LI>
```

```
</xsl:for-each>
```

```
</OL>
```

```
</xsl:template >
```

```
</xsl:stylesheet>
```

Programming Language for XML

```

<Stars>
  <Star>
    <Name>Carrie Fisher</Name>
    <Address>
      <Street>123 Maples
St.</street>
      <City>Hollywood</City>
    </Address>
    <Address>
      <Street>5 Locust Ln.</Street>
      <City>Malibu</City>
    </Address>
  </Star>
  ... more stars
</Stars>

```

```

<OL>
  <LI>
    Carrie Fisher
  </LI>
  <LI>
    Mark Hamill
  </LI>
  ... more stars
</OL><P/>
<OL>
  <LI>
    Hollywood
  </LI>
  <LI>
    Malibu
  </LI>
  ... more cities
</OL>

```

```

<? Xml version = "1.0" encoding = "utf-8" ?>
<xsl:stylesheet xmlns:xsl =
http://www.w3.org/1999/XSL/Transform>
  <xsl:template match = "/">
    <OL>
      <xsl:for-each select =
        "Stars/Star" >
        <LI>
          <xsl:value-of select =
            "Name">
          </LI>
        </xsl:for-each>
      </OL><P/>
      <OL>
        <xsl:for-each select =
          "Stars/Star/Address">
          <LI>
            <xsl:value-of select =
              "City">
            </LI>
          </xsl:for-each>
        </OL>
      </xsl:template >
    </xsl:stylesheet>

```

1. Carrie Fishes
2. Mark Hamill
-
1. Hollywood
2. Malibu
-

Conditions in XSLT

We can introduce branching into our templates by using an if tag.

```
<xsl:if test = "boolean expression" >
```

Whatever appears between its tag and its matched closing tag is executed if and only if the boolean expression is *true*.

```
<? Xml version = "1.0" encoding = "utf-8" ?>
<xsl:stylesheet xmlns:xsl =
http://www.w3.org/1999/XSL/Transform>
  <xsl:template match = "/">
    <TABLE border = "5"><TR><TH>Stars</TH><TR>
      <xsl:for-each select = "Stars/Star" >
        <xsl:if test = "Address/City = 'Hollywood'">
          <TR><TD><xsl:value-of select = "Name"</TD>
        </TR>
        </xsl:if>
      </xsl:for-each>
    </TABLE>
  </xsl:template >
</xsl:stylesheet>
```

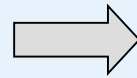
Stars
Carrie Fishes
⋮

```
<TABLE border = "5">
  <TR><TH>Stars</TH><TR>
    <TR>
      <TD>
        Carrie Fishes
      </TD>
    </TR>
    <TR>
      <TD>
        .....
      </TD>
    </TR>
    .....
  </TR>
</TABLE>
```

←----- List all those stars
who have a house
in Hollywood.

Stars
Carrie Fishes
⋮

```
<html>
<body>
  <table border="1">
    <tr>
      <th>Month</th>
      <th>Savings</th>
    </tr>
    <tr>
      <td>January</td>
      <td>$100</td>
    </tr>
  </table>
</body>
</html>
```



Month	Savings
January	\$100

How to use XSTL to make document transformation?

In this example, creating the XML file that contains the information about three students and displaying the XML file using XSLT.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<?xml-stylesheet type = "text/xsl "href = "transform.xsl" ?>
<Student>
  <s>
    <name> David John</name><branch> CSE</branch>
    <age> 23</age><city> Malibu</city>
  </s>
  <s>
    <name> Mary Chen</name><branch> CSE</branch>
    <age> 17</age><city> New York</city>
  </s>
  <s>
    <name> Christ Henry</name><branch> IT</branch>
    <age> 25</age> <city> Washington</city>
  </s>
</student>
```

students.xml


```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
<html> <body>
  <h1 align="center">Students' Basic Details</h1>
  <table border="3" align="center" >
    <tr>
      <th>Name</th>
      <th>Branch</th>
      <th>Age</th>
      <th>City</th>
    </tr>
    <xsl:for-each select="student/s">
      <tr>
        <td><xsl:value-of select="name"/></td>
        <td><xsl:value-of select="branch"/></td>
        <td><xsl:value-of select="age"/></td>
        <td><xsl:value-of select="city"/></td>
      </tr>
    </xsl:for-each>
  </table> </body> </html> </xsl:template> </xsl:stylesheet>
```

transform.xsl

Name	Branch	Age	City
David John	CSE	23	Malibu
Mary Chen	CSE	21	New York
Christ Henry	CSE	22	Washington

How to use XSTL to make document transformation?

```
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
```

(in Java)

```
public class Main {
    public static void main(String args[]) throws Exception {

        StreamSource source = new StreamSource(args[0]);
        StreamSource stylesource = new StreamSource(args[1]);

        TransformerFactory factory = TransformerFactory.newInstance();
        Transformer transformer = factory.newTransformer(stylesource);

        StreamResult result = new StreamResult(System.out);
        transformer.transform(source, result);
    }
}
```

Two inputs: source style-sheet

How to use XSTL to make document transformation?

```
XsltTransform xslTran = new XsltTransform();  
xslTran.Load("transform.xsl"); ←----- an XSTL sheet  
XmlTextWriter writer = new XmlTextWriter("xslt_output.html",  
System.Text.Encoding.UTF8); create a file to store the output  
xslTran.Transform(students.xml, null, writer);
```

↑
a file containing an XML document to be transformed