Introduction

- A key goal of bioinformatics is to create database systems and software platforms capable of storing and analyzing large sets of biological data.
- Hundreds of biological databases are now available and provide access to a diverse set of biological data.
- The exponential growth of biological data sets requires methods for data representation, storage, and exchange.
- In the past few years, many in the bioinformatics community have turned to XML to address the pressing needs associated with biological data.
XML - EXtensible Markup Language

- XML was designed to store and transport data.
- XML was designed to be both human- and machine-readable.
- XML was designed to be operating system and programming language independent.
- Since its introduction, XML has been successfully used to represent a growing set of biological data, including nucleotide sequences, protein–protein interactions, etc.
XML Structure - Tags and Attributes

- An XML file is structured by several XML-elements, also called XML-nodes or XML-tags. XML-elements' names are enclosed by triangular brackets < >, for example:
  `<dna-seq> tagggtaaagt... </dna-seq>`
- An attribute specifies a single property for the element, using a name/value pair, for example:
  `<dna-seq length="24157"> tagggtaaagt... </dna-seq>`
XML Syntax

- XML documents must contain one root element that is the parent of all other elements.
- All XML elements must have a closing tag.
- XML elements must be properly nested.
- XML attribute values must be quoted.
- XML document may contain a prolog (must be the first line).

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- SARS coronavirus Urbani, complete genome. -->
<!-- Accession Number: AY278741 -->
<Bsm1>
  <Definitions>
    <Sequences>
      <Sequence id="AY278741" length="29727">
        <Seq-data>
          atatagtatttacctaccagggatgcccccaacctcgatctcttgtagatctgtctgctatgaaaggaaacgatcagtaaacagcagccttttttctggcttttgctgactgtgcgtatagctgtctgccctggttctctctacagcaagtttacagctgtctgtctgtctgtctgtctgctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctgtctg...
Document Type Definitions (DTDs) describe XML document structures, they contain specific rules, which constrain the content of XML documents. DTD specify the legal elements that can appear in the document and their attributes. XML documents can be checked against their DTD. A document that complies with its DTD is called valid.
There are many XML formats in use in bioinformatics:

- AGAVE
- BioXSD
- BEAST
- BIopolymer Markup Language (BIOML)
- Biological Pathways Exchange (BioPAX)
- **Bioinformatic Sequence Markup Language (BSML)**
- Chado-XML
- DAS XML
- Genome Annotation Markup Elements (GAME)
- Gene Expression Markup Language (GEML)
- Helmholtz Open BioInformatics Technology network (HOBIT)
- HSAML
- KEGG Markup Language (KGML)
- **Systems Biology Markup Language (SBML)**
- Metadata Object Description Schema (MODS)
- OBO XML

*and many others..*
<!DOCTYPE bookstore[

  <!ELEMENT bookstore (book*)>
  <!ELEMENT book (title, author+, year, price)>
  <!ELEMENT title (PCDATA)>
  <!ELEMENT author (PCDATA)>
  <!ELEMENT year (PCDATA)>
  <!ELEMENT price (PCDATA)>

  <!ATTLIST book cat CDATA #REQUIRED>
]>
XML as a tree

<?xml version="1.0" encoding="UTF-8"?>

<bookstore>
  <book cat="COOKING">
    <title>Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book cat="CHILDREN">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>25.99</price>
  </book>
</bookstore>
XPath (XML Path Language)
Introduction

- XPath is used to navigate through elements and attributes in an XML document.
- XPath uses path expressions to select nodes or node-sets in an XML document. These path expressions look very much like the path expressions in a traditional computer file system.
- XPath takes advantage of the tree structure of the XML files.
Basic Syntax

‘/’ is used to search all the sons of the context node.

/\bookstore\/book\title

The root is the context node
Basic Syntax (cont.)

‘//’ is used to search all the descendants of the context node.

//book/price
Basic Syntax (cont.)

text() is used to select text nodes

//book/price/text()
Basic Syntax (cont.)

‘@’ is used to select attribute nodes

/bookstore/book/@cat
Wildcards

We can use wildcards to select unknown nodes:

- ‘*’ matches any element node.
- ‘@*’ matches any attribute node.
- `text()` matches any text node.
- `node()` matches any node of any kind.
Wildcards

//book/*
//book/@*
//book/text()
//book/node()
Predicates

Using predicates we can select nodes that fulfill a certain condition. Predicates are written in square brackets.

Predicates can contain:

- **Arithmetic operators**: +, -, *, div, mod
- **Comparison operators**: =, !=, >, <, >=, <=
- **Boolean operators**: and, or, not
- **Functions**: position(), last(), etc.
Predicates

- //book[price > 28.00]
- //book[position() <= 3]/title
- //book[@cat='COOKING' and year>2000]
- //book[position() = last()] (can be written as //book[last()])
Axes

- We can control the direction in which we perform the search.
- If no axis is specified, the default is `child` which searches in the child of the context node:
  `/bookstore/book/title`
  is actually:
  `/child::bookstore/child::book/child::title`
- A general syntax for XPath expression is:
  `/axisname::nodetest[predicate]/axisname::nodetest[predicate]/...`
Axes

The supported axes are:

- **ancestor** - search in all ancestors of the current node.
- **ancestor-or-self** - search in all ancestors of the current node and the current node.
- **attribute** - search in all attribute nodes of the current node. 
  
  //book/@cat is the same is //book/attribute::cat

- **child** - search in the children of the current node (the default axis)
- **descendant** - search in all the descendants of the current node
Axes (cont.)

- **descendant-or-self** - search in all the descendants of the current node and the current node.
  
  `/bookstore//title` is actually `/bookstore/descendant-or-self::title`

- **following-sibling** - search in all siblings of the current node that appear after the current node

- **preceding-sibling** - search in all sibling of the current node that appear before the current node

- **parent** - search in the parent of the current node

- **self** - search in the current node
Axes (cont.)

/bookstore/book[title='Everyday Italian']/following-sibling::book/attribute::cat

//book/descendant::text()

//book/descendant::*
Functions - beyond node selection

XPath supports a wide variety of functions, some of them are:

- **Functions on numeric values:**
  abs, floor, round, etc.
- **Functions on string values:**
  concat, substring, upper-case, lower-case, etc.
- **Casting functions:**
  number, string, etc.

will return a new string node with the value “Everyday ItalianHarry Potter”
XQuery (XML Query Language)
Introduction

- XQuery is a (functional) programming language for querying and manipulating XML data.
- XQuery is to XML what SQL is to relational databases.
- XQuery is designed to query XML data - not just XML files.
- XQuery is built on XPath expressions (it is actually a superset of XPath).
Basics

- XQuery provides the `doc()` function for opening XML and returning their root element.
- We can then use regular XPath expression on the root element.

- For example:
  
  ```
  doc('bookstore.xml')//books
  ```
  
  will return all the book elements in the XML file ‘bookstore.xml’.
Sequences

- XQuery uses sequences instead of sets of nodes
- Sequences are ordered list of items. Items can be either XML nodes or atomic values.
- Sequences construction: \((\text{arg1}, \text{arg2}, \text{arg3}, \ldots)\)
- A sequence can be empty.
- There is no difference between a single item and a singleton sequence: \(5 = (5)\)
- Sequences can’t be nested: \((1, 2, (3, 4, (5)), 6) = (1, 2, 3, 4, 5, 6)\)
Node Construction

- XQuery allows us to create new XML nodes.
- Using queries and node constructions we can transform one XML document to another.
- Basic syntax: `<new_node>some content</new_node>`

- We will usually want to create new nodes dynamically based on existing data.
- Using curly brackets we can insert any XPath/XQuery expression: `<new_node>{XPath/XQuery expression}</new_node>`
# Node Construction Examples

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;titles&gt;{doc(books.xml)//book/title}&lt;/titles&gt;</code></td>
<td><code>&lt;titles&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;title&gt;Everyday Italian&lt;/title&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;title&gt;Harry Potter&lt;/title&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/titles&gt;</code></td>
</tr>
<tr>
<td><code>&lt;authors&gt;{doc(books.xml)//book/author}&lt;/authors&gt;,&lt;years&gt;{doc(books.xml)//book/year}&lt;/years&gt;</code></td>
<td><code>&lt;authors&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;author&gt;Giada De Laurentiis&lt;/author&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;author&gt;J K. Rowling&lt;/author&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/authors&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;years&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;year&gt;2005&lt;/year&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;year&gt;2005&lt;/year&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/years&gt;</code></td>
</tr>
</tbody>
</table>
Advanced Node Construction

● A new document can be created using:
  \texttt{document \{ document\_content \}}

● We can also create new nodes using:
  \texttt{element \{tag\_expr\} \{content\_expr\}}
  The element’s tag will be the result of evaluating \texttt{tag\_expr} and its content will be the result of evaluating \texttt{content\_expr}

● We can create attributes using:
  \texttt{attribute \{attribute\_expr\} \{attribute\_expr\}}
  The attribute’s name will be the result of evaluating \texttt{attribute\_expr} and its value will be the result of evaluating \texttt{attribute\_expr}. 
## Advanced Construction Examples

<table>
<thead>
<tr>
<th>Query</th>
<th>Result</th>
</tr>
</thead>
</table>
| `document{ (element {doc(books.xml)//book[1]/@cat} {}, element {doc(books.xml)//book[2]/@cat} {}) }` | `<?xml version="1.0" encoding="UTF-8"?>
<COOKING></COOKING>
<CHILDREN></CHILDREN>` |
| `document{ element book { (attribute title {doc(books.xml)//book[1]/title}, attribute year {doc(books.xml)//book[1]/year}) } }` | `<?xml version="1.0" encoding="UTF-8"?>
<book title="Everyday Italian" year="2005"></book>` |
FLWOR Expression

- XQuery provides a convenient way for querying xml data in the form of FLWOR expression, which are similar to SQL queries.
- FLWOR expression consists of 5 parts:
  - For..
  - Let..
  - Where..
  - Order By..
  - Return..
- The Return part must appear in the expression as every expression in XQuery must have a value.
Basic For-Return Expression

- A basic FLWOR expression consists of For and Return. It is used to loop through a sequence of nodes/values.
- Example:

  ```
  for $i in (1, 2, 3) return <i>{$i}</i>
  <i>1</i><i>2</i><i>3</i>
  
  for $i in doc("books.xml")//book/title return <book>
  {$i/text(), "-", $i/parent::book/year/text()}
  </book>
  ```
Let

- We can use Let in a FLWOR expression to define variables.
- Syntax: \texttt{let \$varname := \{expr\}}
  The result of \texttt{expr} evaluation will be assigned to the variable \$varname.
- Example:

  \begin{verbatim}
  let $i := (1, 2, 3) return <i>{$i}</i>
  \end{verbatim}

  \begin{verbatim}
  <i>1 2 3</i>
  \end{verbatim}

- We can have multiple For and Let statements.
Where

- We can use Where in a FLWOR expression to filter results based on a condition.
- Syntax: `where cond`
- Example:

```xml
for $i in (1, 2, 3, 4)
for $j in (1, 2, 3, 4)
where $i mod $j = 0
return <tuple>{($i, $j)}</tuple>
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;tuple&gt;1 1&lt;/tuple&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;tuple&gt;2 1&lt;/tuple&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;tuple&gt;2 2&lt;/tuple&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;tuple&gt;3 1&lt;/tuple&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;tuple&gt;3 3&lt;/tuple&gt;</td>
</tr>
</tbody>
</table>
Order by

- Each XQuery expression returns a sequence.
- We can use Order By in FLWOR expression to control the order of the sequence.
- Example:

```
for $b in doc("books.xml")//book order by $b/price descending return $b
```

| Returns all the <book> elements ordered by their price in a descending order. | Returns all the <book> elements ordered by their price in a descending order. |
## FLWOR Examples

<table>
<thead>
<tr>
<th>FLWOR Example</th>
<th>Description</th>
</tr>
</thead>
</table>
| for $b in doc("books.xml")//book let $c := $b/author return <book>  
  <title>{$b/title}</title>  
  <authors>{count($c)}</authors>  
  </book> | For each book, its title and number of authors |
| for $b in doc("books.xml")//book where $b/@year = "2000" order by $b/author return $b | All books published in 2000 ordered by the author's name. |
### FLWOR Examples

<table>
<thead>
<tr>
<th>FLWOR Expression</th>
<th>Description</th>
</tr>
</thead>
</table>
Summary

- XML is a convenient way for storing and analyzing biological data.
- Using DTD’s, a variety of XML formats were developed, each of them is designed to model specific biological data.
- XPath is a convenient way for searching the XML tree.
- XQuery expands XPath and provides a query language for querying XML data in a SQL-like syntax.

What’s next? XSLT!
Questions?
References

- XML for Bioinformatics by Ethan Cerami
- http://www.w3schools.com/xpath/
- http://www.w3schools.com/xquery/
- Slides of the course Database Systems 236363 in the technion
  http://webcourse.cs.technion.ac.il/236363
- http://www.ebi.ac.uk/Tools/webservices/tutorials/aa_xml_formats