

CSE 131A
“Compiler Construction I”

Lecture 2
Onyx

Today's lecture

- Onyx
- A closer look at FLWR expressions
- OXML document creation

Announcements

- A0 has been posted
- Lab hours have been posted
<http://ieng6.ucsd.edu/~cs131w/OH.txt>
- My office hours:
 - Mondays 2:30 to 4:30 pm
 - Wednesdays 10:00 to 11 am

Declarations

- In languages like C and Java, a variable's type is established in a type declaration, and can be determined at compile time

```
int i = 3;
```

- In Onyx, type declarations may be used in variable or function declarations, but not in `let` or `for` expressions

```
declare function f($x as onyx.types.Integer) as  
    onyx.types.String  
    {string($x) + "string"};
```

- A lot of variable typing is done dynamically

Function definitions

- A function definition in Onyx looks like:
declare function select(\$t as onyx.types.Boolean,
 \$a as onyx.types.Integer, \$b as onyx.types.Integer)
 as onyx.types.Integer
 { if (\$t) then \$a else \$b };
- The function name is **f**
- It *signature* is its return type + the types of its arguments
 - its return type: `onyx.types.Integer`
 - the types of the formal parameters `$t, $a, $b`
`onyx.types.Boolean, onyx.types.Integer, onyx.types.Integer`

Type matching of function signatures

- When we call a function that isn't found, Onyx tries to help locate a possible candidate
- A possible prototype is one where the called function name is either an exact match or a prefix of the function name of the entry

true + 1

```
<onyx.error.SemanticError>  
  <DynamicError column="6" line="1">  
    <ErrorMessage>Function with prototype op:numeric- add(onyx.types.Boolean,onyx.types.Integer) not  
found.</ErrorMessage>  
    <PossibleMatch>onyx.types.Integer op:numeric-add(onyx.types.Integer,onyx.types.Integer) </PossibleMatch>  
    <PossibleMatch>onyx.types.String op:numeric-add(onyx.types.String,onyx.types.String) </PossibleMatch>  
  </DynamicError>  
</onyx.error.SemanticError>
```

The role of binding

- A binding establishes a connection between an identifier and a value

```
let $i := 3
```

- This binds `$i` to the value `3` of type `Integer`
- C & Java bind storage locations to variable names
- Assignment modifies the bound storage location

```
int i = 3;
```

- In Onyx, a variable is just a synonym for a value, and there is no updatable storage

Global variables

- Onyx supports global variable definitions
What is the result?

```
declare variable $z as onyx.types.Integer { 3 };  
declare function g($a,$b) { $a + $z + $b };  
let $a:=3, $y:=4, $x:=5  
    return g($y,$a)
```

- May not reassign global variables

```
declare variable $z as onyx.types.Integer { 3 };  
declare function g($a) { $a + $z };  
declare variable $z as onyx.types.Integer { 9 };  
declare function h($a) { $a * $z };
```

Variables bound in for and let clauses of FLWR expressions

- Any variable bound in a **for** or **let** clause is in scope until the end of the FLWR expression in which it is bound
- If the variable name used in the binding was already bound in the current scope...
- ... it is bound again with a new value ...
- ... hiding any previous definition
- The variable goes out of scope once the FLWR expression that first bound the variable ends

```
let $a := 3
  let $a := 6
  return $a
```

```
let $n := 3
  for $i in (3,4)
    for $j in ($i,7)
      let $n := $n+2
      return $n
```

Another Example

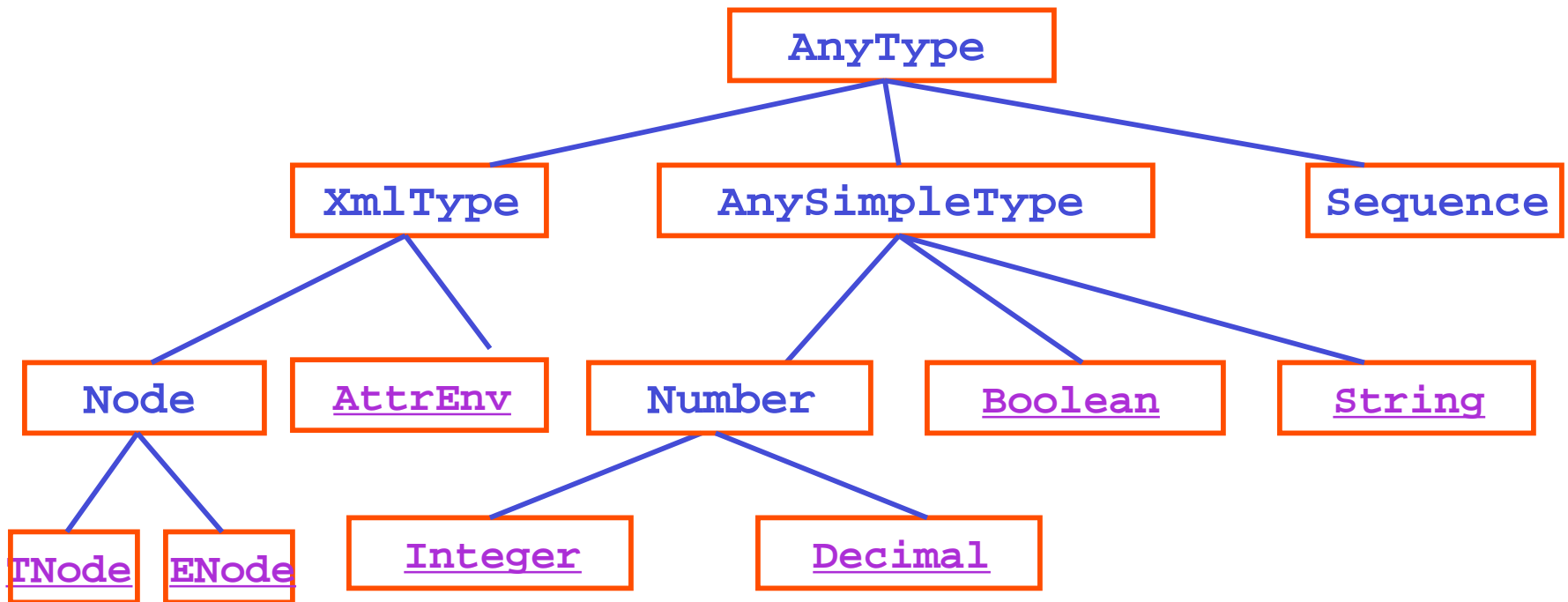
- This FLWR expression
let \$n := 3
 for \$i in (3,4)
 for \$j in (\$i,7)
 let \$k := 12
 let \$n := \$n+2
 where \$j < \$n
 return \$i * \$j
- Evaluates to the sequence 9 16

Type coercions in Onyx

- In Onyx, there are three forms of implicit type conversion
 - comparison operations involving
Integer/Decimal, Integer/String, Decimal/String
 $3 > 1.0, 3 > "1.0", 3 > "1" \Rightarrow \text{true true true}$
 - singleton and Sequence: $3 \equiv (3)$
 - function calls
declare function GT(\$a as onyx.types.Decimal,
 \$b as onyx.types.Decimal)
 as onyx.types.String
 { string(\$a) + " " + string(\$b) };
 $\text{GT}(3,1) \Rightarrow 3\ 1$
- Everything else must be done with explicit type casting functions, `integer`, `string`, etc.

Type hierarchy in Onyx

- Onyx is a strongly typed language:
every value has a definite type



Abstract types in Onyx

- Onyx cannot instantiate objects of an abstract type
`Number` `Node` `AnySimpleType` etc..
- Abstract types play a role in explaining the semantics of the language
- For example, the specification states
“The equality and non-equality operators compare two values of `AnySimpleType` of the same type.”

“Parameters and the return value may be given a definite type, but this information is optional. By default, an untyped parameter or return result is given the type `onyx.types.AnyType`.”

Next

ODOM and the tree structure of OXML

The ODOM, Document order, and the tree structure of OXML

- The Onyx Document Object Model (**ODOM**) defines a tree structure for an OXML document
- Customary tree relationships apply: root, parent, child, sibling, ancestor, descendant
- ODOM supports basic relationships only: children
- We build the rest ourselves
- This is the spirit of a “kernel” language: we rely on simple constructs to help us “bootstrap” the rest
- *Document order* for nodes in an ODOM tree corresponds to a preorder traversal of the tree
 - Attributes come after their element and before any children

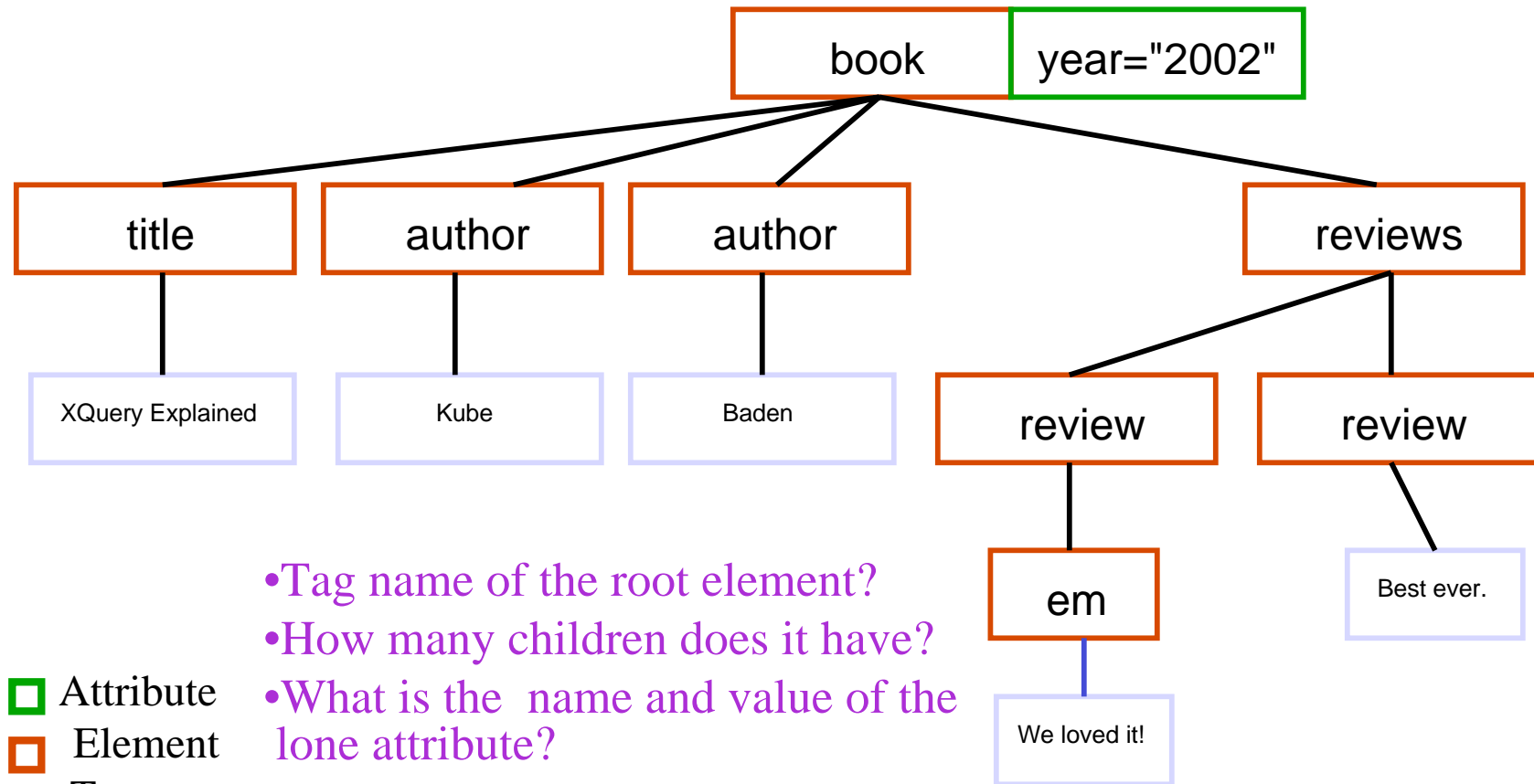
The ODOM

- The ODOM is an abstract interface for manipulating XML documents
- We'll provide you with a library implementing the interface
- ODOM defines two types of Nodes
 - Element**
 - Text**
- It also provides an attribute environment that we use to construct element attributes
- For a given document, these nodes exist in a tree structure that essentially follows the that imposed by the OXML grammar

An OXML document

```
<?xml version="1.0" encoding="utf-8" ?>  
  
<book year="2002">  
  
  <title>XQuery Explained</title>  
  <author>Kube</author><author>Baden</author>  
  
  <reviews>  
    <review> <em>We loved it!</em></review>  
    <review>Best ever.</review>  
  </reviews>  
  
</book>
```

ODOM tree for the OXML document

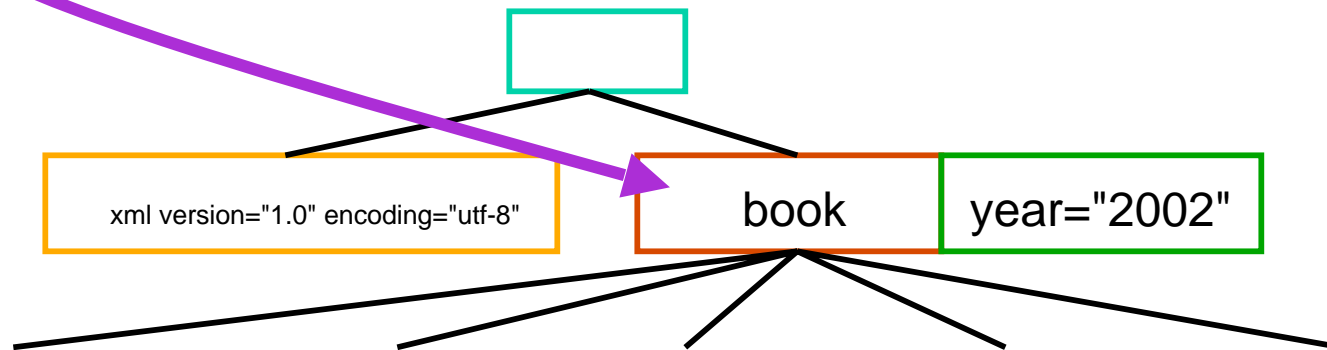


- Tag name of the root element?
- How many children does it have?
- What is the name and value of the lone attribute?

- Attribute
- Element
- Text

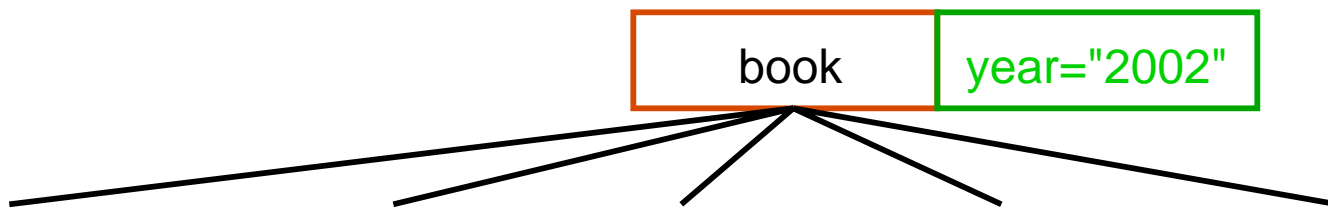
ODOM Node relationships

- There is a **Document** node corresponding to the overall document; sometimes called the *document root* node
- In ODOM, the top level node is either a **text** node or an **element** node
- We will refer to this as the *root element*



ODOM Node relationships

- **Attributes** are not considered children of their corresponding element nodes
 - attributes are properties, not content
- **Text** nodes may not have children



Producing OXML in Onyx

- Onyx permits three kinds of operations on OXML
- Serializing OXML trees for display (output)
- Creating XML nodes
 - the builtin `document()` function (later on)
 - computed constructors `tnode()` and `enode()`
- Selecting XML nodes
 - querying tags `tagname`
 - querying attributes `getAttributeValue`

Recapping from last time: the OXML Document Structure and Grammar

- An OXML document has a recursive structure
 - document ::= prolog element
 - element ::= EmptyElementTag
 | StartTag content EndTag
 - content ::= Text | Element*
 - Text ::= [^<&]*
- An element can contain other elements, interspersed with character text data
 - <a> <b1>abc</b1> <b1>xyz</b1>
 - <a> <b1>Mixed Content with text and elements</b1>
Not Allowed

Node construction

- When node is evaluated in an Onyx program, the following information is available
- For element node constructors: the tag name of the element, any attribute nodes for the element, and any children nodes (elements or text nodes) for the element
- For text node constructors: the tag name of the element, any attribute nodes for the element, and the text
- For Attribute constructors: the name of the attributes, and their values
- The value of the constructor is a node. How to create it?

Onyx and returned results

- The result of an Onyx program is always an OXML Document
- If the returned result is a non-node type, Onyx wraps the result to conform, using a **<Result>** tag

3+4 ⇒

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

```
<onyx-result>
```

```
  <Result>7</Result>
```

```
</onyx-result>
```

- The **<Result>** tag isn't used if the result is a Document

`tnode("TNODE", attrenv(), "Content")` ⇒

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

```
<onyx-result>
```

```
  <TNODE>Content</TNODE>
```

```
</onyx-result>
```

Constructing text nodes

Text ::= [^<&]*

- To create content we need a text or an element node
- We construct a text node with a given tag name

```
let $attr := attrenv()  
let $elt := tnode( "book", $attr , "abc")  
return $elt
```

```
<?xml version="1.0" encoding="UTF-8"?>  
<onyx-result>  
  <book>abc</book>  
</onyx-result>
```

Constructing element Nodes

element ::= EmptyElementTag | StartTag content EndTag
content ::= Text | Element*

- OXML documents have a recursive structure
- An element can contain other elements, interspersed with character text data
- Once we instantiate an element node, we can add children to it, a side effect producing operation

```
let $attr := attrenv()  
let $elt := enode( "D", $attr )  
let $elt2 := tnode( "E", $attr , "5")  
let $elt := addChildNode( $elt, $elt2 )  
let $elt3 := tnode( "F", $attr , "6")  
let $elt := addChildNode( $elt, $elt3 )
```

```
<D>  
  <E>5</E>  
  <F>6</F>  
</D>
```

Attributes

- A tag may contain zero or more attributes

EmptyElementTag ::= '<' Name (Attribute)* '/>'

StartTag ::= '<' Name (Attribute)* '>'

EndTag ::= '</' Name '>'

Attribute ::= Name '=' Quotes Attvalue Quotes

- No two occurrences of the same attribute name may appear in a given tag: an element's attributes form a set
`<element symbol="Fe" num="56">Iron</element>`
- Attributes are considered to represent *properties* of the elements in whose tags they appear; they are not part of the element's *content*

Working with Attributes

Attribute ::= Name '=' Quotes Attvalue Quotes

- Attributes may represent *properties* of an element, but are not part of the element's *content*
- Once you have an `attenv` object, you can set its attribute values with the `addAttribute()` method (producing a side effect)
- Once you have the attributes, you can construct nodes
- An element's attributes form a set: duplicates

```
<element symbol="Fe" num="56">Iron</element>
```

```
let $attr := attenv()  
let $attr := addAttribute(attenv(), "symbol", "Fe")  
let $attr := addAttribute($attr, "num", "78")  
let $attr := addAttribute($attr, "num", "56")  
let $elt := tnode( "element", $attr , "Iron")  
return $elt
```

Empty elements

EmptyElemTag ::= '<' Name (Attribute)* '/>'

S Tag ::= '<' Name (Attribute)* '>'

E Tag ::= '</' Name '>'

let \$attr := attrenv()

let \$elt := enode("a", \$attr)

return \$elt

<onyx-result>

<a/>

</onyx-result>

Some limitations

- There are restrictions on tag names and attribute names

```
let $attr := attrenv()
```

```
let $elt := tnode( "<", $attr , "NOT ALLOWED")
```

```
return $elt
```

```
<onyx.error.SemanticError>
```

```
<DynamicError column="13" line="2">Tagname &lt; is an invalid  
  OXML tagname</DynamicError>
```

```
</onyx.error.SemanticError>
```

DOM behavior

- When adding a child to an element node, duplicates are ignored
- The Onyx XML library inherits this behavior from the W3C DOM
- Only the most recently added child is represented in the output

```
let $a := attenv()
```

```
let $e := enode("S", $a), $c := enode("foo", $a)
```

```
let $e := addChildNode($e, $c)
```

```
let $t := tnode("T", $a, "e"), $c := addChildNode($c, $t)
```

```
let $e := addChildNode($e, $c)
```

```
return $e
```

```
<S>  
  <foo>  
    <T>e</T>  
  </foo>  
</S>
```

Side effects and duplicate removal

- Because `addChildNode()` is a side-effect producing operation, the `for` loop returns 4 references to the same object
- We still create 4 bindings of `elt`
- The **DOM** library removes the duplicates

```
let $attr := attrenv(),
```

```
    $elt := enode( "D", $attr )
```

```
    for $i in 1 to 4
```

```
        let $tag := "E" + string($i),
```

```
            $schild := tnode( $tag,$attr, string($i)),
```

```
            $elt := addChildNode( $elt, $schild )
```

```
    return $elt
```

```
<D>  
  <E1>1</E1>  
  <E2>2</E2>  
  <E3>3</E3>  
  <E4>4</E4>  
</D>
```

Selecting attributes

- Node has the instance method `getAttr`
- `getAttributeKey` returns a sequence of attribute names
- We can iterate over these to extract all the keys

```
let $attr := attrenv()           <element symbol="Fe" num="56">Iron</element>
let $attr := addAttribute(attrenv(),"symbol","Fe")
let $attr := addAttribute($attr,"num","78")
let $attr := addAttribute($attr,"num","56")
let $elt := tnode( "element", $attr , "Iron")
let $ae := getAttrEnv($elt)
let $keys := getAttributeKeys($ae)
for $k in $keys
return getAttributeValue($ae,$k)

<Result>56 Fe</Result>
```

Testing

- In addition to testing our Java code, i.e. JUnit, we can also perform valuable testing in Onyx itself

```
let $attr := attrenv()
let $symbol := "symbol", $num := "num"
let $attr := addAttribute(attrenv(),$symbol,"Fe")
let $attr := addAttribute($attr,$num,"56")
let $elt := tnode( "element", $attr , "Iron")
return $elt
```

```
<onyx-result>
  <element num="56" symbol="Fe">Iron</element>
</onyx-result>
```

Testing

```
<element num="56" symbol="Fe">Iron</element>
```

```
let $ae := getAttrEnv($elt),  
    $keys := getAttributeKeys($ae),  
    $k1 := first($keys), $k2 := first(tail($keys))  
return  
  (length($keys) = 2) and  
  (if ($k1 = $num)  
    then  
    ((getAttributeValue($ae,$k1) = "56") and ($k2 = $symbol) and  
     (getAttributeValue($ae,$k2) = "Fe"))  
    else  
    (($k1 = $symbol) and (getAttributeValue($ae,$k1) = "Fe") and  
     ($k2 = $num) and (getAttributeValue($ae,$k2) = "56"))))
```

Formatting OXML

- White space is often needed to improve the readability of OXML
- We've provided `xmlpp` a pretty-printing filter and is available in the 131A environment (`~/../public/Tools`)
- Don't insert spaces, tabs, and other characters between element tags unless you intend to change the structure of the document
- Use `diff` to compare such output against a reference
- Use `xmldiff` to compare OXML documents

A real world problem

- Search a database of baseball players
- Return a collection of records consisting of the surnames of all players who throw left-handed

The Database: stats.xml

```
<SEASON>
  <LEAGUE>
    <DIVISION>
      <TEAM>
        <PLAYER> </PLAYER>
        . . .
        <PLAYER> </PLAYER>
      </TEAM>
      . . .
      <TEAM> .. </TEAM>
    </DIVISION>
    . . .
    <DIVISION> . . . </DIVISION>
  </LEAGUE>
  <LEAGUE> . . . </LEAGUE>
  . . .
</SEASON>
```

A Closer look...

```
<TEAM>
  <TEAM_CITY>Atlanta</TEAM_CITY>
  <TEAM_NAME>Braves</TEAM_NAME>
  <PLAYER>
    <SURNAME>Springer</SURNAME>
    <GIVEN_NAME>Russ</GIVEN_NAME>
    <THROWS>Right</THROWS>
    <POSITION>Relief Pitcher</POSITION>
    . . .
  </PLAYER>
  .
  <PLAYER>
    . . .
  </PLAYER>
</TEAM>
```

A query in Onyx

Return a collection of records consisting of the surname and games played of all players who played between 50 and 100 game (will return to this later in the quarter)

The result of that query will be a sequence of XML elements like

```
<player>  
  <SURNAME>Guillen</SURNAME>  
  <GAMES>83</GAMES>  
</player>  
  . . .
```

```
<player>  
  <SURNAME>Berg</SURNAME>  
  <GAMES>81</GAMES>  
</player>
```

... one for each qualifying player in the database

Code for the curious

Return a collection of records consisting of the surname and games played of all players who played between 50 and 100 game (will return to this later in the quarter)

```
let $players := getDescendantOrSelfByTagName( $d, "PLAYER" )
for $player in $players
  let $surname := getChildrenByTagName( $player, "SURNAME" ),
      $games := getChildrenByTagName( $player, "GAMES" ),
      $en := enode("player",attrenv()),
      $en := addChildNode($en,first($surname)),
      $en := addChildNode($en,first($games)),
      $g := toInt(string(first($games)))
  where (( $g >= 50) and ($g <= 100))
  return $en
```

Next time

- Overview of compilers
- Introduction to lexical analysis