CSE 131A
Lecture 17

Data base query and the ODOM

Scott B. Baden
Department of Computer Science & Engineering
University of California, San Diego
Winter 2007

Readings

- Onyx Manual and Semantic Specification
  ieng6.ucsd.edu/~cs131w/Assign/A3/SemanticSpec.html
- Onyx XML/DOM Appendix
  ieng6.ucsd.edu/~cs131w/Assign/A3/Appendix2_OXMLODOM.html

Discussion on sequences

- A Sequence is an ordered collection of zero or more items
- Sequences are constructed using the sequence construction operator comma ‘,’

  3, 4, 5

- Parenthesis are used for grouping purposes
- The value of a parenthesized expression or expression sequence is the value of the contained expression or expression Sequence
- According to the grammar

  ExprList ::= ExprSingle (, ExprSingle)*

Singleton sequences

- A Sequence containing exactly one item is a singleton sequence
- An item is identical to a singleton sequence containing that item in the context where a sequence is required `declare function f($a as onyx.types.Sequence){$a}; f(3)`
- An item can be promoted to a sequence contained that item
- But not the other way around:

```
let $a :=(3,4), $b := tail($a) return $b + 7
Function with prototype
op:numeric-add(onyx.types.Sequence,onyx.types.Integer)
not found
```

Abstract types in Onyx

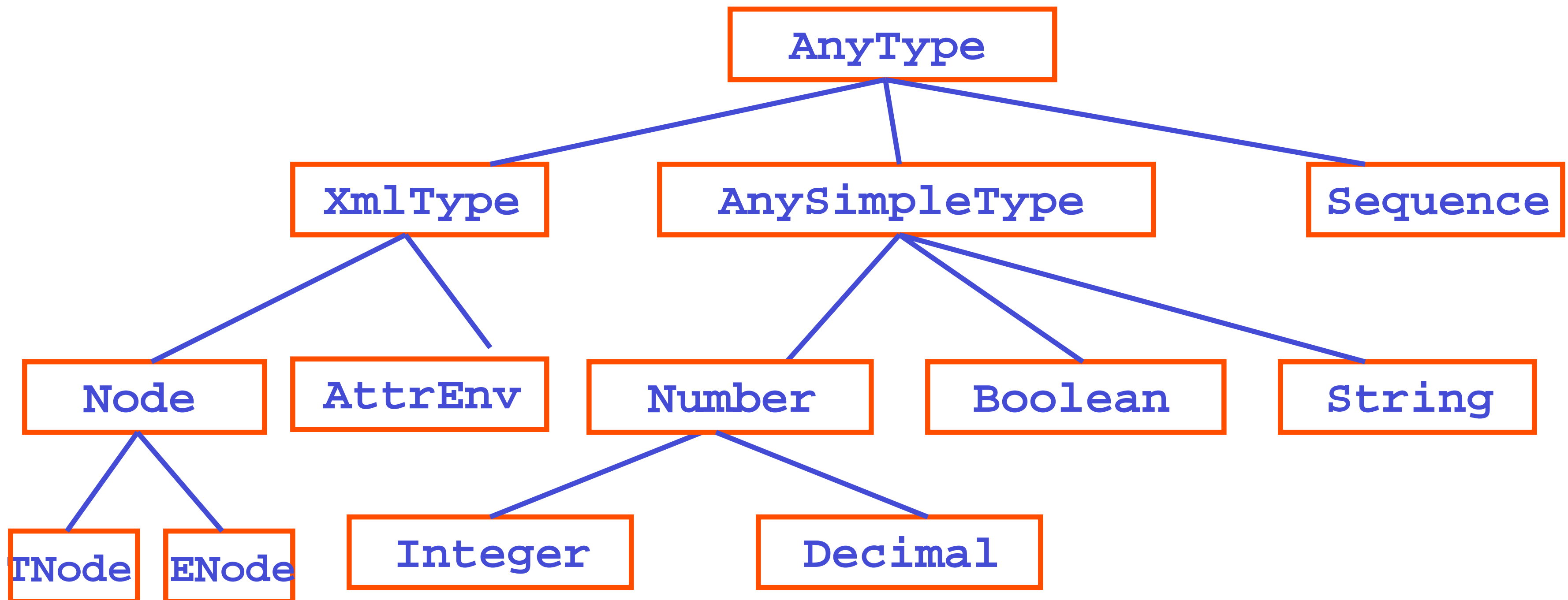
- Abstract types don't exist in Onyx code
- But they play a role in explaining the semantics of the language
- For example, the specification states

  "If an exact function prototype match is found, it is selected, otherwise type substitution is applied based on available prototypes with the same function name."

  "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 onxy.types.AnyType."

Type hierarchy in Onyx



Type matching of arguments

- To resolve a function call, attempt to match the number and types of arguments to the function prototype definition
- If an exact argument type match isn't possible type substitution is applied based on available prototypes with the same function name
  - The use of a value whose actual type is derived from the expected type
    `declare function g ($a){$a + 4};g(1)`
  - The passed value retains its original type within the body of the function
- If there are no available type substitutions, then type promotion is applied
  - e.g., Integer to Decimal, for builtin operators
    `declare function f ( $a as onyx.types.Decimal) {(2,$a)};`
  - The actual parameter is converted to the expected type across the call and within the body of the function, $a is of type Decimal

Matching with multiple arguments

- If there are multiple matches, then the prototype requiring the least number of type promotions is used
  - For counting purposes, type substitutions are ignored
  - # of arguments must agree

```
declare function f($a,$b){"A"};
declare function f($a as onyx.types.Integer, $b as onyx.types.Decimal){"B"};
declare function f($a as onyx.types.Decimal, $b as onyx.types.Integer){"C"};
    f(1,2) → A
```

Ambiguity

- When there are multiple matches, Onyx tries to help locate a possible candidate

```
declare function f ( $a as onyx.types.Decimal ) { (4,$a) };
declare function f ( $a as onyx.types.Sequence ) { (5,$a) };
f(3)  →

<onyx.error.SemanticError>
  <DynamicError column="2" line="3">
  <ErrorMessage>Function with prototype f(onyx.types.Integer) not found
   </ErrorMessage>
  <PossibleMatch>onyx.types.AnyType f(onyx.types.Decimal)</PossibleMatch>
  <PossibleMatch>onyx.types.AnyType f(onyx.types.Sequence)</PossibleMatch>
 </DynamicError>
</onyx.error.SemanticError>
```

Ambiguity with a mixture of substition and promotion

- Recall that Onyx does not count substitutions when it attempts to match against the fewest number of promotions
- With multi argument functions, it may not possible to resolve a prototype with a mixture of type substitutions and exact matches only
- The outcome is not defined

```
declare function f ($a,$b){"A"};
declare function f ($a as onyx.types.Integer, $b ){"B"};     f(1,2) →
Function with prototype f(onyx.types.Integer,onyx.types.Integer) not found
Possible matches
  onyx.types.AnyType f(onyx.types.AnyType,onyx.types.AnyType)
 onyx.types.AnyType f(onyx.types.Integer,onyx.types.AnyType)
```

- But note, the following, where the second definition overrides the first

```
 declare function f ($a){"A"};
declare function f ($a as onyx.types.Integer){"B"};   f(1) → B
```

The ODOM

OXML, ODOM and Onyx

- Onyx provides basic operations for manipulating collections of OXML documents represented in memory via the ODOM interface
- The ODOM operations are supported via builtin functions
- They may be used to build sophisticated queries
- Onyx supports three kinds of operations on OXML

- Serializing OXML trees for display (output)

- Creating XML nodes
  - the builtin document() function
  - computed constructors tnode( ) and enode( )

- Selecting XML nodes
  - querying tags tagname
  - querying attributes getAttributeValue

Supporting the ODOM in Onyx

- You only have to implement the call method for the Node constructors and deconstructors that interface with the provided Onyx_XML library
- This is identical to how the other built-ins are handled
  - Get FunctionTableEntry from functionTable
  - Lookup based on function name and signature
  - bind arguments and instantiate a new context
  - FunctionTableEntry.call()
  - pop off the arguments
- The Onyx_XML library does the "heavy lifting"

Sidebar: working with the function table

- If you used a java.util.Vector or java.util.HashTable to implement functionTable
  - There is a search method that checks objects for equality using the equals() method
  - Define equals() inside your functionTableEntries and use HashTable's search method
- For built-in functions, the name and number of arguments is known in advance
  - Make a separate class that extends functionTableEntry for every user-defined function
  - Pre-initialize the functionTable with the built-ins
- For user-defined functions, create an instance of a functionTableEntry in the functionDefinition ASTNode

ODOM and the tree structure of OXML

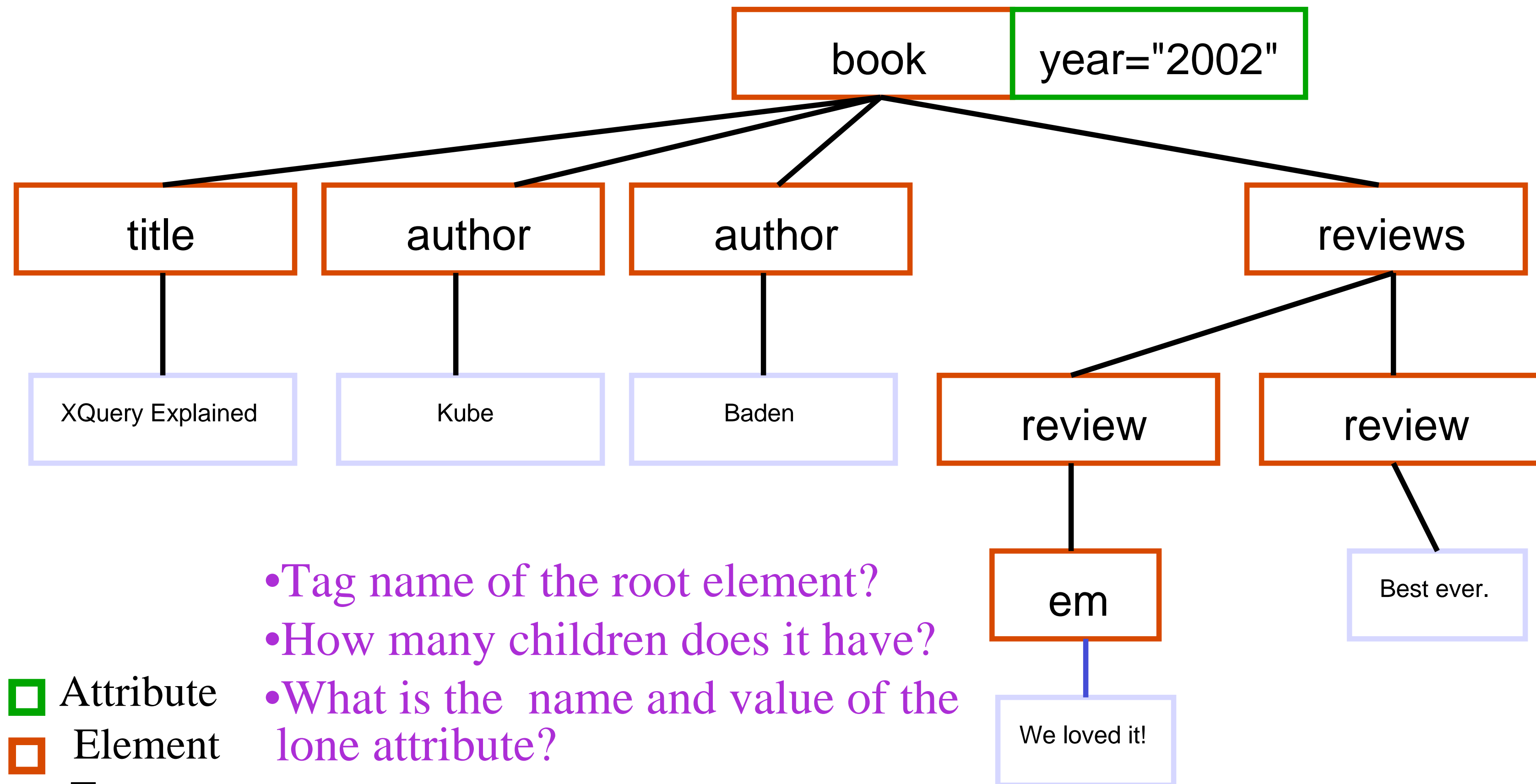
- The tree structure of OXML is made explicit in the Object Document Object Model
- Customary tree relationships apply: root, parent, child, sibling, ancestor, descendant
- ODOM provides basic relationships: children
- We build the rest ourselves
- This is the spirit of a “kernel” language: we rely on simple constructs to help us “bootstrap” the rest
- ODOM defines two types of Nodes

  Element **Text**
- It also provides an attribute environment that we use to construct element attributes

An OXML document

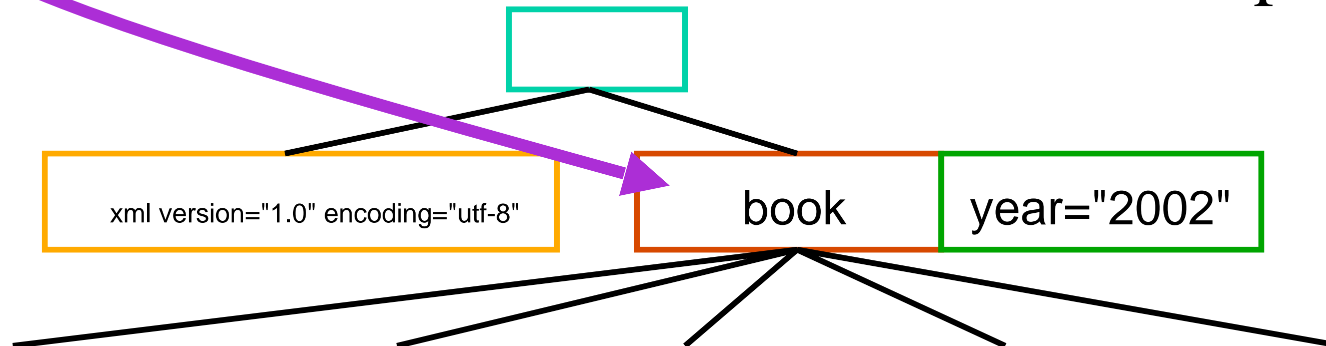
```
<?xml version="1.0" encoding="utf-8" ?>

 <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



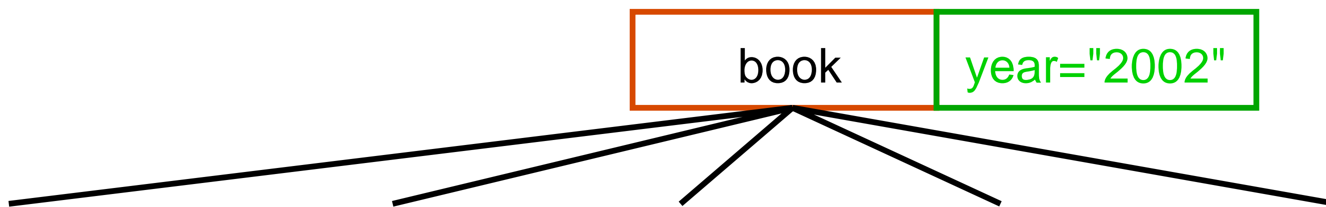
ODOM Node relationships

- In DOM, there is a Document node that corresponds 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*
- We use the print method of element or text node to produce a well formed xml document with prolog



ODOM Node relationships

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



Interfacing with XML files

- If the result of evaluating an Onyx program includes values which are OXML nodes, this OXML must be output in printable form
- This can be easily implemented using the print method of Node
- To read in an XML document: the builtin Onyx document( ) function...
  - takes one argument that evaluates to a string, and
  - returns a Node representing the document root of the OXML document

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?

OXML Grammar review

- The syntax of an OXML document can be (almost) defined using a BNF-like context-free grammar
- From the W3C XML 1.0 documentation, with some simplifications
- A document consists of a prolog followed by an element

document ::= prolog element

- The element contains the document, the remaining parts contain additional information used to interpret the document
- Onyx_xml generates the prolog for us: the print method

OXML Grammar: element

element ::= EmptyElemTag | STag content ETag
content ::= Text | Element*
Text ::= [^<&]*

- To create content we need a text or element node
- We construct a text node with a given tag name
- There are no attributes here

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

<book>abc</book>

Restrictions

- Tag and attribute names may not contain disallowed symbols

```
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>
```

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 attrenv object, you can set its attribute values with the addAttribute( ) method
- 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 := attrenv()
let $attr := addAttribute(attrenv(),"symbol","Fe")
let $attr := addAttribute($attr,"num","78")
let $attr := addAttribute($attr,"num","56")
let $elt := tnode( "element", $attr , "Iron")
return $elt
```

OXML Grammar: element

```
element ::= EmptyElemTag | STag content ETag
content   ::=   Text | Element*
Text ::= [^<&]*
```

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

```
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 )
```

```
<onyx-result>
  <D>
    <E>5</E>
    <F>6</F>
  </D>
</onyx-result>
```

Removal of duplicates

- 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

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

```
let $e := enode("S", attrenv()), $c := enode("foo",attrenv())
let $e := addChildNode($e,$c)
let $t := tnode("T",attrenv(),"e"),  $c := addChildNode($c, $t)
let $e := addChildNode($e,$c)
return $e
```

More of the grammar

- Content is **one occurrence** of **Text**

  abc xyz

- OR … **zero or more Elements**

  <a> abc xyz

- Mixed content with text and elements is *not* allowed

  **<a> NOT ALLOWED **

- Grammar

  content ::= Text | Element*

Empty elements

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

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

ETag ::= '</' Name '>

```
let $attr := attrenv()
let $elt := enode( "a", $attr )
return $elt
```

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

Selecting attributes

- Node has the instance method getAttrEnv
- getAttributeKey returns a sequence of attribute names
- We can iterate over these to extract all the keys, but the order of the attributes is not defined

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

```
let $attr := attrenv()
let $attr := addAttribute(attrenv(),"symbol","Fe")
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>

Attribute ordering

- The ordering of attributes is not defined
- It is not significant in comparing documents with xmldiff

```
<?xml version="1.0" encoding="UTF-8"?>
<onyx-result>
  <element num="56" symbol="Fe">Iron</element>
</onyx-result>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<onyx-result>
  <element symbol="Fe" num="56">Iron</element>
</onyx-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")))
```

Next time

Database queries with Onyx