

CSE 131A

Lecture 18

Data base queries put to work

Scott B. Baden

Department of Computer Science & Engineering

University of California, San Diego

Winter 2007

Announcements

- The deadline for A3 is Monday 3/12, **11.59 PM** (not 9.00 PM)
- The Final exam will be held in CNTR 212 Tuesday, March 20th
- There will be an overflow room downstairs in CNTR 109
- Room assignments
 - If the first letter of your last name begins with the letter A through K, go to CNTR 212
 - Else, go to CNTR 109

Unspecified behavior

- “I was under the impression we should be programming based on the semantics and not the interpreter” **YES!**
- The spec is the authoritative source regarding semantics
- The interpreter enables you to explore or discover unspecified behavior that you might or might not have thought about otherwise!
- You should always hand analyze any output from the interpreter portal and verify against what you expected

Anomalous behavior

- e
`<onyx.error.ParserError>`
 `<StaticError column="-1" line="-1">Syntax Error</StaticError>`
`</onyx.error.ParserError>`
- Should be
 `<StaticError column="1" line="1">Syntax Error</StaticError>`
- (a,b)
`<StaticError column="3" line="1">Syntax Error</StaticError>`
- 2 = true
 `<DynamicError column="3" line="1">Function with prototype
op:equals(onyx.types.Integer,onyx.types.Boolean) not
found</DynamicError>`

Anomalous behavior

- `2 = true`
`<DynamicError column="3" line="1">Function with prototype
op:equals(onyx.types.Integer,onyx.types.Boolean) not
found</DynamicError>`
- `true = (3,4), 1=(3,4)` is OK
`<DynamicError column="6" line="1">
<ErrorMessage>Function with prototype
op:equals(onyx.types.Boolean,onyx.types.Sequence)
not found</ErrorMessage>
<PossibleMatch>onyx.types.Boolean
op:equals(onyx.types.AnySimpleType,onyx.types.AnySimpleType)<
/PossibleMatch>
</DynamicError>`

Anomalous string conversion

- `string()` doesn't accept enodes
`string(enode("test", attenv()))`
- `<DynamicError column="1" line="1">Function with prototype string(onyx.types.ENode) not found</DynamicError>`
- But note `string((1, enode("test", attenv())))`

`<Result>`

`[1 <?xml version="1.0" encoding="UTF-8"?><test/>]`

`</Result>`

- Behavior not defined
- Document implies this is an error, but doesn't discuss the case of a Sequence

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

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>
```

Reporting possible matches

- In what order do we report the possible matches

```
declare function f($a as onyx.types.String, $b as onyx.types.Sequence){1};  
declare function f($b as onyx.types.Integer,$b as onyx.types.Integer){2};  
f(3,1.0)
```

- Represent the prototype as a string, and sort

```
<ErrorMessage>Function with prototype f(onyx.types.Integer,onyx.types.Decimal) not found</ErrorMessage>  
  <PossibleMatch>onyx.types.AnyType f(onyx.types.Integer,onyx.types.Integer)</PossibleMatch>  
  <PossibleMatch>onyx.types.AnyType f(onyx.types.String,onyx.types.Sequence)</PossibleMatch>  
</DynamicError>
```

Data base query with Onyx

- How to perform database queries with Onyx
- Onyx is based on the XML Query Language, XQuery
- XQuery has various constructs to facilitate querying
- Of note: the XPath syntax
- The notation is more compact than with Onyx
- We can think of having a source -to-source translator from path expressions to Onyx ODOM navigation calls
- Thus, Onyx is the target

For Reference

- Essential XQuery - The XQuery Language
<http://www.yukonxml.com/articles/xquery>
- Learning tools
<http://www.activsoftware.com/xml/xpath>
<http://www.zvon.org/xxl/XPathTutorial>
- W3C XML Query (XQuery)
<http://www.w3.org/XML/Query>
<http://www.w3.org/TR/xpath20>
- Java DOM documentation, especially the `org.w3c.dom` package in Java 1.6
<http://java.sun.com/webservices/docs/1.6/api>
- Wikipedia entry on XPath
<http://en.wikipedia.org/wiki/Xpath>

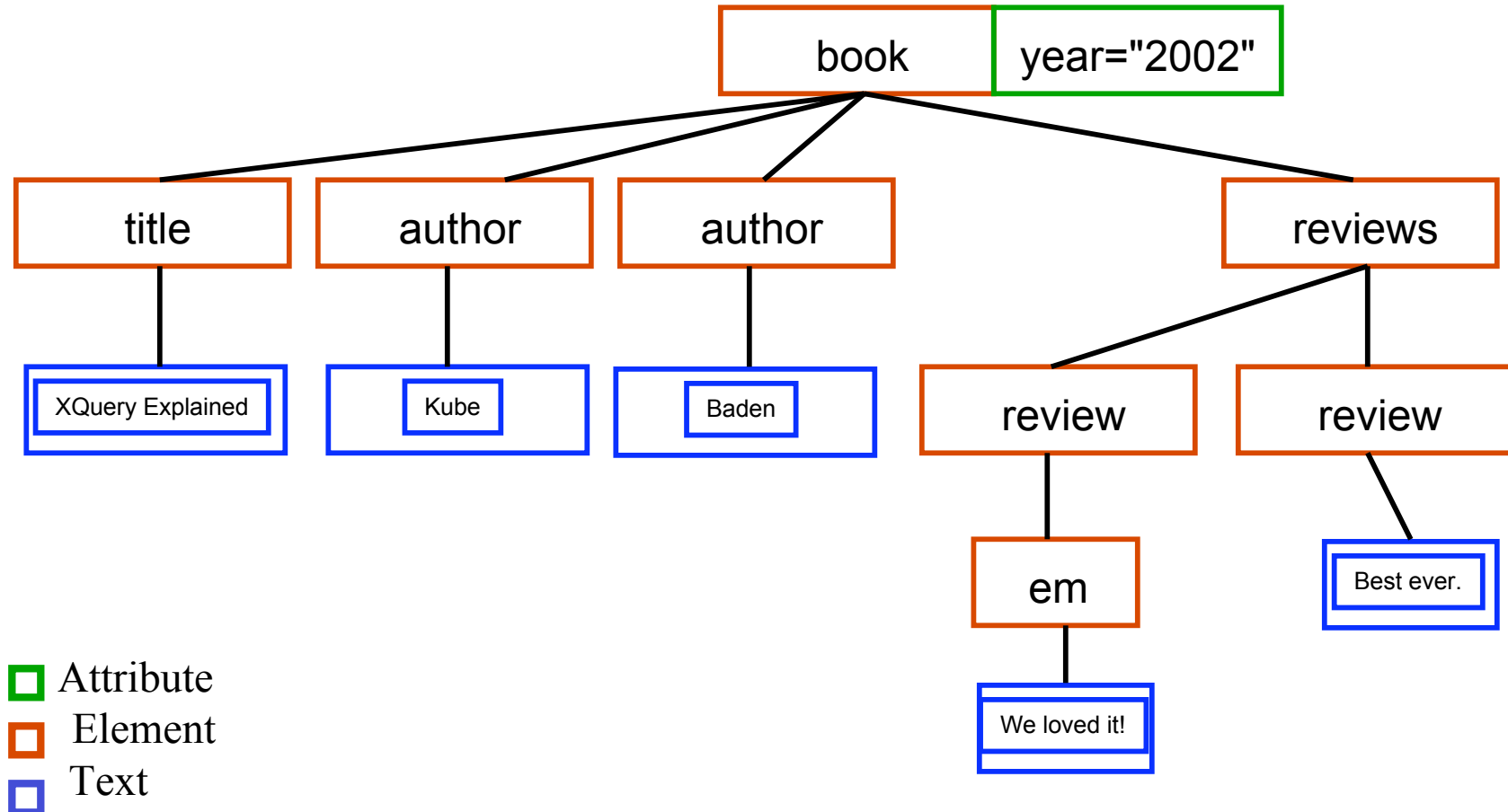
Path Expressions

- Path expressions are used to locate and select nodes in an XML document according to the corresponding DOM tree structure
- We'll use OXML and ODOM here
- The value of a path expression is a sequence of (zero or more) nodes
- A path expression is broken down into a set of *step expressions*

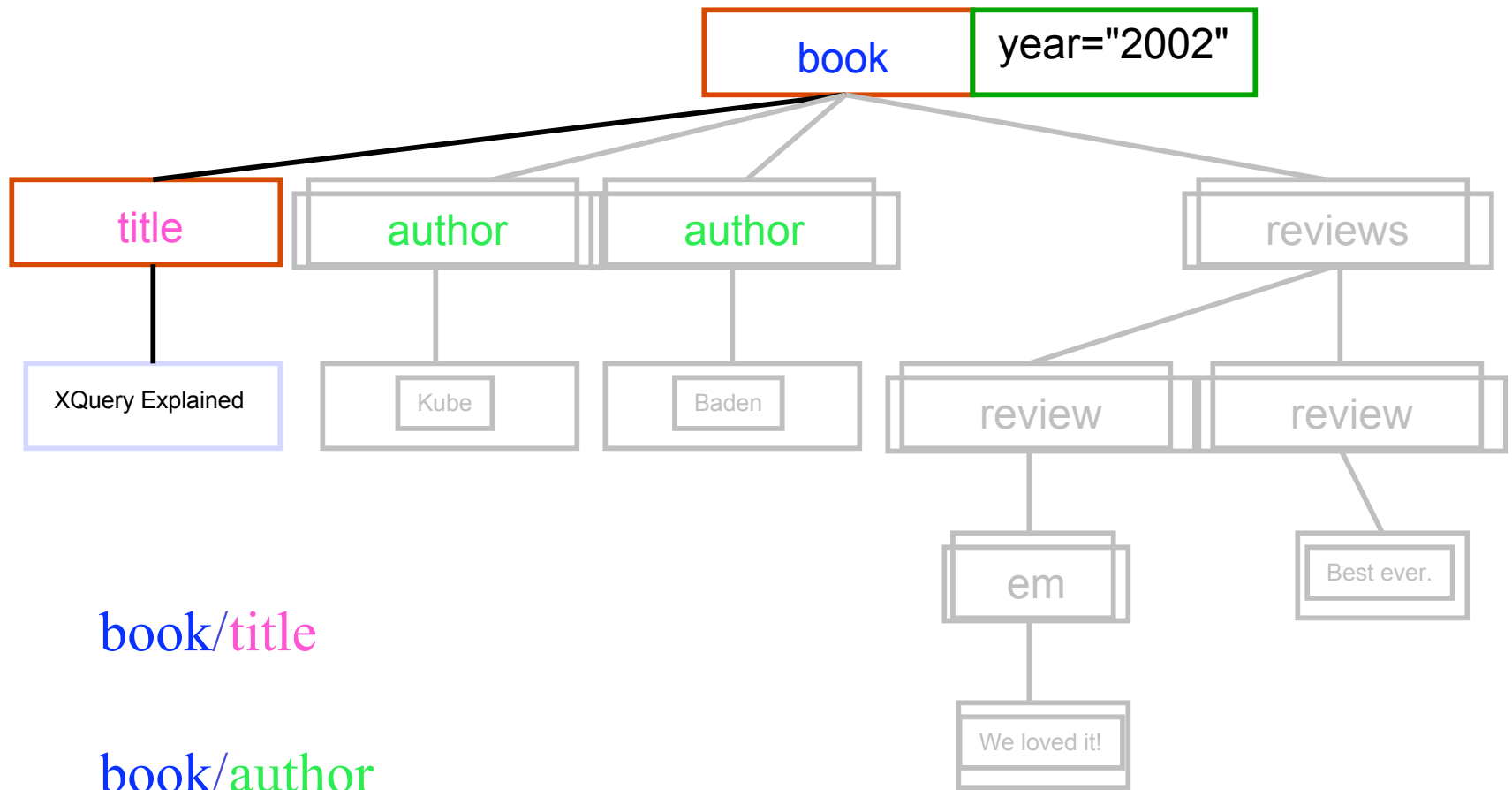
An OXML document

```
<?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



Visualizing path expressions



book/title

book/author

Selecting tree nodes

- We can think of a step expression as a kind of associative indexing operation
- We provide a name (optionally a predicate)...
- ... we locate nodes with a tag that matches the name (and satisfies any condition specified by the predicate)
- Each step moves one level further down the tree, narrowing the search for the next step
- Each step provides a *context* for the next one

The semantics of path expressions

- Given an DOM Node or sequence of Nodes, path expressions select certain descendant nodes for further processing

- We'll look at a subset of XPath operators:

- the “children of” step operator `/`

`$d/b` = *children* of node `$d` with tag names equal to `b`

- the “descendants or self of” step operator `//`

`$d//b` = *descendants* of `$d` (and `$d` itself) with tag names equal to `b`

- the “attributes of” step operator `/@`

`$d/fruit [@color="yellow"][@taste="sour"]` = nodes with attributes color = “yellow” and taste=“sour”

- The result of is a sequence of Nodes

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

The simplest step: Children

- Given a sequence of nodes $\$d$
- The path expression $\$d/AAA/CCC$ returns the sequence of nodes with tag “CCC” that are children of the root node with tag AAA
- If there is deeper nesting, all the sequences are concatenated together into a single sequence

```
<AAA>  
  <BBB/>  
  <CCC/>  
  <BBB/>  
  <BBB/>  
  <DDD>  
    <BBB/>  
  </DDD>  
  <CCC/>  
</AAA>
```

Descendents

- The path expression `$d//b` returns the *descendants* of the nodes in the sequence of nodes `$d ...`
... with tag names equal to 'b' ...
... as well as `$a` itself
- `$d//DDD/BBB` select all elements `BBB` which are children of `DDD`
- What about `$d//BBB`

```
<AAA>
  <BBB/>
  <CCC/>
  <BBB/>
  <DDD>
    <BBB/>
  </DDD>
  <CCC>
    <DDD>
      <BBB/>
      <BBB/>
    </DDD>
  </CCC>
</AAA>
```

Descendants

- The path expression `$d//b` returns the *descendants* of the nodes in the sequence of nodes `$d ...`
 - ... with tag names equal to 'b' ...
 - ... as well as `$a` itself
- `$d//DDD/BBB` select all elements `BBB` which are children of `DDD`
- What about `$d//BBB`

```
<AAA>
  <BBB/>
  <CCC/>
  <BBB/>
  <DDD>
    <BBB/>
  </DDD>
  <CCC>
    <DDD>
      <BBB/>
      <BBB/>
    </DDD>
  </CCC>
</AAA>
```

Tags at multiple levels

- \$d//CCC
- select all elements CCC

```
<AAA>  
  <CCC>  
    <BBB/>  
    <BBB/>  
    <BBB/>  
  </CCC>  
  <DDD>  
    <BBB/>  
    <BBB/>  
  </DDD>  
<EEE>  
  <CCC/>  
  <DDD/>  
</EEE>  
</AAA>
```

Predicates with attributes

- The **@** operator refers to attributes of an element
- **$\$d//b[@attr=\$c]$** returns
 - the descendants of node **$\$d$** ...
 - ... with tag names equal to **b** and...
 - ... with attribute **$attr$** having value **$\$c$**
- For each **Node** in **$\$d//b$** , we evaluate the condition within []; if its boolean value is **true**, the **Node** is added to the resulting sequence of Nodes
- Multiple predicates are possible
 $\$d//b[@attr1=\$c1][@attr2=\$c2]...[@attrN=\$cN]$

Multiple predicates

- Consider the following predicated path expression
\$d / fruit [@color="yellow"] [@taste="sour"]
- evaluates to the list of all (element) children of **\$d** with
 - tagname **fruit** and
 - attribute name **color** with value **"yellow"** and
 - attribute named **taste** with value **"sour"**

Onyx support for queries

- Onyx supports
Attribute and child

- API provides

children (\$n as Enode) as sequence

tagname (\$n as Node) as String

getAttributeValue(\$a as AttrEnv, \$s as string) as AnyType

string(\$n as Tnode) as String

Implementing path expressions in Onyx

- To evaluate a path expression we iterate over sequences of nodes, querying the nodes' properties
- We need to perform dynamic type checking, in case a node is not of the correct type

isENode(\$n as anyType) as Boolean

isTNode(\$n as anyType) as Boolean

isNode(\$n as anyType) as Boolean

Query examples

<bib> <!-- an example from §1.1.2 of <http://www.w3.org/TR/xmlquery-use-cases> -->

<book year="1994"> <title>TCP/IP Illustrated</title>
<author><last>Stevens</last><first>W.</first></author> <publisher>Addison-
Wesley</publisher> <price> 65.95</price> </book>

<book year="1992"> <title>Advanced Programming in the Unix environment</title>
<author><last>Stevens</last><first>W.</first></author> <publisher>Addison-
Wesley</publisher> <price>65.95</price> </book>

<book year="2000"> <title>Data on the Web</title>
<author><last>Abiteboul</last><first>Serge</first></author>
<author><last>Buneman</last><first>Peter</first></author>
<author><last>Suciu</last><first>Dan</first></author> <publisher>Morgan Kaufmann
Publishers</publisher> <price>39.95</price> </book>

<book year="1999"> <title>The Economics of Technology and Content for Digital
TV</title> <editor> <last>Gerbarg</last><first>Darcy</first>
<affiliation>CITI</affiliation> </editor> <publisher>Kluwer Academic
Publishers</publisher> <price>129.95</price> </book>

</bib>

The query

- Extract the book fields
- For each book, extract the author field
- For each author, extract the last name field

```
for $book in document("book.xml")//book
  for $author in $book/author
    return $author/last
```

has value

<last>Stevens</last>

<last>Stevens</last>

<last>Abiteboul</last>

<last>Buneman</last>

<last>Suciu</last>

Implementing the query in Onyx

```
{-- for $book in document("book.xml")//book --}  
for $bk in document("book.xml")  
  let $book := getDescendOrSelfByTag ( $bk, "book")  
  for $b in $book  
{-- for $author in $book/author --}  
{-- return $author/last --}  
    let $au := getChildrenByTag( $b, "author" )  
    for $author in $au  
      for $a in getChildrenByTag( $author,"last")  
        {-- return $author/last --} return $a
```

Implementing the getChildrenByTag

```
declare function getChildrenByTag  
  ( $node, $tag ) as onyx.types.Sequence {  
  if ( isENode( $node ) )  
  then  
    for $child in children($node)  
      where tagname($child) = $tag  
      return $child  
  else  
    () children ($n as ENode) as sequence  
    tagname ($n as Node) as String  
};
```

Implementing the getDescendOrSelfByTag

```
declare function getDOS ( $node, $tagname ) as  
  onyx.types.Sequence {  
    if ( tagname($node) = $tagname ) then $node else () ,  
    (if( isENode($node) )  
  then  
    for $child in children($node)  
    return getDOS( $child, $tagname )  
  else () )  
};
```

\$d//CCC

```
<AAA>  
  <CCC>  
    <BBB/>  
    <BBB/>  
    <BBB/>  
  </CCC>  
  <EEE>  
    <CCC/>  
    <DDD/>  
  </EEE>  
</AAA>
```

Implement the query with a predicate

- Extract the book fields only for year 2000
- For each book, extract the author field
- For each author, extract the last name field

```
for $book in document("book.xml")//book[@year=2000]
  for $author in $book/author
    return $author/last
```

has value

<last>Abiteboul</last>

<last>Buneman</last>

<last>Suciu</last>

Obtaining content from an element

- We got back elements, but sometimes we want the text

```
<sentence>  
  <w> hello </w>  
  <w> out </w>  
  <w> there! </w>  
</sentence>
```

```
for $w in document("t.xml")//w  
return $w/text( )  
hello out there!
```

Obtaining content from an element

- `String(tnode)` provides the desired functionality
for `$w` in `document("t.xml")//w`
return `$w/text()`

`for` `$w1` in `document("t.xml")`

`let` `$w2 := getDescendOrSelfByTag($w1, "w")`

`for` `$w` in `$w2`

`return`(

`if`(`isTNode($w)`)

`then` `string($w)`

`else` ()

)

`<sentence>`

`<w> hello </w>`

`<w> out </w>`

`<w> there! </w>`

`</sentence>`

`<onyx-result>`

`<Result>hello out there!</Result>`

`</onyx-result>`

Useful example xml files

- Look in
~/.. public/Examples/xmlFiles
- Especially `stepexpr.xml`

```
<root node="0">
  <tree/>
  <child parent="0" node="1">
    <gchildren>
      <child parent="1" node="4"/>
      <child parent="1" node="5"/>
      <child parent="1" node="6"/>
    </gchildren>
  </child>
  <child parent="0" node="2">
    <gchild parent="2" node="7"/>
    <child parent="2" node="8"/>
    <child parent="2" node="9"/>
  </child>
  <child parent="0" node="3">
    <child parent="3" node="10"/>
    <child parent="3" node="11"/>
    <child parent="3" node="12"/>
  </child>
</root>
```

Examining the AST with Onyx

- Extract all the variable names

```
let $d := document("stats98lefty.onyx.xml"),  
    $vars := getDOS( $d, "onyx.ast.Variable" )  
for $v in $vars  
    return string($v)
```

```
<Result>$node $tagname $node $node $node $child $tagname  
$node $node $child $tagname $child $players $player $player  
$player $position $surname $textcontent $throws $throws $throws  
$tn</Result>
```

- Extract unique names
- Check that the builtins use compatible types
- Do Type inferencing

Examining the AST with Onyx

- Extract all the names of the function calls
(parser output public A2 test stats98lefty.onyx)

```
let $d := document("stats98lefty.onyx.xml"),
```

```
    $names := getDOS( $d, "onyx.ast.FunctionCall")
```

```
    for $n in $names
```

```
        let $a := getAttrEnv($n)
```

```
            return string(getAttributeValue($a,"name"))
```

```
<Result> tagname isENode children getDescendantOrSelfByTagName  
isENode children tagname getDescendantOrSelfByTagName document  
getChildrenByTagName getChildrenByTagName getChildrenByTagName  
string first string first tnode attrenv length isTNode first string  
first</Result>
```

- Extract unique names?

Type inferencing on the AST

- For each AST node type, check that the arguments are type compatible with the node's requirements $1 + 2*3$

```
<onyx.ast.Query>
```

```
  <onyx.ast.ExprList>
```

```
    <onyx.ast.Operator name="op:numeric-add">
```

```
      <onyx.ast.Constant datatype="onyx.types.Integer">1</onyx.ast.Constant>
```

```
      <onyx.ast.Operator name="op:numeric-multiply">
```

```
        <onyx.ast.Constant datatype="onyx.types.Integer">2</onyx.ast.Constant>
```

```
        <onyx.ast.Constant datatype="onyx.types.Integer">3</onyx.ast.Constant>
```

```
      </onyx.ast.Operator>
```

```
    </onyx.ast.Operator>
```

```
  </onyx.ast.ExprList>
```

```
</onyx.ast.Query>
```