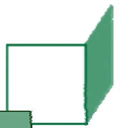
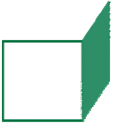
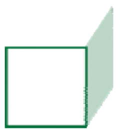
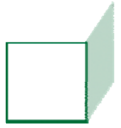


Chapter 2

XPath







OBJECTIVES

After completing "XPath," you will be able to:

- Describe the role of XPath in the grander scheme of XML transformation and document processing.
- Construct well-formed XPath expressions using verbose, formal syntax.
- Construct well-formed XPath expressions using abbreviated syntax.
- Write XPath expressions to select information from an XML document: everything from simple "paths" to specific queries based on multiple criteria.
- Use aggregate functions to perform calculations over an entire document, such as getting counts and sums.

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Addressing XML Content

- **Many XML applications need to address elements of XML content.**
  - This is not to be confused with **parsing**, which is a way of converting an XML document to its abstract content model.
  - Neither is addressing the same as reading or writing information. The **Document Object Model** offers one way to represent content nodes as in-memory objects, and to read and manipulate XML information.
  - Addressing is more basic: we're just concerned now with finding a way to uniquely, accurately, and usefully identify one or more pieces of information in an XML source.
- **We'd like to be able to define desired content – for whatever purpose – and have that definition applied to extract the information we need from a particular XML document.**
  - This may be a single value, or a list or set of XML **nodes**:

Give me a list of all account IDs.

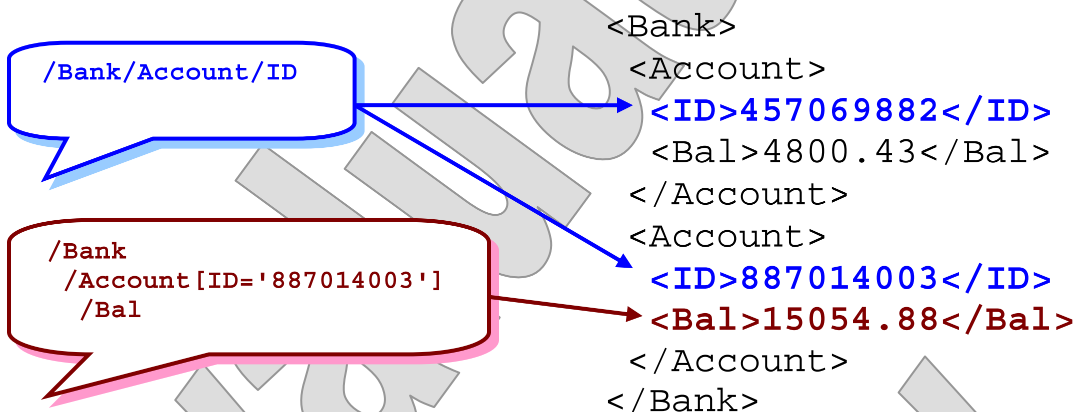
Give me the balance of account 887014003.

```
<Bank>
 <Account>
  <ID>457069882</ID>
  <Bal>4800.43</Bal>
 </Account>
 <Account>
  <ID>887014003</ID>
  <Bal>15054.88</Bal>
 </Account>
</Bank>
```

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

XPath

- The **XPath** specification (version 1.0 at the moment, with a 2.0 on the way) is recommended by the W3C as a standard means of addressing XML content.
  - It defines an expression-based grammar for addresses.
  - It defines the interpretation of this grammar, but some enclosing technology such as XSLT implements the translation from address to value or **node set**.



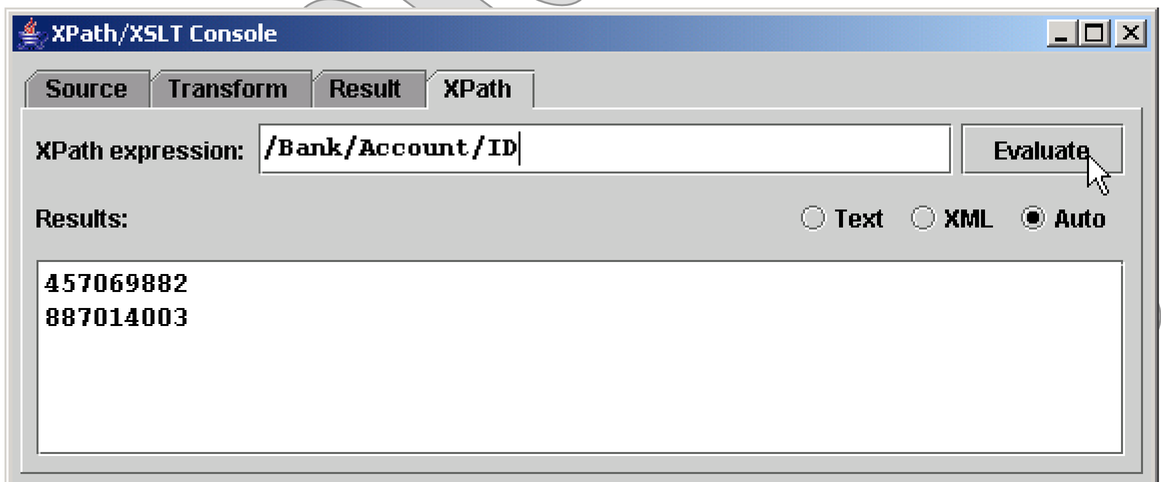
- XPath is then a basic technology, and as such it is integrated into many other W3C specifications:
  - We'll focus in this module on use of XPath in **XSLT**.
  - The **XPointer** specification requires XPath expressions to specify ranges or fragments of information in documents.
  - The **DOM** itself uses XPath in its more advanced levels.
  - **XML Query** is built on the capabilities of XPath.
  - **XML Schema** uses XPath expressions to define unique keys and key references.

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Evaluation Only

The XPath Console

- In the previous chapter we saw that we can use the **console** application to test XSLT transforms.
- The tool also provides a simple means of testing XPath expressions.
- From the working directory **Examples\Bank**, open the console application, and load **Bank.xml**.
- Hit **Ctrl-Page-Up**, which would ordinarily go “up” to the previous tab, but in this case will wrap around to select the **XPath** tab on the far right.
- Enter the first expression on the previous page, and see the evaluated expression shown in the text view:



- Try out the second expression as well, and keep the console running through this chapter’s lecture, to try out the expressions shown as examples of various XPath techniques.

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Using the XPath Console

- **There are two possible output forms for XPath expressions in this console:**
  - **Text**, showing only the text of selected nodes
  - **XML**, showing the XML representation of the nodes themselves, possibly including child nodes, attributes, etc.
- **You can also let the console auto-select between these – this is the default choice when you start up the application.**
- **The heuristic for this selection is as follows:**
  - If the nodes in the set have child elements, they will be shown in full XML view.
  - If they do not, they will be shown as plain text.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Using the XPath Console

- **Try out the different options for:**
  - "/Bank/Account" – the auto-selected view will be XML, because there's an interesting tree of content to show:

```
<Account>
  <ID>457069882</ID>
  <Bal>4800.43</Bal>
 </Account><Account>
  <ID>887014003</ID>
  <Bal>15054.88</Bal>
 </Account>
```

  - … but you can render just the text if you like:

```
  457069882
  4800.43

  887014003
  15054.88
```

  - "/Bank/Account/ID" – this will be text by default:

```
457069882
887014003
```

  - … but you can see the markup if you like:

```
<ID>457069882</ID>
<ID>887014003</ID>
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

XPath in Transformations

- **As we've seen, XSLT uses XPath in two main ways:**

```
<xsl:template match="//Car" >
  Found a car, model year is
    <xsl:value-of select="Year" />.
</xsl:template>
```

  - The **xsl:template** element defines the desired source information in its **match** attribute as an XPath expression.
  - The template will produce its output once for each node found in the source document that fits the XPath expression. Above, the template writes a value derived from the matched element, using **xsl:value-of** and its **select** attribute.

- **There are slight consequences on syntax when writing XPath expressions inside XSLT stylesheets:**
  - Typically XPath expressions will be XML attribute values; this constrains the use of single or double quotation marks, since those used in the XPath expression must not be the style used to enclose the whole expression.

```
<xsl:template match="//Car[Make="AMC"]" >
   ... is not well-formed XML!
```

  - The less-than operator cannot be used literally, as it would be mistaken as the beginning of markup.

```
<xsl:template match="//Car[Price &lt; 1000]" >
  ... is correct; using < directly would fail
```

- **These restrictions affect the XPath console, since it uses an XSLT transform to produce its results.**

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The XML InfoSet

- **The XML InfoSet is the single authoritative specification of the allowable content in an XML document, and of how that content can be structured.**
  - This is a useful reference, especially with so many nearly-identical definitions flying around.
  - The key idea in the InfoSet specification is that an XML document is merely a textual representation, based in a precise grammar, of information.
  - The fact that the grammar is so precise makes it tempting to think that the XML grammar defines the rules for XML content. It does not, although it must be in sync with those rules.
  - Consider one redundancy in the XML grammar: an empty element can be represented by either a start-tag-end-tag sequence, or by a single empty-element tag.
  - The InfoSet spec clarifies that these two syntaxes cannot be overloaded to mean different things as a legal extension of XML. This may seem obvious; yet it had not been written down anywhere until the InfoSet specification came along!
- **To avoid confusion between the XML document itself and the pure document content, developers – and other specification authors – are increasingly referring to a document's content as "an infoset".**

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

XPath Tree Structure

- **Like many other XML models, XPath views the content of an XML document as a node tree.**
- **The tree starts (naturally!) with a root node.**
  - This is not the root element! In fact, the root element – or, as we'll call it for minimal ambiguity, the **document element** – is actually the sole child of the root node.
  - From there, child elements are child nodes in the tree.
  - Note that attributes are nodes of their own, with the declaring element as parent. However, attribute nodes are not considered children of the parent element. This little paradox is one departure from the DOM tree view.
  - Also, the text for an element or attribute is considered a separate child node, although in XPath this is seldom apparent because XPath parsers will automatically convert a node to its string form, which is the value of that text node.

Evaluation Only

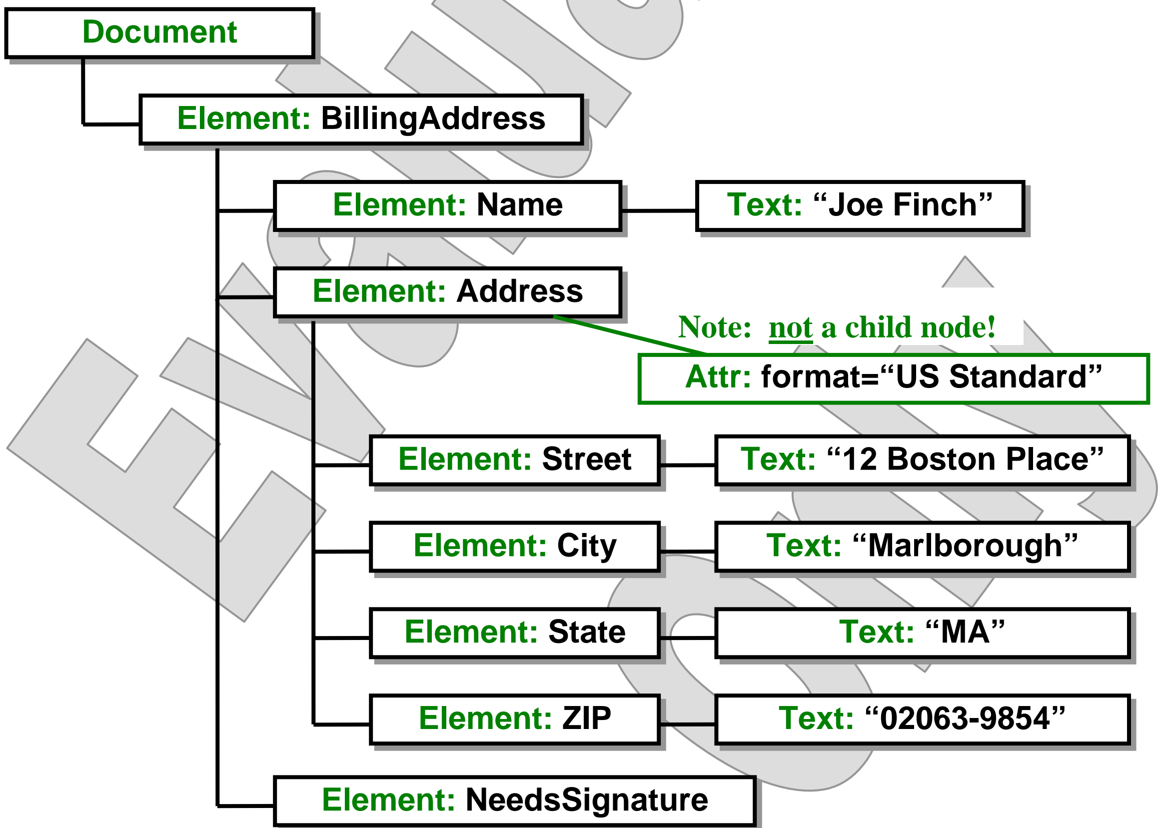
© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

A Simple Tree

EXAMPLE

- In **Examples\Billing\Step1** there is a simple XML document; the corresponding XPath tree view is shown below:

```
<BillingAddress>
  <Name>Joe Finch</Name>
  <Address format="US Standard">
    <Street>12 Boston Place</Street>
    <City>Marlborough</City>
    <State>MA</State>
    <ZIP>02063-9854</ZIP>
  </Address>
  <NeedsSignature />
</BillingAddress>
```



© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Document Order

- **For purposes of evaluating an XPath expression, the XPath tree is also understood as a series of nodes, in a precise document order.**
  - This order can be found by walking the tree from the root node down to leaf nodes, moving to children before siblings.
  - An easy way to see this ordering is to recognize that it matches the visual layout of the XML document exactly. Accordingly, attributes appear before children.
  - So, for the example on the previous page, the ordering would be:

```
Root
BillingAddress
Name
"Joe Finch"
Address
Format
"US Standard"
Street
"12 Boston Place"
City
"Marlborough"
State
"MA"
ZIP
"02063-9854"
NeedsSignature
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

XPath Expressions

- **The core construct in XPath is the expression.**
  - An expression is a string of symbols that can be evaluated as some piece of XML content.
  - Expressions can be nested in other expressions – we'll look at this more carefully a little later.
- **Every XPath expression evaluates to one of the following types:**
  - Node Set (which may contain one or many nodes, or may be an empty set)
  - String
  - Number (floating-point decimal)
  - Boolean
- **An expression of one type can be converted, implicitly, to any other type, by various rules that we'll discover as we go through this chapter.**
- **One can also explicitly convert using an XPath function named for the desired type, as in:**

```
boolean (someNodeSet)
string (someNumber)
number (someString)
node-set (anyPrimitiveValue)
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Context

- **All expressions, including location paths, are evaluated based on an understanding of context.**
- **The context includes several features that may be brought into play by various types of expressions:**
  - The c**ontext node** is the starting node for location paths, and the node on which various XPath functions will operate.
  - **Context position** describes the **proximity position** of the context node – typically this means its position amonst its siblings. Proximity position is discussed later in this chapter.
  - The **context size** is maximum value of proximity position – hence the total number of siblings-plus-self.
  - Variable bindings, available library functions, and namespaces in force are also part of the context. These especially are defined by XPath but provided at a higher level such as the XSLT engine or XPointer evaluator.
- **Context can change when an expression A evaluates another expression B contained within A.**
  - The context node is changed by many sorts of expressions. For instance, location paths internally work from the context node through several intermediates to a new node or node set.
  - Position and size can only be changed by predicate expressions – we'll see how this can happen in later examples.

Evaluation only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

XPath Grammar, From the Top

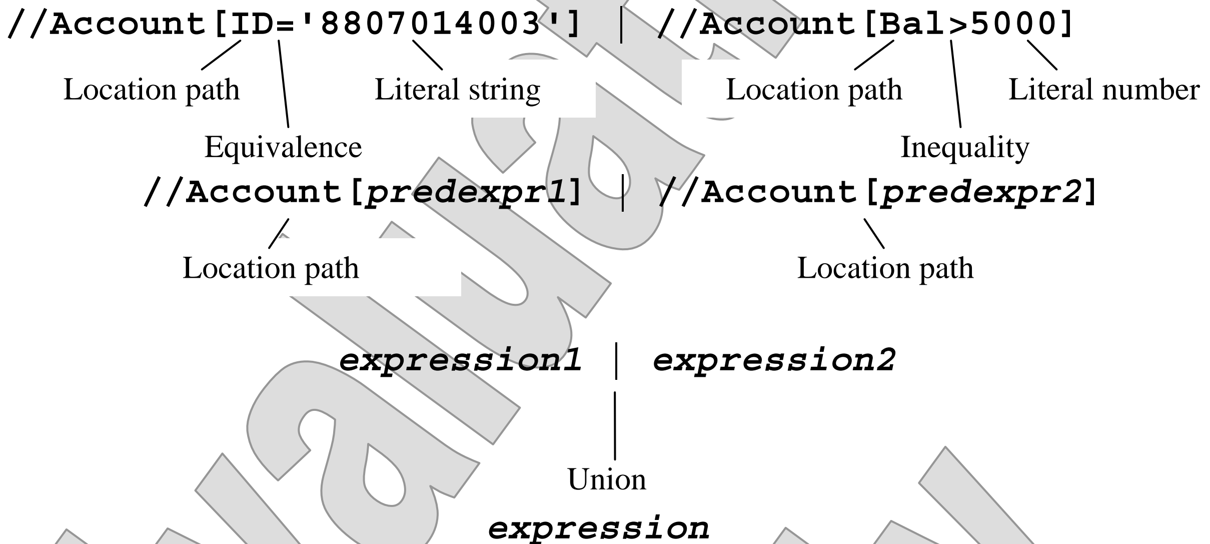
- **An XPath address is always a valid XPath expression.**
- **Expressions can be formed from other expressions by a number of means:**
  - Logical operators `and` and `or`, and the negating function `not`
  - Arithmetic operators `+`, `-`, `*`, `div`, and `mod`
  - Comparisons `=`, `!=`, `<`, `>`, `<=`, and `>=`
  - Unions of node sets using |
  - Appending one or more **predicates** to an expression
- **None of the above techniques addresses information, in and of itself.**
- **The key expression type is the location path, which does directly address information.**
  - Location paths can be combined into complex expressions.
  - Location paths are composed of one or more **location steps**.
  - Expressions can also be included as parts of location steps, via the location steps' **predicates**.
  - Note that the predicate is an element that is used both inside location paths and in general expression construction.
- **The other fundamental expression type is the literal expression, such as a raw number or quoted string.**

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Decomposing an Expression

- **Consider the following example of a complex expression.**
  - It is progressively built up from its constituent parts, which are labeled by their expression types.
  - The progression below describes the process of evaluating the expression, working from part to whole.



- Note however that before a constituent expression can be evaluated, its **context** must be understood, which means that the evaluating process must actually work from the top to the bottom and then back again.
- For instance, "Bal>5000" would be meaningless at the top level of the document; it is only useful in the context of `Account` elements.

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Location Paths

- **The most common type of expression is the location path, which navigates the XPath tree from one point to another. Location paths always return node sets.**
  - An **absolute location path** starts with a slash character and thus indicates that it will navigate from the root node.
  - A **relative location path** starts with a **location step** and navigates from a node defined by the expression's **evaluation context**.
  - In either case, a location path is then composed of one or more location steps, separated by slashes. Each location step navigates from the node reached by the previous one.
- **The common syntax for location paths is modeled on the UNIX syntax for file and directory paths.**
  - Hence the slashes separating steps, and the leading slash to indicate an absolute path.
  - Simple location paths look very much like file paths.
- **There is a more formal and exacting syntax on which this common one is based.**
  - Each location step is formed according to the pattern `axis::node-test[predicate-expr]`.
  - We'll look at the formal syntax now, and then rediscover the more common syntax by applying a combination of default values and abbreviations for elements of the formal syntax.

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Evaluation Only

Axis, Node Test, and Predicate

- **Whether stated explicitly or implied by the UNIX-style syntax, each location step includes these three components:**
  - Axis
  - Node test
  - Predicate
- **Each of these is expressed differently and works in a different way.**
- **What they have in common is that they each qualify the definition of addressed content.**
- **Think of the entire document tree as a set of candidate nodes from which the location step is going to select zero, one, or more.**
  - Each part of the location step limits this set by setting criteria.
  - Each is governed by its **context**, which is understood prior to evaluation for a given location step, and then progressively qualified by axis and node test.
  - The intersection of the set of all document nodes and the three sets defined by axis, node test and predicate forms the result of evaluating the location step.

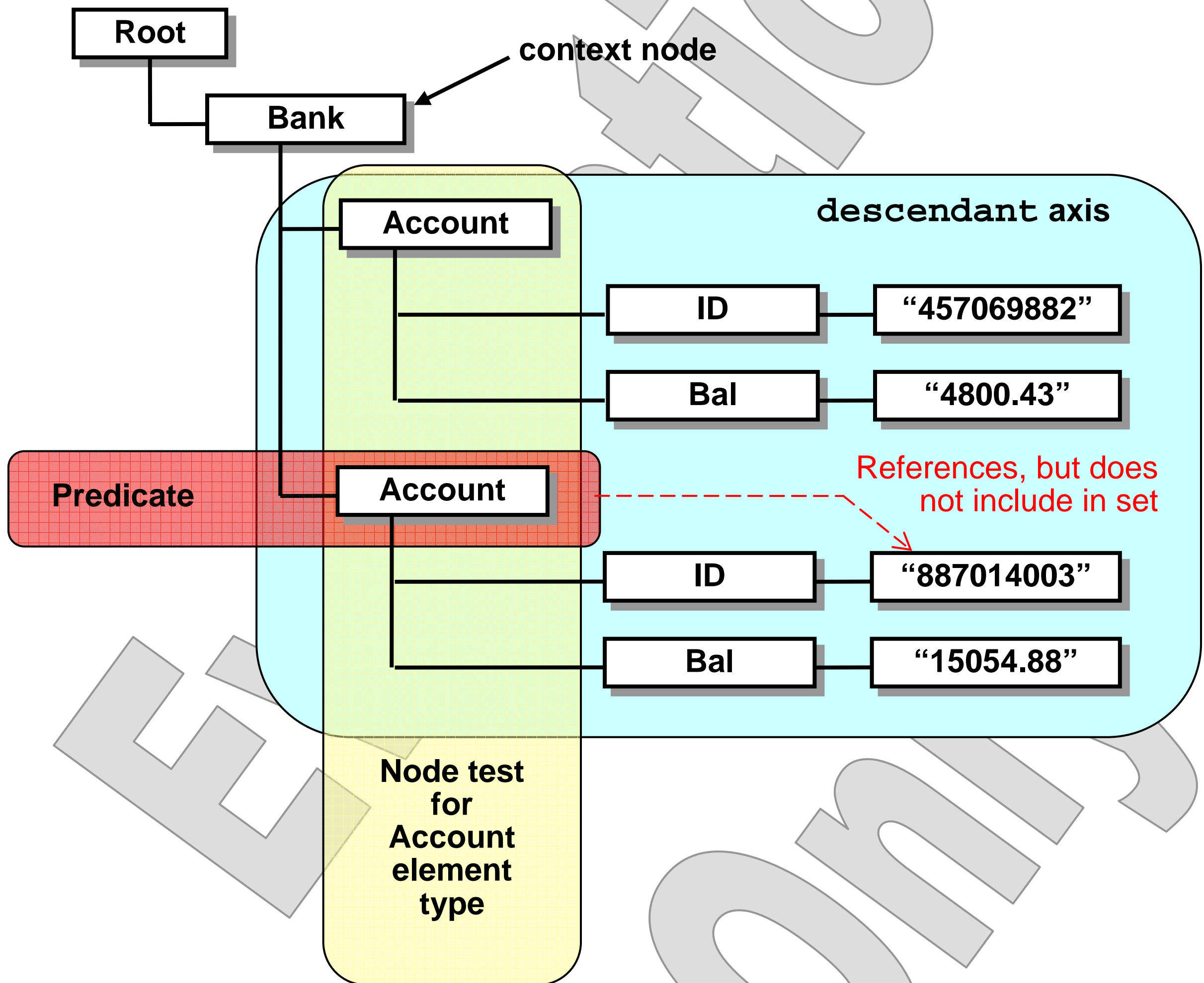
Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Finding the Bank Balance

EXAMPLE

- Let's say we need to evaluate the location step `"descendant::Account[ID='887014003']"` in the context of the `<Bank>` node.
  - Each part of the location step homes in on the desired information:



Note that if there were nodes outside the axis that would have met the node test or predicate conditions, they would have ultimately been excluded.

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Axis

```
descendant::Account[ID='887014003']
```

- **The axis in a location step qualifies the possible results of navigating that step by identifying a subset of the whole infoset, based on the starting node's position in the tree.**
- **The possible axis values are enumerated by the XPath specification. The most common and important axes are:**
  - **self**, reducing the possible results of the location step to the current node (or null, no nodes at all)
  - **child**, identifying direct child elements of the current node for testing
  - **descendant**, testing all children and their children, and so on
  - **parent**, testing only the parent node
  - **ancestor**, testing the parent and its parent, and so on to the root node
  - **attribute**, testing all attributes of an element node (note that none of the above axes would include attribute nodes, except possibly **self** if the starting node were an attribute)
  - **descendant-or-self**, **ancestor-or-self**
- **There are others more obscure, many having to do with testing siblings and elements before or after the starting node in the document order.**

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Node Test

```
descendant::Account[ID='887014003']
```

- **The node test selects certain nodes from the axis in one of a few basic ways:**
  - A **name test**, which selects nodes by element type
  - A **node type test**, which selects nodes by node type, that is, attributes, text nodes, namespace nodes, etc.
  - A test for processing instructions of a certain name
- **Name tests are by far the most common.**
- **In the Billing.xml example, the following location step would select all children of element type City:**

```
child::City            (try //Address/child::City)
```

- **This location step would select either the starting node or an empty node set, depending on whether the starting node were an attribute node:**

```
self::attribute()
```

- **Here's a test that always returns the starting node – this may seem pointless, but there must be a node test for each location step, so sometimes this is needed to enable other components, such as predicates:**

```
self::node ()
```

Evaluation Only

The Predicate

```
descendant::Account[ID='887014003']
```

- **The predicate is a final qualifier in the location step. It consists of a bracketed predicate expression, which is any XPath expression, evaluated as a boolean.**
- **Each node in the set defined by the axis and node test is tested against this predicate, and is retained in the set only if the predicate tests true.**
- **Note that all XPath expressions can be converted to boolean type:**
  - A node set is converted to **true** if and only if it is non-empty.
  - A string is converted to **true** if and only if its length is greater than zero.
  - A non-zero number (except any not-a-number constant) is converted to **true**; zero is converted to **false**.
- **Here's a location step that selects all children of type Address with an attribute format:**

```
child::Address[attribute::format]
```

  - Note the nesting of a location step (as a relative location path) as a predicate expression. This is very common.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Abbreviations

- **Using all the formal syntax for axes, node tests, and predicates, location paths can become quite unwieldy.**
- **Several abbreviations are available to simplify location paths and to make them more readable.**
  - If no axis is specified, **child** is the default.
  - The **attribute** axis and following double-colon can be shorthanded with the `@` symbol.
  - The **descendant-or-self** axis and following double-colon can be shorthanded with a double-slash. This can be used in lieu of a single slash either in the midst of a relative path or at the beginning of an absolute path.
  - The starting node can be represented with a dot.
  - The parent node can be represented with a double-dot.
  - The wildcard `*` can be used for the `node ()` test that selects all nodes.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Using Abbreviations

- **Here then are abbreviated versions of the examples on the previous pages:**

Complete expression	Abbreviated expression
`child::City`	`City`
`child::node ()`	`*`
`self::node ()`	`.`
`self::text ()`	`text ()`
`child::node ()/ attribute::node ()`	`*/@*`
`child::Address [attribute::format = "formal"]`	`Address [@format="formal"]`

- **Here are all ZIP codes of US standard addresses:**

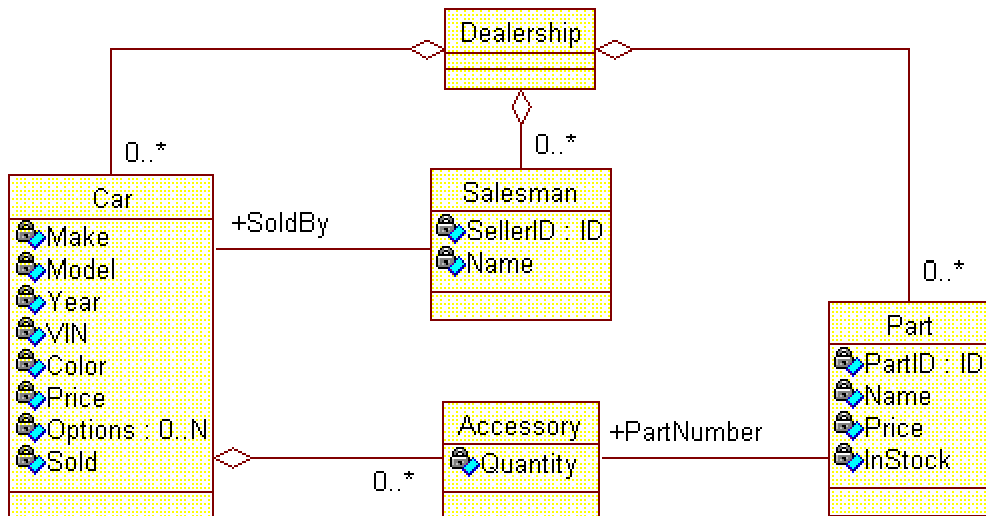
```
//Address[@format="US Standard"]/ZIP
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Car Dealership Case Study

- Along with the **Bank.xml** and **Billing.xml** source documents, we'll now begin to use a more complex model for our XPath exercises.
- This is Car Dealership case study, which models the current inventory and sales status of a small car dealership.



- Find the following files in **Examples\Cars\XPath\Step1** – we'll also encounter them in many other locations throughout the course:
  - **OnTheLot.xml** lists several cars, each with a year, make, model, vehicle identification number, color, and price, and attributes that describe a car as sold and sold by a certain salesman. Parts and salesmen are also listed.
  - **Cars.dtd** describes the exact vocabulary.

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Evaluation Only

Addressing OnTheLot.xml

EXAMPLE

- In the XPath console, load the document **OnTheLot.xml** from **Examples\Cars\XPath\Step1**.
- Try the following expressions as a way of experimenting with the axis/node-test/predicate and abbreviated grammars discussed thus far:

```
/descendant-or-self::Car[child::Make='Hyundai']
//Car[Make='Hyundai']

//Car/child::Model
//Car/Model

//Car/child::*/attribute::*
//Car/*/@*

//Car[attribute::Sold='TRUE']
//Car[@Sold='TRUE']
//Car[@Sold]
```

  - Why do the last three expressions all produce the same results?
  - They are not actually identical in meaning, as the previous pairs are.
  - The last expression finds `<Car>`s with <u>any</u> **Sold** attribute, whatever its value.
  - As it happens, the source document doesn't have any **Sold** attributes with a value other than "TRUE", so the results are the same in this case.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Absolute vs. Relative Location Paths

- **At this point it bears repeating that there location paths can be absolute (beginning with a slash character) or relative (beginning with a location step).**
- **In fact there are three common approaches, which, most simply, are two slashes, one slash, or no slashes.**
  - The double-slash at the beginning indicates an absolute path but also the descendant-or-self axis.
  - **"//A"** will select any and all `<A>` elements in the document.
  - **"/A"** will select an `<A>` existing as the document element, or will select nothing at all.
  - **"A"** will select any `<A>` elements as direct children of the context node.
- **Once we start working in XSLT, the relative location path will become much more common.**
  - Typically one uses absolute paths to find things, such as when applying XSLT templates.
  - Then relative paths are used within templates, where often one wants to work from the context node – which often is not the root node.
  - In fact the XPath for a template's **match** attribute is not really used to address any content, if you think about it; it is rather a statement of conditions for matching, known in XSLT as a **pattern**. Patterns are almost always relative location paths.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Nesting Expressions

- **Again, the location path is just one type of expression.**
- **Within location steps, predicates are based on predicate expressions, which can include:**
  - Other location paths
  - Calls to XPath **functions**, or to functions made available by an enclosing technology such as XSLT
  - Type conversions
  - Additional predicates
- **Note that a location step can only include one predicate; however, once a location path has been evaluated as an expression, additional predicates can be applied to the resulting node set.**
- **This is a convenient and useful feature.**
- **It can also make trouble.**

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Proximity Position

- A node's **proximity position** is an integer describing its position in a node set and relative to an axis.
- The `position` function can be used to derive this value.
  - This can only be used in a predicate expression or sub-expression.
  - The position value is 1-based.

```
//Car[position () = 2] produces
  <Car>
    <Make>Ford</Make>
    <Model>Pinto</Model>
    <Year>1974</Year>
    <VIN>AR7993</VIN>
    <Color>Dust</Color>
    <Price>0.99</Price>
  </Car>
```

  - In a predicate, a literal number where a boolean is expected will be evaluated as a position, so the following expression is the same as the previous one:

```
//Car[2]
```

- The `last` function gives the maximum possible position; that is, it returns the context size.
  - Here is a location path that addresses the last child of each `<Car>` element – a mixed bag of prices and options:

```
//Car/*[last()]
```

Evaluation Only

Boolean Operators

- **Boolean expressions can be formed in several ways:**
  - Conversion from other types, such as testing that a node set is not empty – this often occurs implicitly, as in predicate evaluation, but you can force such a conversion using the `boolean` function
  - Comparing values explicitly for equivalence or inequality
  - Combining other boolean expressions using logical operators
  - Using the XPath functions `true` and `false`
- **The logical operators are `and` and `or`; negation of an expression's value is handled by a function, `not`.**

```
//Car[@Sold="TRUE"]/@SellerID
//Car[Price > 1000]
//Car[Price > 1000 or Year > 1980]
```

- **The order of precedence is as follows:**
  - Inequality expressions (`a < b`, `c >= d`)
  - Equivalence expressions (`a = b`, `c != d`)
  - `and`
  - `or`

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Comparing Primitive Values

- **Booleans, numbers, and strings can be tested for equivalence implicitly, relying on built-in type conversions.**

  When evaluating a comparison, the XPath application will look for either of the expressions to be boolean, in which case it will convert the other to boolean, if necessary.

  If neither is boolean, the application will determine if either is a number, in which case it will convert the other to a number if necessary.

  The remaining possibility is that both are strings.

- **Comparisons are evaluated differently.**
  - Both arguments are converted to numbers and tested as such.
  - Boolean `true` is converted to 1; `false` to 0.
  - Strings are parsed for numeric value, including decimal points an unary `+/-` signs.

- **Some examples of boolean expressions that use comparisons and logical operators:**

```
//Car[Make='Ford']/Price
//Part[@InStock > 3 and Price > 50]
//Part[not (@InStock &lt; 4 or Price &lt;= 50)]
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Comparing Node Sets to Primitives

- **When a node set is involved in a comparison – either equivalence or inequality – the result is based on iteration over the values in the node set(s).**
- **A node set tested against a single value will be tested member-by-member, and the overall operation will evaluate to `true` if and only if it is true for at least one member of the set.**
  - Thus testing a set for equivalence to a string is the same as asking if that string exists somewhere in the set.
  - Here's an example expression that selects the name of each salesman who's sold at least one car – that is, whose ID is somewhere to be found in the set of IDs of sold cars:

```
//Salesman[@SellerID = //Car/@SoldBy]/Name
```

  - To test whether a node set "is less than" a number is to ask if there are any members which are less than that number.
  - Note that if the node set **HighAndLow** contains the numbers 1 and 10, then all of the following expressions evaluate to `true`:

```
HighAndLow < 5
HighAndLow > 5
5 < HighAndLow
```

XPATH 2.0

  - XPath 2.0 will provide a means of choosing between this "any" logic and "all" logic.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Comparing Two Node Sets

- **Comparing two node sets results in two nested iterations; again, a single `true` comparison will break those loops and return `true`.**
  - So two node sets are equivalent if at least one member can be found in each that is equivalent to the other; this is counterintuitive, but true nonetheless.
  - To see if there are any cars priced the same as any parts (there are not):

```
boolean (//Car/Price = //Part/Price)
```

  - Likewise, one node set "is less than" another node set if at least one member is less than at least one member of that other set.
  - Thus the set of car prices is both less than <u>and</u> greater than the set of part prices – clearly some of the usual properties of comparisons don't hold for node sets – and so these will both be true:

```
boolean (//Car/Price &lt; //Part/Price)
boolean (//Car/Price > //Part/Price)
```

- **To get "all" rather than "any" testing for inequalities, use the opposite comparison operator and the `not` function – or, again, wait for 2.0.**

```
      { 1, 2 } &lt; { 3, 4 }   = true – intuitive
      { 2, 3 } &lt; { 1, 4 }   = true - bizarre

not ( { 1, 2 }  >=  { 3, 4 } ) = true
not ( { 2, 3 }  >=  { 1, 4 } ) = false
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

XPath Functions

- We've discussed a couple of the functions in the XPath library:
  - **position** gives the proximity position of a given node.
  - **last** gives the context size.
- We won't explicitly consider each function in the XPath library, but some common useful functions are:
  - **count**, to get the number of nodes in a set (this is distinct from **last** mostly in its use and availability; **last** relies on context, while **count** takes a node set as an explicit parameter)

```
count (//Car) is 7
count (//Car[1]/*) is 6
count (//Option) is 3
count (//Car[count (Option) = 0]) is 5
```

  - **sum**, to get the sum of the numeric values in a node set

```
count (//Part/@InStock) is 5
sum (//Part/@InStock) is 47
```

  - **id**, to get the element of a given ID (based on the presence of an attribute whose type is defined as ID)

```
id ("Genghis")/Name = "Genghis Farrah"
```

XPATH 2.0

- Expect a greatly expanded function library in XPath 2.0, including **min/max**, **intersection/union**, and more powerful string manipulation.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Multiple Predicates

- **Remember that predicates can be applied as part of location steps or to any node-set expression.**
  - The governing axis and node-set are apparent in the location step.
  - The axis applied to determine position when a predicate is applied to a node-set expression is always the **child** axis, relative to the parent of each node in the set.
- **One common usage is to chain additional predicates to a single location step:**

```
axis::node-test[pred1][pred2]
```

- **Note that the first predicate will define the node set for the next, and so on, so the order of the predicates can make a big difference:**
  - The second car that's been sold is:

```
//Car[@Sold='TRUE'][2]
```

  - Switching the predicates yields nothing, as the second car (overall) has not been sold!

```
//Car[2][@Sold='TRUE']
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Experimenting with XPath

LAB 2A

In this lab you will experiment with the XPath syntax, creating simple to moderately complex expressions and trying them out against the Car Dealership XML document.

Detailed instructions are contained in the Lab 2A write-up at the end of the chapter.

Suggested time: 45-60 minutes.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Forward and Reverse Axes

- **Both the node set and the axis are implicit in the evaluation context for the position function.**
  - Note that XPath defines some axes as **forward** and some as **reverse**, based on the direction in which they carry the navigation from the context node.
  - Most axes are forward.
  - **parent**, **ancestor**, **ancestor-or-self**, and others that implicitly work back through the document order are reverse axes.
  - Position is calculated in the direction of the axis, so the direct parent has position = 1 on the **ancestor** axis. This is only sensible.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Changing Axis Direction

- **However, when chaining predicates, the axis in the location step only applies to the <u>first</u> predicate.**
  - The second predicate is applied to the resulting expression, using the implicit **child** axis.
  - This can lead to counterintuitive results if the location step uses a reverse axis.
  - For instance, consider this expression evaluated for the Car Dealership infoset – it gets the fifth car:

```
//Car[6]/preceding-sibling::Car[1]
```

  - If we add a predicate to the location step to get the car previous to number 6 that's been sold, we might try this:

```
//Car[6]/preceding-sibling::Car[@Sold='TRUE'][1]
```

  - This doesn't get the preceding sold car! The location step uses a reverse axis, but as soon as it's evaluated, it's a normal expression. The second predicate is applied to the <u>forward</u> axis for the resulting node set of sold cars, and we get the <u>first</u> sold car in the document.
  - Bearing this in mind, the correct expression would be:

```
//Car[6]
    /preceding-sibling::Car[@Sold="TRUE"][last ()]
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

String Manipulation

- **XPath provides some primitive string-manipulation functions. We'll introduce a few of them here.**
- **Try these expressions out using the XPath console:**
  - The **`string-length`** function gives the string length.

```
string-length (//Car[1]/Make)
3
```

  - The **`concat`** function will concatenate two strings, returning the new string.

```
concat (concat (//Car[1]/Make, ' '), //Car[1]/Model)
AMC Pacer
```

  - The **`substring`** function will return an indexed substring of a single source string. (The index is 1-based and the substring length is optional.)

```
substring (//Car[1]/Model, 2, 3)
ace
substring (//Car[1]/Model, 2)
acer
```

  - The **`translate`** function will perform character replacement in a string, based on a string of target characters and a string of replacement characters.

```
translate (//Car[1]/Make, 'EMMY', 'ABBA')
ABC
```

XPATH 2.0

- **Again, expect string manipulation to get more sophisticated in XPath 2.0.**

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

What About Numbers?

- **XPath supports integers, as we've seen in using the `count` and `sum` functions, and numbers can have floating-point values, too.**
- **Arithmetic operations are supported:**
  - `+`, `-`, `*`, `div` operators for basic arithmetic
  - A `mod` operator for modulus calculation
- **This is plenty for most operations.**
- **Formatted number output is not a strong suit of XPath.**
  - XSLT provides this through its own `format-number` function; we'll look at this in a later chapter.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

XPath and Namespaces

- **Thus far we've worked with XML documents whose elements and attributes are all in the null namespace.**
- **In practical application, XPath expressions will often be required to address information in a document in which one or more non-null namespaces are in play.**
- **Fortunately, XPath is entirely namespace-aware.**
  - Element and attribute names are used in location steps, and these names are **qualified names**, or **QNames**, in the parlance of the XML Namespaces recommendation.
  - A node test that states an unqualified name is testing for elements or attributes of that name in the null namespace.
  - To test for qualified names in the source document, use a qualified name in the node test. The prefixes don't have to match, so long as they are mapped to the same namespace URI in their respective documents.
  - One frustrating limitation is that XPath cannot use a default namespace. An unqualified name always references the null namespace, even if a default is declared for the transform or some other enclosing element.
  - This can make complex XPath expressions all the more error-prone and difficult to read, but there is no way around it. Declare very short prefixes for use with XPath!
- **This problem will be fixed in XPath 2.0.**

XPATH 2.0

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

XPath Repairman

LAB 2B

Optional

In this lab you will test your knowledge of the precise XPath syntax by fixing a few expressions that don't parse.

Detailed instructions are contained in the Lab 2B write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

SUMMARY

- While a bit curious at first, the XPath expression language is a good tool for addressing items of XML information.
- It is not perfect, by any means, and the 2.0 specification should fix some of the more glaring errors and downright strange behaviors.
- XPath is key to describing selections of XML nodes; it is impossible to do anything useful in XSLT or XPointer without XPath.
- It's worth noting that the overwhelming majority of XPath expressions one will ever write are supported by the simplest and most stable parts of the grammar.
- We've delved into some of the oddities of the syntax, and once in a while you may find yourself needing some of these techniques, but most expressions you'll ever need are intuitive to write and to read.

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Quick Reference – XPath Functions

- Here is a brief list of those functions – consult the XPath specification for complete documentation:

```
Node Set Functions:
number last ()
number position () <!-- 1-based in axis dir. -->
number count (node-set)
node-set id (object)
string local-name (node-set?)
string namespace (node-set?)
string name (node-set?)

String Functions:
string string (object)
string concat (string, string+)
boolean starts-with (string, string)
boolean contains (string, string)
string substring (string, number,  number?)
string substring-before (string, string)
string substring-after (string, string)
number string-length (string)
string normalize-space (string?)
string translate (string, string, string)

Boolean functions:
boolean boolean (object)
boolean true ()
boolean false ()
boolean lang (string)

Number functions:
number number (object)
number sum (node-set)
number floor (number)
number ceiling (number)
number round (number)
```

Evaluation Only

© 2002-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.