

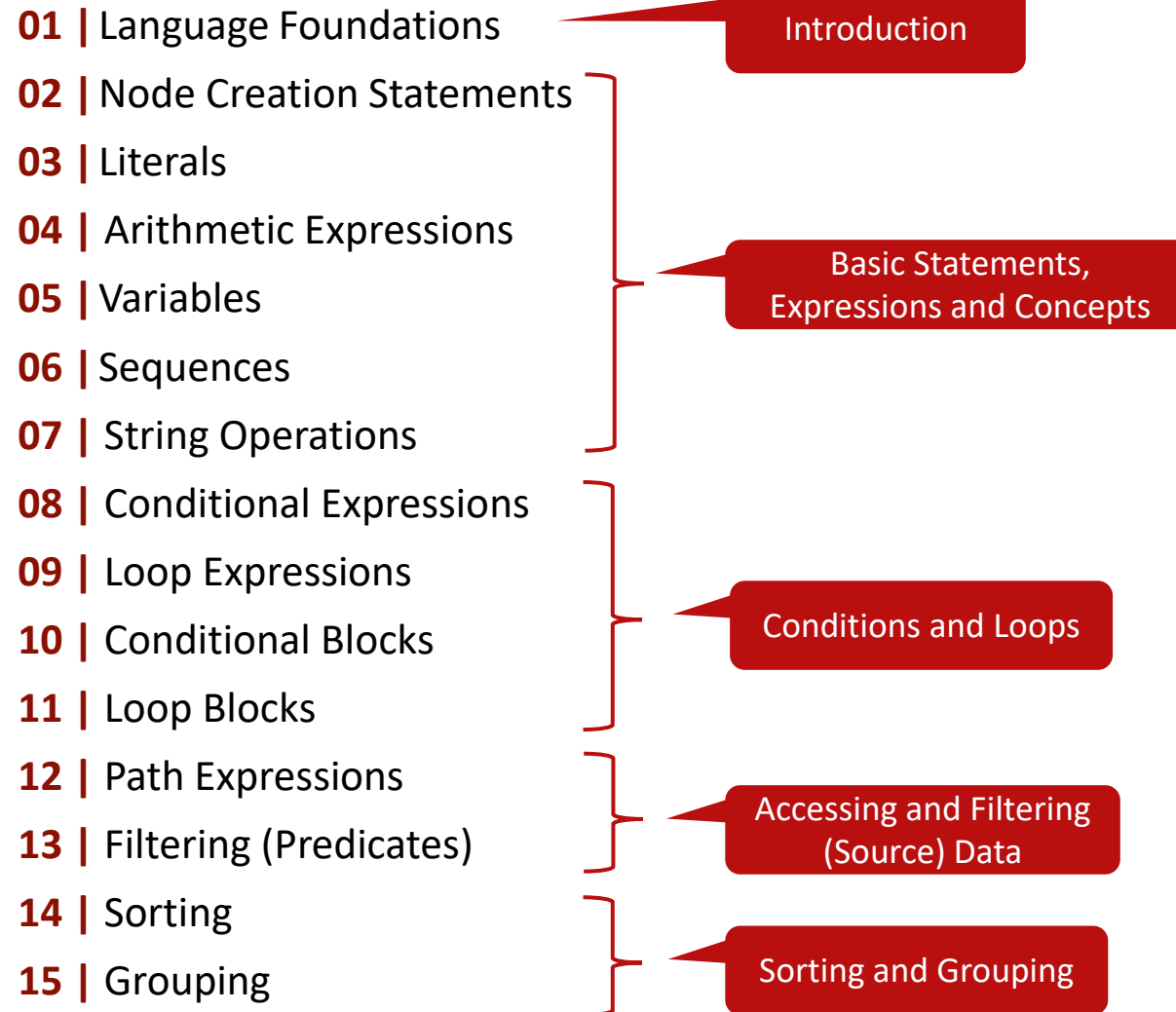
SEEBURGER



BIS Mapping Language Tutorial



Overview

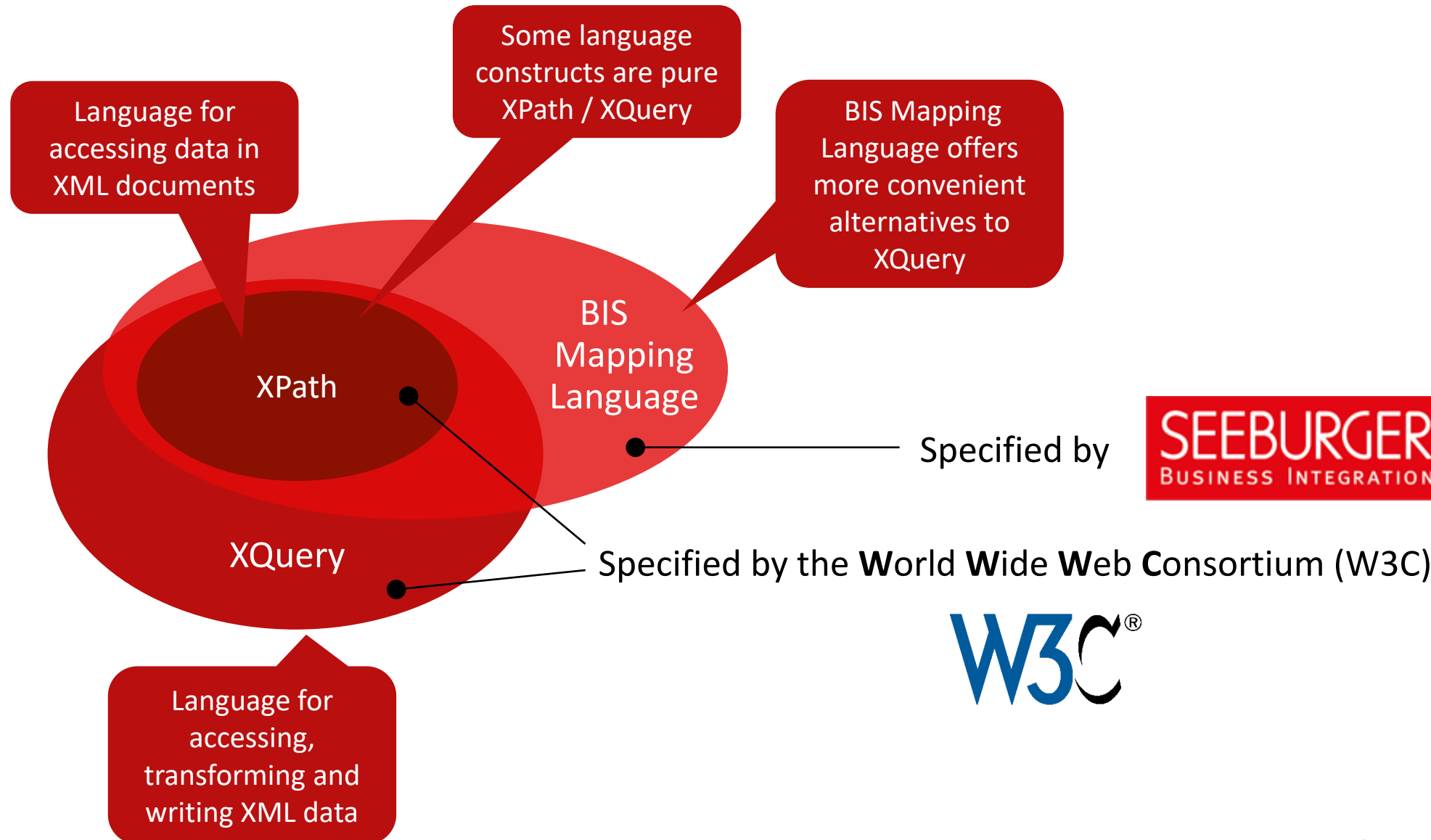




1

Language Foundations

Language Foundations





- BIS Mapping Language is based on **XPath** and **XQuery**
- BIS Mapping Language allows to transform many different formats, not just XML
- BIS Mapping Language offers **easier syntax** for common tasks



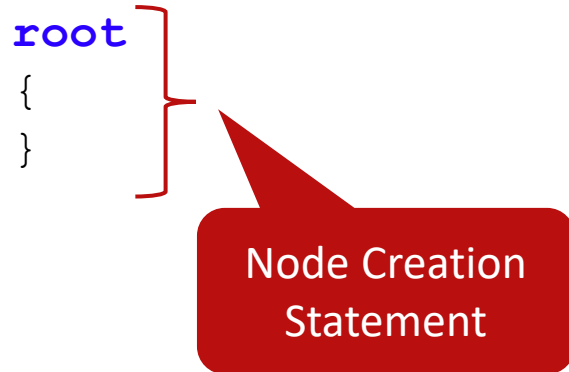


2

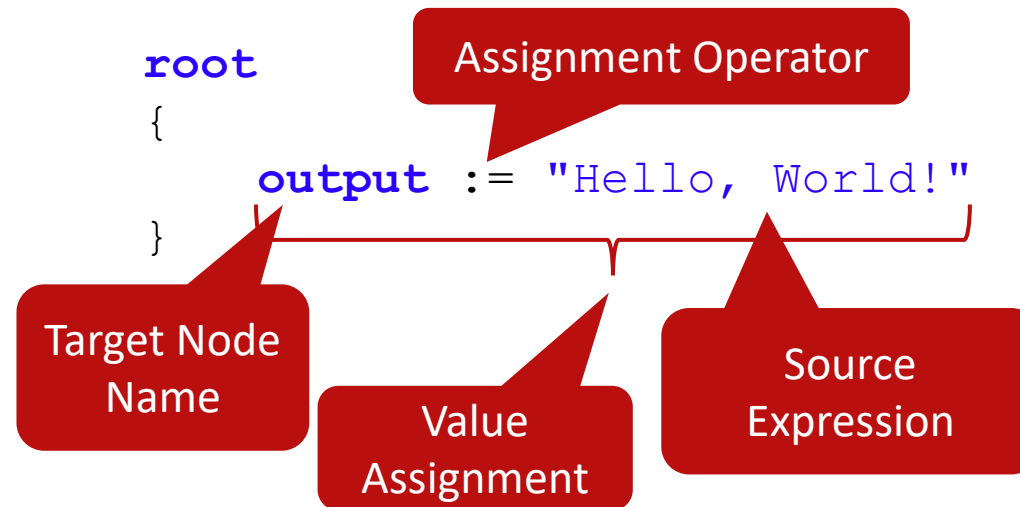
Node Creation Statements

Node Creation Statements

Creating Complex Nodes



Creating Leaf Nodes



Output

```
<root>  
  <output>Hello, World!</output>  
</root>
```

3

Literals

String Literals

Comments start with (: and end with :).
They can span multiple lines.

(: String literals can be enclosed in single or double quotes :)

myString := "Hi!"

Double quotes

myString := 'Hi!'

Single quotes

(: This is especially useful if the string content contains quotation marks :)

myString := "Let's go"

myString := 'Hamlet: "To be, or not to be, that is the question"'

Escaped single
quote

(: Quotation marks inside strings must be escaped with backslashes :)

myString := 'Bertrand Russell: "Science is what you know. Philosophy is what you don\'t know."'

Output

```
<root>
  <myString>Hi!</myString>
  <myString>Hi!</myString>
  <myString>Let's go</myString>
  <myString>Hamlet: "To be, or not to be, that is the question"</myString>
  <myString>Bertrand Russell: "Science is what you know. Philosophy is what you don't know."</myString>
</root>
```

Integer and Decimal Literals

```
(: Integer literals :)
myInteger := 42
myInteger := -23

(: Native support for big integers :)
myLargeInteger := 123987418753462384721379456189327645891264

(: Decimal literals :)
myDecimal := 3.1415
myDecimal := -6.8

(: Native support for big decimals :)
myLargeDecimal := 867168719871984687169875637841358713.681738735
```

Output

```
<root>
  <myInteger>42</myInteger>
  <myInteger>-23</myInteger>
  <myLargeInteger>123987418753462384721379456189327645891264</myLargeInteger>
  <myDecimal>3.1415</myDecimal>
  <myDecimal>-6.8</myDecimal>
  <myLargeDecimal>867168719871984687169875637841358713.681738735</myLargeDecimal>
</root>
```

Boolean Literals

```
root
{
  myBoolean := true()
  myBoolean := false()
}
```

Boolean literals need
the () suffix to be
distinguishable from
path expressions

Output

```
<root>
  <myBoolean>true</myBoolean>
  <myBoolean>false</myBoolean>
</root>
```

4

Arithmetic Expressions

Arithmetic Expressions

root

{

additionResult := 23 + 42

subtractionResult := 42 - 23

multiplicationResult := 23 * 42

divisionResult := 42 **div** 23

Division operator
is div, not /

integerDivisionResult := 42 **idiv** 23

moduloResult := 8 **mod** 3

Integer division cuts
off fraction digits

result1 := 2 * 3 + 4

result2 := 2 * (3 + 4)

Use parentheses to
prioritize
computations against
default operator
precedence

Remainder after
division

}

Output

<root>

<additionResult>**65**</additionResult>

<subtractionResult>**19**</subtractionResult>

<multiplicationResult>**966**</multiplicationResult>

<divisionResult>**1.8260869565**</divisionResult>

<integerDivisionResult>**1**</integerDivisionResult>

<moduloResult>**2**</moduloResult>

<result1>**10**</result1>

<result2>**14**</result2>

</root>

Rounding

```
root
{
  floor := floor(1.9)
  ceil  := ceil(1.1)
  round := round(1.5)
  advancedRound := round(1.5, 0, "HALF_DOWN")
}
```

Always rounds down

Always rounds up

Advanced round()
function takes
number of fraction
digits and rounding
mode

Output

```
<root>
  <floor>1</floor>
  <ceil>2</ceil>
  <round>2</round>
  <advancedRound>1</advancedRound>
</root>
```



5

Variables

Variable Declarations with Let Clauses

let \$x := 42

Assignment
Operator

Variable Names
are prefixed
with \$

Note: Each variable can
only be assigned
exactly once.



Variables cannot be re-assigned (they are **immutable**). The following operations are illegal in functional / declarative languages:

```
let $i := 42
$i := 23
$i := $i + 1
```



How can we aggregate data like sums?
How can we append strings consecutively, if we cannot re-assign variables?

Imperative vs. Functional Code

Imperative Pseudocode

```
$mySum = 0
$x = computeSomething()
$mySum = $mySum + $x
$y = computeSomethingElse()
$mySum = $mySum + $y
```

Imperative Pseudocode

```
$myString = ""
$x = computeString()
$myString = $myString + $x
$myString = $myString + " "
$y = computeOtherString()
$myString = $myString + $y
```

Functional Pseudocode

```
$x = computeSomething()
$y = computeSomethingElse()
$mySum = sum($x, $y)
OR
$mySum = $x + $y
```

Compute/collect
data/operands first, then
compute the result at the
end

Functional Pseudocode

```
$x = computeString()
$y = computeOtherString()
$myString = concat($x, " ", $y)
OR
$myString = $x + " " + $y
```

Compute/collect
data/operands first, then
compute the result at the
end

Variable Scopes

root

{

```
let $x := 42
```

Scope of \$x

element

{

```
myNumber := $x
```

```
let $y := "test"
```

```
myString := $y
```

Scope of \$y

}

```
rootLevelNumber := $x
```

}

Output

```
<root>
```

```
  <element>
```

```
    <myNumber>42</myNumber>
```

```
    <myString>test</myString>
```

```
  </element>
```

```
  <rootLevelNumber>42</rootLevelNumber>
```

```
</root>
```

6

Sequences

Sequences

Sequence
containing 3
numbers

```
let $myNumbers := (1, 2, 3)
```

```
numberCount := count($myNumbers)
```

count() returns the number of
elements in a sequence

```
firstNumber := $myNumbers[1]
```

```
number := $myNumbers
```

If a sequence is assigned
to a simple target node,
the node is repeated
automatically.

Elements in sequences can
be accessed with indices in
square brackets.
Note: all indices are 1-based,
not 0-based!

Output

```
<root>
  <numberCount>3</numberCount>
  <firstNumber>1</firstNumber>
  <number>1</number>
  <number>2</number>
  <number>3</number>
</root>
```

Empty Sequences

There is no `null` concept in XPath / XQuery, instead **empty sequences** are used

```
let $emptySequence := ()
```

```
emptySequenceElementCount := count($emptySequence)
```

```
emptySequenceOutput := $emptySequence
```

It can be configured whether nodes are created when empty sequences are assigned, or skipped completely

Output

```
<root>
  <emptySequenceElementCount>0</emptySequenceElementCount>
  <emptySequenceOutput/>
</root>
```

In this case the node was created with an empty value

Function with Sequence Parameters

```
let $myNumbers := (1, 2, 3)
```

```
sum := sum($myNumbers)
```

```
average := avg($myNumbers)
```

```
minimum := min($myNumbers)
```

```
maximum := max($myNumbers)
```

Some functions
take a sequence
as parameter

Output

```
<root>  
  <sum>6</sum>  
  <average>2</average>  
  <minimum>1</minimum>  
  <maximum>3</maximum>  
</root>
```



- Every expression evaluates to a **sequence** at runtime.
- Sequences can contain any number of elements
 - 0: empty sequence
 - 1: atomic value or node
 - n: sequence of values and/or nodes



7

String Operations

String Concatenation and Basic String Operations

root

Native String Concatenation with || Operator

```
concatenatedString := "See" || "burger"  
concatenatedString2 := concat("See", "burger")
```

Functional String Concatenation with concat() Function

```
trimmedString := trim(" input with spaces ")
```

```
left := left("Seeburger", 3)
```

First 3 characters

```
right := right("Seeburger", 6)
```

Last 6 characters

```
substring := substring("Seeburger", 4, 4)
```

```
replace := replace("Seeburger", "e", "3")
```

```
length := string-length("Seeburger")
```

Common mistake: don't use
length() for strings, use
string-length() instead.

Start index and length.
Remember: indices are 1-based

Output

```
<root>
```

```
<concatenatedString>Seeburger</concatenatedString>
```

```
<concatenatedString2>Seeburger</concatenatedString2>
```

```
<trimmedString>input with spaces</trimmedString>
```

```
<left>See</left>
```

```
<right>burger</right>
```

```
<substring>burg</substring>
```

```
<replace>S33burg3r</replace>
```

```
<length>9</length>
```

```
</root>
```

Splitting and Joining Strings

Decomposes the string and produces a sequence containing three strings: ("a", "b", "c")

```
root
{
  let $tokens := tokenize("a-b-c", "-")
  tokenCount := count($tokens)
  token := $tokens

  joinedString := string-join($tokens, ".")
}
```

Concatenates all elements in the sequence \$tokens, adding the specified delimiter between each element

Output

```
<root>
  <tokenCount>3</tokenCount>
  <token>a</token>
  <token>b</token>
  <token>c</token>
  <joinedString>a.b.c</joinedString>
</root>
```

8

Conditional Expressions

If Expressions and Comparison Expressions

```
root
{
```

```
  let $myNumber := 42
  numberComparisonResult := if ($myNumber < 100)
    then "smaller than 100"
    else "greater than or equal to 100"
```

```
  let $myString := "test"
  stringComparisonResult := if ($myString = "test")
    then "string is equal to 'test'"
    else "strings are not equal"
```

```
}
```

Output

```
<root>
  <numberComparisonResult>smaller than 100</numberComparisonResult>
  <stringComparisonResult>string is equal to 'test'</stringComparisonResult>
</root>
```

Condition
(in braces)

Comparison Operators:
< <= > >= = !=

If Expression

If expressions always return another expression, depending on the condition. The else part is mandatory.

Sequences as Comparison Operands

```
let $myString := "test"
stringToSequenceComparisonResult := if ($myString = ("test", "foo"))
    then "found a match"
    else "no match found"
```

XPath comparison semantics for sequences: all combinations are checked. If one of the combinations yields true, the whole comparison results is true.

Output

```
<root>
  <stringToSequenceComparisonResult>found a match</stringToSequenceComparisonResult>
</root>
```

Combining Multiple Conditions

```
let $myNumber := 42  
let $myString := "test"
```

```
multipleConditionsResult := if ($myNumber = 42 and $myString = "test")  
    then "both true"  
    else "at least one false"
```

Conditions can be combined
using the boolean operators
and / or

Output

```
<root>  
  <multipleConditionsResult>both true</multipleConditionsResult>  
</root>
```

9

Loop Expressions

For Expressions

root

```
{  
  let $strings := (" a ", " b ", " c ")  
  numberOfInputStrings := count($strings)  
  
  let $trimmedStrings := for $string in $strings return trim($string)  
  
  numberOfOutputStrings := count($trimmedStrings)  
  output := $trimmedStrings  
}
```

Note that all strings contain
leading and trailing
whitespace

Trims each string,
producing a new sequence
with the same element
count

Output

```
<root>  
  <numberOfInputStrings>3</numberOfInputStrings>  
  <numberOfOutputStrings>3</numberOfOutputStrings>  
  <output>a</output>  
  <output>b</output>  
  <output>c</output>  
</root>
```


10

Conditional Blocks

If Blocks

If block with `then` block and `else` block.
Both blocks are completely independent
and can contain arbitrary statements
(including nested if blocks).

```
root
{
  let $myNumber := 43
  if ($myNumber = 42)
  {
    result := "The answer to the ultimate question of life, the universe and everything"
    a := "foo"
  }
  else
  {
    result := "It's just a random number"

    let $string := "bar"
    b := $string
  }
}
```

Else
branch is
optional

Output

```
<root>
  <result>It's just a random number</result>
  <b>bar</b>
</root>
```

If Expressions vs. If Blocks

	If Expression	If Block
Returns an Expression	✓	✗
Mandatory Else Branch	✓	✗
Can Contain Statements	✗	✓
Has Curly Braces {}	✗	✓

If Expressions vs. If Blocks

```
root
{
  if ($myNumber = 42)
  {
    output := "special number"
  }
  else
  {
    output := "random number"
  }

  MORE COMPACT VERSION:

  output := if ($myNumber = 42)
    then "special number"
    else "random number"
}
```

If block that could be re-written as single assignment with if expression

More compact, equivalent code using a single assignment and an if expression

11

Loop Blocks

For Loop Blocks

```
root
{
  for $myNumber in (1, 2, 3)
  {
    number := $myNumber
  }
}
```

Loop over
sequence of
numbers

Output

```
<root>
  <number>1</number>
  <number>2</number>
  <number>3</number>
</root>
```

For Expressions vs. For Blocks

	For Expression	For Block
Returns an Expression	✓	✗
Mandatory <code>return</code> Clause	✓	✗
Can Contain Statements	✗	✓
Has Curly Braces <code>{}</code>	✗	✓

For Loop Blocks with Index Variable

`$i` does not have to be initialized – it is initialized implicitly in every loop iteration

Index variable provides the number of the current iteration

Loop over sequence of strings

```
root
{
  for $myString at $i in ("a", "b", "c")
  {
    index := $i
    string := $myString
  }
}
```

`$i` is 1 in the first iteration, 2 in the second iteration, etc.

Output

```
<root>
  <index>1</index>
  <string>a</string>
  <index>2</index>
  <string>b</string>
  <index>3</index>
  <string>c</string>
</root>
```


For Loop Blocks with Node Creation Statements

```
root
{
  for $myNumber in (1, 2, 3)
  {
    output
    {
      number := $myNumber
    }
  }
}
```

Creates an output node in each loop iteration

for is replaced with "create-for-each" operator <-

Equivalent, more compact loop

Target node name

```
root
{
  output <- $myNumber in (1, 2, 3)
  {
    number := $myNumber
  }
}
```

Output

```
<root>
  <output>
    <number>1</number>
  </output>
  <output>
    <number>2</number>
  </output>
  <output>
    <number>3</number>
  </output>
</root>
```

12

Path Expressions

Absolute Path Expressions

Absolute Path Expressions begin with a /

absolutePathOutput := /StoreSalesData/Shops/Shop/ShopAddress/City

Results in a sequence of City nodes

Semantics:

1. Starting at the root of the input message, select all nodes named StoreSalesData
2. From all resulting nodes, select all child nodes named Shops
3. From all resulting nodes, select all child nodes named Shop
4. From all resulting nodes, select all child nodes named ShopAddress
5. From all resulting nodes, select all child nodes named City

Output

```
<root>
  <absolutePathOutput>Berlin</absolutePathOutput>
  <absolutePathOutput>Paris</absolutePathOutput>
  <absolutePathOutput>New York</absolutePathOutput>
</root>
```

Relative Path Expressions

Parent loop provides the context for relative path expressions

```
root <- /StoreSalesData
{
  relativePathOutput := Shops/Shop/ShopAddress/City
}
```

Relative Path Expressions do not begin with a /

Results in a sequence of City nodes

Semantics:

1. Starting at the current loop context node `StoreSalesData`, select all child nodes named `Shops`
2. From all resulting nodes, select all child nodes named `Shop`
3. From all resulting nodes, select all child nodes named `ShopAddress`
4. From all resulting nodes, select all child nodes named `City`

Output

```
<root>
  <absolutePathOutput>Berlin</absolutePathOutput>
  <absolutePathOutput>Paris</absolutePathOutput>
  <absolutePathOutput>New York</absolutePathOutput>
</root>
```

Relative Path Expressions

```
root <- /StoreSalesData
{
  Stores
  {
    Store <- Shops/Shop
    {
      ID := @id
      Name := ShopAddress/Name
      City := ShopAddress/City
    }
  }
}
```

Parent loop provides the context for relative path expressions

3 relative path expressions

These relative path expressions will only select the nodes below the **current** Shop in the loop iteration

Output

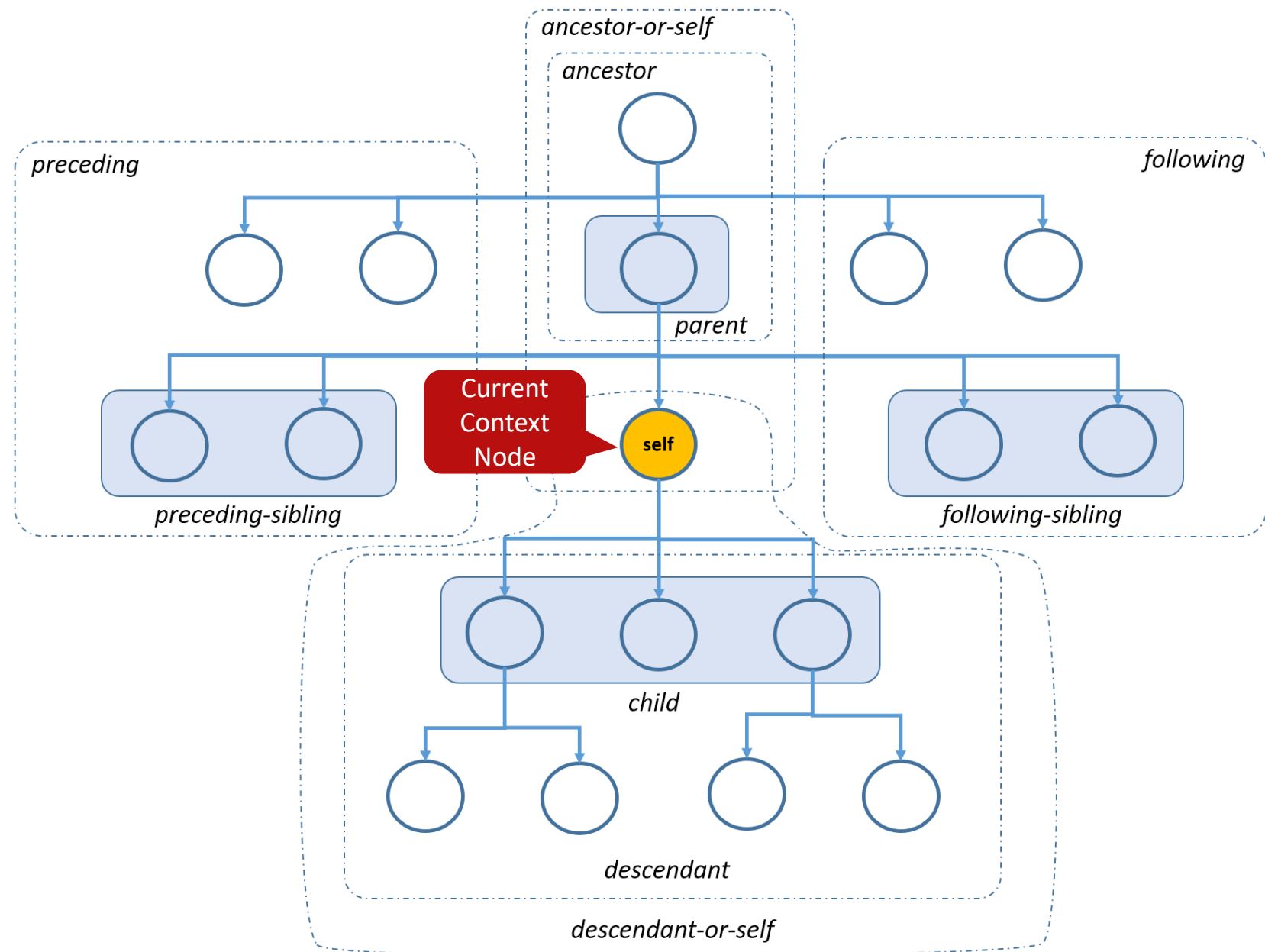
```
<root>
  <Stores>
    <Store>
      <ID>8852662</ID>
      <Name>Shop Floor Berlin</Name>
      <City>Berlin</City>
    </Store>
    <Store>
      <ID>000005</ID>
      <Name>Shop Floor Paris</Name>
      <City>Paris</City>
    </Store>
    <Store>
      <ID>888888881</ID>
      <Name>Shop Floor New York</Name>
      <City>New York</City>
    </Store>
  </Stores>
</root>
```



- Path expressions can be interpreted as nested loops
- Each step produces a new result sequence
- Each step builds upon all results of the previous step



XPath Axes



XPath Axes

```
root <- /StoreSalesData
```

```
{
```

```
  Stores
```

```
  {
```

```
    Store <- Shops/Shop
```

```
    {
```

```
      ID := @id
```

```
      ID2 := attribute::id
```

The two path expressions are equivalent

```
    for ShopAddress
```

```
    {
```

```
      Name := Name
```

```
      Name2 := child::Name
```

The two path expressions are equivalent

```
      City := City
```

Parent Node

```
      ShopId := ../@id
```

Selects parents, grandparents etc. with the name Shop

```
      AncestorShopId := ancestor::Shop/@id
```

```
    DescendantName := descendant::Name
```

Selects children, grandchildren etc. with the name Name

```
    FollowingSiblingCount := count(following-sibling::Shop)
```

```
    PrecedingSiblingCount := count(preceding-sibling::Shop)
```

Selects following/preceding siblings of the current Shop with the name Name

```
  }
```

```
}
```

```
}
```

Output

```
<root>
  <Stores>
    <Store>
      <ID>8852662</ID>
      <ID2>8852662</ID2>
      <Name>Shop Floor Berlin</Name>
      <Name2>Shop Floor Berlin</Name2>
      <City>Berlin</City>
      <ShopId>8852662</ShopId>
      <AncestorShopId>8852662</AncestorShopId>
      <DescendantName>Shop Floor Berlin</DescendantName>
      <FollowingSiblingCount>2</FollowingSiblingCount>
      <PrecedingSiblingCount>0</PrecedingSiblingCount>
    </Store>
    <Store>
      <ID>000005</ID>
      <ID2>000005</ID2>
      <Name>Shop Floor Paris</Name>
      <Name2>Shop Floor Paris</Name2>
      <City>Paris</City>
      <ShopId>000005</ShopId>
      <AncestorShopId>000005</AncestorShopId>
      <DescendantName>Shop Floor Paris</DescendantName>
      <FollowingSiblingCount>1</FollowingSiblingCount>
      <PrecedingSiblingCount>1</PrecedingSiblingCount>
    </Store>
    <Store>
      <ID>888888881</ID>
      <ID2>888888881</ID2>
      <Name>Shop Floor New York</Name>
      <Name2>Shop Floor New York</Name2>
      <City>New York</City>
      <ShopId>888888881</ShopId>
      <AncestorShopId>888888881</AncestorShopId>
      <DescendantName>Shop Floor New York</DescendantName>
      <FollowingSiblingCount>0</FollowingSiblingCount>
      <PrecedingSiblingCount>2</PrecedingSiblingCount>
    </Store>
  </Stores>
</root>
```


13

Filtering (Predicates)

Filtering Data using Predicates

Filters (Predicates) can be added at any step in path expressions in square brackets

```
root <- /StoreSalesData
{
  firstShopId := Shops/Shop[1]/@id
  lastShopId := Shops/Shop[last()]/@id
  parisShopStreet := Shops/Shop/ShopAddress[City = "Paris"]/Street
  newYorkShopReportDate := Shops/Shop[ShopAddress/CountryCodeIso = "US" and ShopAddress/City = "New York"]/ReportDate
}
```

This predicate will filter the first Shop, the other Shops will be ignored

Filters the last shop. `last()` is a contextual function that returns the index of the last child under the current context node

Filters all ShopAddresses with a specific City, then selects the Street

Filters all Shops by CountryCodeIso and City, then selects the resulting report date(s)

Output

```
<root>
  <firstShopId>8852662</firstShopId>
  <lastShopId>88888881</lastShopId>
  <parisShopStreet>Rue du Jour 1</parisShopStreet>
  <newYorkShopReportDate>2019-05-06T23:57:31.000-04:00</newYorkShopReportDate>
</root>
```

14

Sorting

Sorting Data using Order By Clauses

Every loop can have an
order by clause with
one or more sorting
criteria

ShopsOrderedByCity

```
{  
  Shop <- $shop in Shops/Shop order by $shop/ShopAddress/City  
  {  
    name := ShopAddress/Name  
    city := ShopAddress/City  
  }  
}
```

Output

```
<root>  
  <ShopsOrderedByCity>  
    <Shop>  
      <name>Shop Floor Berlin</name>  
      <city>Berlin</city>  
    </Shop>  
    <Shop>  
      <name>Shop Floor New York</name>  
      <city>New York</city>  
    </Shop>  
    <Shop>  
      <name>Shop Floor Paris</name>  
      <city>Paris</city>  
    </Shop>  
  </ShopsOrderedByCity>  
</root>
```

Sorting Data using Order By Clauses

Sort criteria can be arbitrarily complex

Each order criteria can have the flag ascending (default) or descending

ShopsOrderedByNumberOfItemsSold

```
{  
  Shop <- $shop in Shops/Shop order by sum($shop/ItemSales/Item/Quantity) descending  
  {  
    name := ShopAddress/Name  
    city := ShopAddress/City  
    totalItemSales := sum($shop/ItemSales/Item/Quantity)  
  }  
}
```

Output

```
<root>  
  <ShopsOrderedByNumberOfItemsSold>  
    <Shop>  
      <name>Shop Floor New York</name>  
      <city>New York</city>  
      <totalItemSales>1447</totalItemSales>  
    </Shop>  
    <Shop>  
      <name>Shop Floor Berlin</name>  
      <city>Berlin</city>  
      <totalItemSales>191</totalItemSales>  
    </Shop>  
    <Shop>  
      <name>Shop Floor Paris</name>  
      <city>Paris</city>  
      <totalItemSales>96</totalItemSales>  
    </Shop>  
  </ShopsOrderedByNumberOfItemsSold>  
</root>
```



15

Grouping

Grouping Data using Group By Clauses

Every loop can have a `group by` clause with one or more grouping criteria

Groups items by `@id` and loops over the resulting groups

```
root <- /StoreSalesData
{
  itemGroup <- $items in Shops/Shop/ItemSales/Item group by $id := $items/@id
  {
    itemId := $id
    itemName := $items[1]/Desc
    numberOfItemSaleRecords := count($items)
    numberOfItemsSold := sum($items/Quantity)
  }
}
```

Grouping criteria can be assigned to variables directly in the `group by` clause

`$items` is a sequence containing all items in the group (it has at least one element in each iteration)

Output

```
<root>
  <itemGroup>
    <itemId>8546665252</itemId>
    <itemName>Lumberjack Shirt</itemName>
    <numberOfItemSaleRecords>3</numberOfItemSaleRecords>
    <numberOfItemsSold>173</numberOfItemsSold>
  </itemGroup>
  ...
</root>
```



- Loops in XQuery are also called **FLWOR** Statements:
 - For
 - Let
 - Where
 - Order By (+ Group By)
 - Return
- The BIS Mapping Language offers the same capabilities for loops with an easier syntax.





2024 SEEBURGER AG. All rights reserved.

The information in this document is proprietary to SEEBURGER. Neither any part of this document, nor the whole of it may be reproduced, copied, or transmitted in any form or purpose without the express prior written permission of SEEBURGER AG. Please note that this document is subject to change and may be changed by SEEBURGER at any time without notice. SEEBURGER's Software product, the ones of its business partners may contain software components from third parties.

As far as reference to other brands is concerned, we refer to the following:

SAP®, SAP® R/3®, SAP NetWeaver®, SAP Cloud Platform & Cloud Platform Integrator®, SAP Archive Link®, SAP S/4HANA®, SAP® GLOBAL TRADE Service® (SAP GTS), SAP Fiori®, ABAP™ and SAP ARIBA® are registered trade marks of the SAP SE or the SAP Deutschland SE & Co. KG (Germany). Microsoft, Windows, Windows Phone, Excel, Outlook, PowerPoint, Silverlight, and Visual Studio are registered trademarks of Microsoft Corporation in the United States and other countries. Linux is a registered trade mark of Linus Torvalds in the United States and other countries. UNIX, X/Open, OSF/1, and Motif are registered trademarks of the Open Group. Adobe, the Adobe logo, Acrobat, Flash, PostScript, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and / or other countries. HTML, ML, XHTML, and W3C are trademarks, registered trademarks, or claimed as generic terms by the Massachusetts Institute of Technology (MIT), European Research Consortium for Informatics and Mathematics (ERCIM), or Keio University. Oracle and Java are registered trademarks of Oracle and its affiliates.

All other company and software names mentioned are registered trademarks or unregistered trademarks of their respective companies and are, as such, subject to the statutory provisions and legal regulations. 4invoice®, iMartOne®, SEEBURGER®, SEEBURGER Business-Integration Server®, SEEBURGER Logistic Solution Professional®, SEEBURGER Web Supplier Hub®, WinELKE®, SEEBURGER File Exchange®, SEEBURGER Link®, SMART E-Invoice® and other products or services of SEEBURGER which appear in this document as well as the according logos are marks or registered marks of the SEEBURGER AG in Germany and of other countries worldwide. All other products and services names are marks of the mentioned companies.

All contents of the present document are noncommittal and have a mere information intention. Products and services may be country-specific designed. All other mentioned company and software designations are trade marks or unregistered trade marks of the respective organizations and are liable to the corresponding legal regulations.

- The information in this document is proprietary to SEEBURGER. No part of this document may be reproduced, copied, or transmitted in any form or purpose without the express prior written permission of SEEBURGER AG.
- This document is a preliminary version and not subject to your license agreement or any other agreement with SEEBURGER. This document contains only intended strategies, developments, and functionalities of the SEEBURGER product and is not intended to be binding upon SEEBURGER to any particular course of business, product strategy, and/or development. Please note that this document is subject to change and may be changed by SEEBURGER at any time without notice.
- SEEBURGER assumes no responsibility for errors or omissions in this document. SEEBURGER does not warrant the accuracy or completeness of the information, text, graphics, links, or other items contained within this material. This document is provided without a warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability, fitness for a particular purpose, or non-infringement.
- SEEBURGER shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials. This limitation shall not apply in cases of intent or gross negligence.
- The statutory liability for personal injury and defective products is not affected. SEEBURGER has no control over the information that you may access through the use of hot links contained in these materials and does not endorse your use of third-party web pages nor provide any warranty whatsoever relating to third-party web pages.