Part III

Well-Formed XML

Outline of this part

Formalization of XML

- We will now try to approach XML in a slightly more formal way. The nuts and bolts of XML are pleasingly easy to grasp.
- This discussion will be based on *the* central XML technical specification:
  - **Extensible Markup Language (XML) 1.0 (Second Edition) W3C Recommendation 6 October 2000** (`http://www.w3.org/TR/REC-xml`)

Visit the W3C site

This lecture does *not* try to be a "guided tour" through the XML-related W3C technical documents (*boring!*).

Instead we will cover the basic principles and most interesting ideas. Visit the W3C site and use the original W3C documents to get a full grasp of their contents.

Elements

- The **element** is the main markup construct provided by XML.
  - Marked up document region (**element content**) enclosed in matching **start** end **closing (end) tags**:
    - **start tag**: <*t*> (*t* is the **tag name**),
    - matching **closing tag**: </*t*>

Well-formed XML (fragments)

```
<foo> okay </foo>
<This-is-a-well-formed-XML-tag.> okay
</This-is-a-well-formed-XML-tag.>
<foo>okay</foo>
```

Non-well-formed XML

```
<foo> oops </bar>
<foo> oops </Foo>
<foo> oops ... ⟨EOT⟩
```

Elements

- The **element** is the main markup construct provided by XML.
  - Marked up document region (**element content**) enclosed in matching **start** end **closing (end) tags**:
    - **start tag**: <*t*> (*t* is the **tag name**),
    - matching **closing tag**: </*t*>

Well-formed XML (fragments)

```
<foo> okay </foo>
<This-is-a-well-formed-XML-tag.> okay
</This-is-a-well-formed-XML-tag.>
<foo>okay</foo>
```

Non-well-formed XML

```
<foo> oops </bar>
<foo> oops </Foo>
<foo> oops ... ⟨EOT⟩
```


- Element content may contain document characters as well as **properly nested elements**, so-called **mixed content**):

Well-formed XML

```
<foo><bar>
       <baz> okay </baz>
     </bar>
     <ok> okay </ok> still okay
</foo>
```

Non-well-formed XML

```
<foo><bar> oops </foo></bar>
<foo><bar> oops </bar><bar> oops </foo></bar>
```

✎ **Check for proper nesting**

Which data structure would you use to straightforwardly implement the check for proper nesting in an XML parser?

- Element content may be **empty**:
  - The fragments <*t*></*t*> and <*t*/> are well-formed XML and considered equivalent.
- Element nesting establishes a **parent–child relationship between elements**:
  - In the XML fragment <*p*><*c*>. . . </*c*>. . .<*c*′>. . . </*c*′></*p*>,
    - element p is *the* **parent** of elements c, c',
    - elements c, c' are **children** of element p,
    - elements c, c' are **siblings**.
- There is exactly one element that encloses the **whole** XML content: the **root element**.

Non-well-formed XML

```
<one>
   one eins un
</one>
<two> two zwei deux </two>
```


Attributes

- Elements may further be classified using **attributes**:
  (It is common practice to denote an attribute named *a* by @*a* in written text (**at**tribute *a*).)

  <*t* *a*="..." *b*='...' ...> ...</*t*>

  - An attribute **value** is restricted to character data (attributes may *not* be nested),
  - attributes are *not* considered to be children of the containing element (instead they are **owned** by the containing element).

Well-formed XML (fragment)

```
<price currency="US$" multiplier='1'>
  23.45
</price>
<price>
   <currency>US$</currency>
   <multiplier>1</multiplier>
   23.45
</price>
```

Entities

- In XML, document **content** and **markup** are specificed using a *single set of characters*.
- The characters { <, >, &, ", ' } form pieces of XML markup and may instead be denoted by **predefined entities** if they actually represent content:

Character	Entity
<	<
>	>
&	&
"	"
'	'

Well-formed XML

```
<operators>Valid comparison operators
  are &lt;, =, &amp; &gt;.</operators>
```

- The XML entity facility is actually a versatile recursive **macro** expansion machinery (more on that later).

Well-Formedness

- The W3C XML recommendation is actually more formal and rigid in defining the syntactical structure of XML:

  "A textual object is **well-formed** *XML if,*

  1. *Taken as a whole, it* **matches the production labeled** *document.*
  2. *It meets all the* **well-formedness constraints** *given in this [the W3C XML Recommendation] specification. ..."*

Well-formedness #1: Context-free Properties

1. All **context-free** properties of well-formed XML documents are concisely captured by a **grammar** (using an EBNF-style notation).

   - **Grammar**: system of **production (rule)s** of the form

$$lhs ::= rhs$$

Excerpt of the XML grammar

<table>
<tr><td>[1]</td><td>document</td><td>::=</td><td>prolog element Misc*</td></tr>
<tr><td>[2]</td><td>Char</td><td>::=</td><td>⟨a Unicode character⟩</td></tr>
<tr><td>[3]</td><td>S</td><td>::=</td><td>('␣' | '\t' | '\n' | '\r')+</td></tr>
<tr><td>[4]</td><td>NameChar</td><td>::=</td><td>Letter | Digit | '.' | '-' | '_' | ':'</td></tr>
<tr><td>[5]</td><td>Name</td><td>::=</td><td>(Letter | '_' | ':') (NameChar)*</td></tr>
<tr><td>[10]</td><td>AttValue</td><td>::=</td><td>'"' ([^<&"] | Reference)* '"'</td></tr>
<tr><td></td><td></td><td>|</td><td>''' ([^<&'] | Reference)* '''</td></tr>
<tr><td>[14]</td><td>CharData</td><td>::=</td><td>[^<&]*</td></tr>
<tr><td>[22]</td><td>prolog</td><td>::=</td><td>XMLDecl? Misc*</td></tr>
<tr><td>[23]</td><td><i>XMLDecl</i></td><td>::=</td><td>'<?xml' <i>VersionInfo EncodingDecl? S?</i> '?>'</td></tr>
<tr><td>[24]</td><td>VersionInfo</td><td>::=</td><td>S 'version' Eq (''' VersionNum ''' | '"' VersionNum '"')</td></tr>
<tr><td>[25]</td><td>Eq</td><td>::=</td><td>S? '=' S?</td></tr>
<tr><td>[26]</td><td>VersionNum</td><td>::=</td><td>([a-zA-Z0-9_.:] | '-')+</td></tr>
<tr><td>[27]</td><td>Misc</td><td>::=</td><td>S</td></tr>
<tr><td>[39]</td><td>element</td><td>::=</td><td>EmptyElemTag</td></tr>
<tr><td></td><td></td><td>|</td><td>STag content ETag</td></tr>
<tr><td>[40]</td><td><i>STag</i></td><td>::=</td><td>'<' <i>Name</i> (<i>S Attribute</i>)* <i>S?</i> '>'</td></tr>
<tr><td>[41]</td><td>Attribute</td><td>::=</td><td>Name Eq AttValue</td></tr>
<tr><td>[42]</td><td><i>ETag</i></td><td>::=</td><td>'</' <i>Name S?</i> '>'</td></tr>
<tr><td>[43]</td><td>content</td><td>::=</td><td>(element | CharData | Reference)*</td></tr>
<tr><td>[44]</td><td><i>EmptyElemTag</i></td><td>::=</td><td>'<' <i>Name</i> (<i>S Attribute</i>)* <i>S?</i> '/>'</td></tr>
<tr><td>[67]</td><td>Reference</td><td>::=</td><td>EntityRef</td></tr>
<tr><td>[68]</td><td>EntityRef</td><td>::=</td><td>'&' Name ';'</td></tr>
<tr><td>[84]</td><td>Letter</td><td>::=</td><td>[a-zA-Z]</td></tr>
<tr><td>[88]</td><td>Digit</td><td>::=</td><td>[0-9]</td></tr>
</table>

N.B.

- The numbers in [·] refer to the correspondig productions in the W3C XML Recommendation.

Expression. . .	. . . denotes	
r^*	$\epsilon, r, rr, rrr, \ldots$	zero or more repetions of r
r^+	rr^*	one or more repetions of r
$r?$	$r \mid \epsilon$	optional r
`[abc]`	`a` | `b` | `c`	character class
`[^abc]`		inverted character class

Remarks

Rule. . .	. . . implements this characteristic of XML:
[1]	an XML document contains exactly one **root element**
[10]	attribute values are enclosed in `"` or `'`
[22]	XML documents may include an optional **declaration prolog**
[14]	characters `<` and `&` may *not* appear literally in element content
[43]	element content may contain character data and entity references as well as nested elements
[68]	entity references may contain arbitrary entity names (other than `lt`, `amp`, . . .)
⋮	⋮

- As usual, the XML grammar may systematically be transformed into a program, an **XML parser**, to be used to check the syntax of XML input.

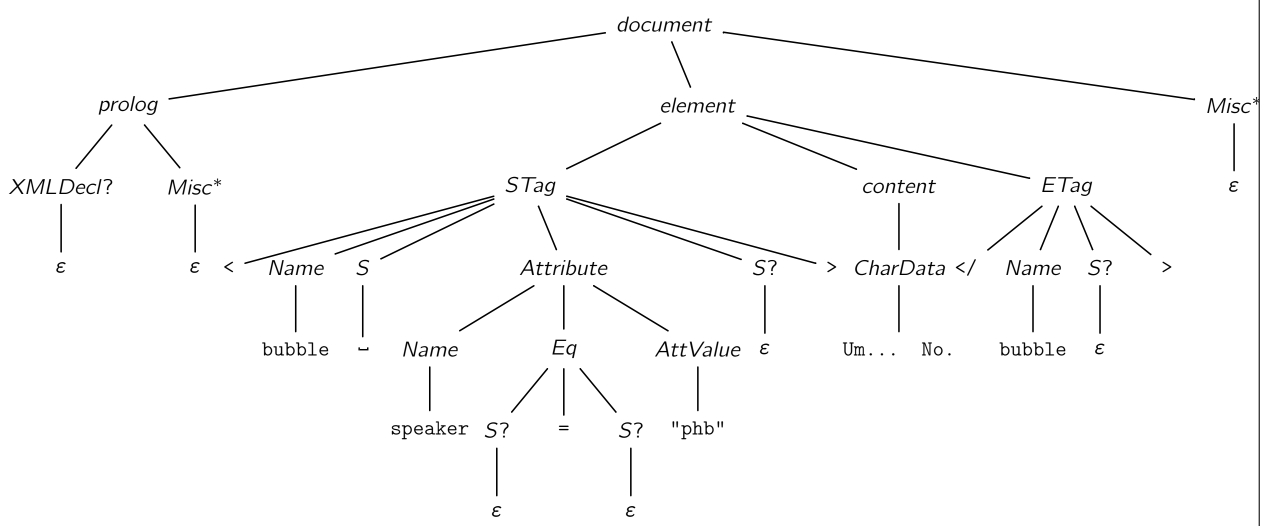
Parsing XML

1. Starting with the symbol *document*, the parser uses the *lhs* ::= *rhs* rules to expand symbols, constructing a **parse tree**.
2. The leaves of the parse tree are characters which have no further expansion.
3. The XML input is **parsed** successfully if it perfectly matches the parse tree's **front** (concatenate the parse tree leaves from left to right[1]).

[1] **N.B.**: $\texttt{x}\varepsilon\texttt{y} = \texttt{xy}$.

Example 1

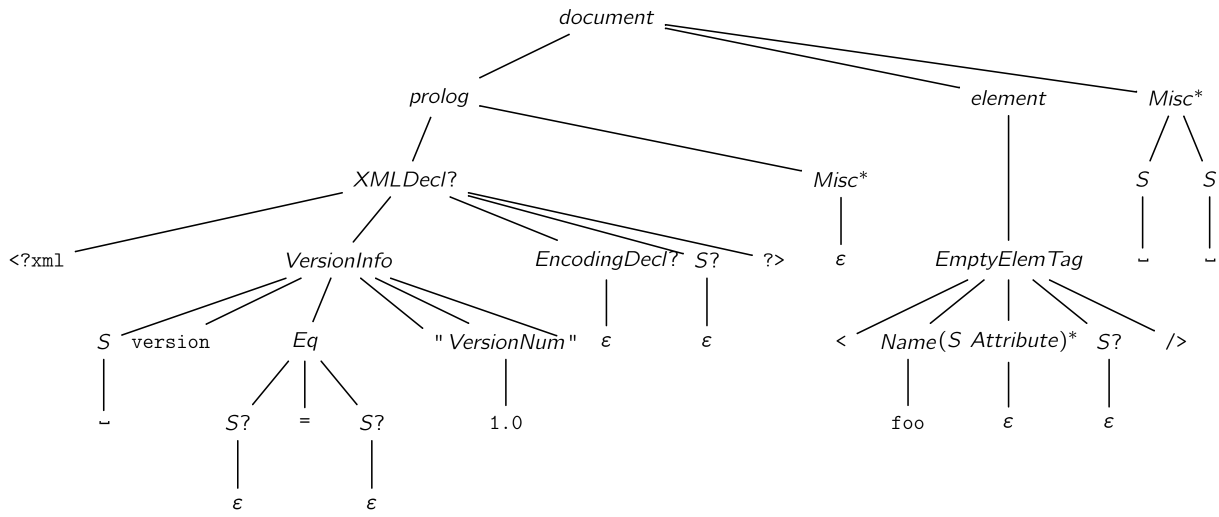
Parse tree for XML input
<bubble speaker="phb">Um... No.</bubble> :



Example 2

Parse tree for the "minimal" XML document

```
<?xml version="1.0"?><foo/>␣␣
```



Well-formedness #2: Context-dependent Properties

- The XML grammar cannot enforce *all* XML **well-formedness constraints** (**WFCs**).
- Some XML WFCs depend on
  1. what the XML parser has **seen before** in its input, or
  2. on a **global state**, *e.g.*, the definitions of user-declared entities.
- These WFCs cannot be checked by simply comparing the parse tree front against the XML input (**context-dependent WFCs**).

Sample WFCs

WFC	Comment
(2) Element Type Match	The *Name* in an element's end tag must match the element name in the start tag.
(3) Unique Att Spec	No attribute name may appear more than once in the same start tag or empty element tag.
(5) No < in Attribute Values	The replacement text of any entity referred to directly or indirectly in an attribute value (other than `<`) must not contain a <.
(9) No Recursion	A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

All 10 XML WFCs are given in `http://www.w3.org/TR/REC-xml`.

✎ How to implement the XML WFC checks?

Devise methods—besides parse tree construction—that an XML parser could use to check the XML WFCs listed above.

Specify *when* during the parsing process you would apply each method.

The XML Text Declaration `<?xml...?>`

- Remember that a well-formed XML document may start off with an optional header, the **text declaration** (grammar rule [23]).
  - **N.B.** Rule [23] says, *if* the declaration is actually there, no character (whitespace, *etc.*) may preceed the leading `<?xml`.

✎ The leading `<?xml`

Can you imagine why the XML standard is so rigid with respect to the placement of the `<?xml` leader of the text declaration?

- An XML document whose text declaration carries a *VersionInfo* of `version="1.0"` is required to conform to W3C's XML Recommendation posted on October 6, 2000 (see `http://www.w3.org/TR/REC-xml`).

XML Documents and Character Encoding

- For a computer, a character like `X` is nothing but an 8 (16/32) bit **number** whose value is *interpreted* as the character `X` when needed (*e.g.*, to drive a display).
- Trouble is, a large number of such *number* → *character* mapping tables, the so-called **encodings**, are in parallel use today.
- Due to the huge amount of characters needed by the global computing community today (Latin, Hebrew, Arabic, Greek, Japanese, Chinese . . . languages), **conflicting intersections** between encodings are common.

  Example:

  0xa4 0xcb 0xe4 0xd3 $\xrightarrow{\texttt{iso-8859-7}}$?, Λ δ Σ
  0xa4 0xcb 0xe4 0xd3 $\xrightarrow{\texttt{iso-8859-15}}$ € Ë ä Ó

XML Text Declarations | XML and Character Encoding

XML Documents and Character Encoding

- For a computer, a character like `X` is nothing but an 8 (16/32) bit **number** whose value is *interpreted* as the character `X` when needed (*e.g.*, to drive a display).
- Trouble is, a large number of such *number* → *character* mapping tables, the so-called **encodings**, are in parallel use today.
- Due to the huge amount of characters needed by the global computing community today (Latin, Hebrew, Arabic, Greek, Japanese, Chinese . . . languages), **conflicting intersections** between encodings are common.

  Example:

  0xa4 0xcb 0xe4 0xd3 $\xrightarrow{\texttt{iso-8859-7}}$?, Λ δ Σ
  0xa4 0xcb 0xe4 0xd3 $\xrightarrow{\texttt{iso-8859-15}}$ € Ë ä Ó

Torsten Grust (WSI) | Database-Supported XML Processors | Winter 2012/13 | 44

XML Documents and Character Encoding

- For a computer, a character like `X` is nothing but an 8 (16/32) bit **number** whose value is *interpreted* as the character `X` when needed (*e.g.*, to drive a display).
- Trouble is, a large number of such *number* → *character* mapping tables, the so-called **encodings**, are in parallel use today.
- Due to the huge amount of characters needed by the global computing community today (Latin, Hebrew, Arabic, Greek, Japanese, Chinese . . . languages), **conflicting intersections** between encodings are common.
  Example:

$$\text{0xa4 0xcb 0xe4 0xd3} \xrightarrow{\texttt{iso-8859-7}} \boxed{?}, \Lambda\,\delta\,\Sigma$$

$$\text{0xa4 0xcb 0xe4 0xd3} \xrightarrow{\texttt{iso-8859-15}} \text{€ Ë ä Ó}$$

Unicode

- The **Unicode** (`http://www.unicode.org/`) Initiative aims to define a new encoding that tries to embrace all character needs.
- The Unicode encoding contains characters of “all” languages of the world, plus scientific, mathematical, technical, box drawing, . . . symbols (see `http://www.unicode.org/charts/`).
- Range of the Unicode encoding: `0x0000`–`0x10FFFF` (16 × 65536 characters).
  - Codes that fit into the first 16 bits (denoted `U+0000`–`U+FFFF`) have been assigned to encode the most widely used languages and their characters (**Basic Multilingual Plane, BMP**).
  - Codes `U+0000`–`U+007F` have been assigned to match the 7-bit ASCII encoding which is pervasive today.

UTF-32

- Current CPUs operate most efficiently on **32-bit words** (**16-bit words**, **8-bit bytes**).
- Unicode thus developed **Unicode Transformation Formats (UTF)** which define how a Unicode character code between `U+0000`–`U+10FFFF` is to be mapped into a 32-bit word (16-bit words, 8-bit bytes).

UTF-32 (map a Unicode character into a 32-bit word)

1. Map any Unicode character in the range `U+0000`–`U+10FFFF` to the corresponding 32-bit value `0x00000000`–`0x0010FFFF`.
2. **N.B.** For each Unicode character encoded in UTF-32 we waste at least 11 zero bits.

UTF-16

. . . map a Unicode character into one or two 16-bit words

1. Apply the following mapping scheme:

Unicode range	Word sequence
`U+000000`–`U+00FFFF`	□□□□□□□□□□□□□□□□
`U+010000`–`U+10FFFF`	110110□□□□□□□□□□ 110111□□□□□□□□□□

2. For the range `U+000000`–`U+00FFFF`, simply fill the □ positions with the 16 bit of the character code.
(Code ranges `U+D800`–`U+DBFF` and `U+DC00`–`U+DFFF` are unassigned!)
3. For the `U+010000`–`U+10FFFF` range, subtract `0x010000` from the character code and fill the □ positions using the resulting 20-bit value.

Example

Unicode character `U+012345` (`0x012345` − `0x010000` = `0x02345`):
UTF-16: `1101100000001000` `1101111101000101`

UTF-8

N.B. UTF-16 is designed to facilitate efficient and robust decoding:

- If we see a leading 11011 bit pattern in a 16-bit word, we know it is the first **or** second word in a UTF-16 multi-word sequence.
- The sixth bit of the word then tells us if we actually look at the first or second word.

UTF-8 (map a Unicode character into a sequence of 8-bit bytes)

- UTF-8 is of special importance because
  (a) a stream of 8 bit bytes (*octets*) is what flows over an IP network connection,
  (b) text-processing software today is built to deal with 8 bit character encodings (`iso-8859-x`, ASCII, *etc.*).

UTF-8 encoding

1. Apply the following mapping scheme:

Unicode range	Byte sequence
`U+000000–U+00007F`	`0□□□□□□□`
`U+000080–U+0007FF`	`110□□□□□ 10□□□□□□`
`U+000800–U+00FFFF`	`1110□□□□ 10□□□□□□ 10□□□□□□`
`U+010000–U+10FFFF`	`11110□□□ 10□□□□□□ 10□□□□□□ 10□□□□□□`

2. The spare bits (□) are filled with the bits of the character code to be represented (rightmost □ is least significant bit, pad to the left with 0-bits).
   Examples:
   - Unicode character `U+00A9` (© sign):
     UTF-8: `11000010 10101001` (`0xC2 0xA9`)
   - Unicode character `U+2260` (math relation symbol ≠):
     UTF-8: `11100010 10001001 10100000` (`0xE2 0x89 0xA0`)

Advantages of UTF-8 encoding

N.B. UTF-8 enjoys a number of highly desirable properties:

- For a UTF-8 multi-byte sequence, the **length of the sequence** is equal to the number of leading 1-bits (in the first byte), *e.g.*:

  $\underline{111}00010\ 10001001\ 10100000$

  (Only single-byte UTF-8 encodings have a leading 0-bit.)
- **Character boundaries** are simple to detect (even when placed at some arbitrary position in a UTF-8 byte stream).
- UTF-8 encoding does not affect (binary) sort order.
- Text processing software which was originally developed to work with the pervasive 7-bit ASCII encoding remains functional.
  This is especially true for the C programming language and its string (`char[]`) representation.

C and UTF-8

Can you explain the last points made?

Advantages of UTF-8 encoding

N.B. UTF-8 enjoys a number of highly desirable properties:

- For a UTF-8 multi-byte sequence, the **length of the sequence** is equal to the number of leading 1-bits (in the first byte), *e.g.*:

  $\underline{111}00010\ 10001001\ 10100000$

  (Only single-byte UTF-8 encodings have a leading 0-bit.)
- **Character boundaries** are simple to detect (even when placed at some arbitrary position in a UTF-8 byte stream).
- UTF-8 encoding does not affect (binary) sort order.
- Text processing software which was originally developed to work with the pervasive 7-bit ASCII encoding remains functional.
  This is especially true for the C programming language and its string (`char[]`) representation.

C and UTF-8

Can you explain the last points made?

Advantages of UTF-8 encoding

N.B. UTF-8 enjoys a number of highly desirable properties:

- For a UTF-8 multi-byte sequence, the **length of the sequence** is equal to the number of leading 1-bits (in the first byte), *e.g.*:

  `11100010 10001001 10100000`

  (Only single-byte UTF-8 encodings have a leading 0-bit.)
- **Character boundaries** are simple to detect (even when placed at some arbitrary position in a UTF-8 byte stream).
- UTF-8 encoding does not affect (binary) sort order.
- Text processing software which was originally developed to work with the pervasive 7-bit ASCII encoding remains functional.
  This is especially true for the C programming language and its string (`char[]`) representation.

✎ C and UTF-8

Can you explain the last points made?

XML Text Declarations | Unicode

Advantages of UTF-8 encoding

N.B. UTF-8 enjoys a number of highly desirable properties:

- For a UTF-8 multi-byte sequence, the **length of the sequence** is equal to the number of leading 1-bits (in the first byte), *e.g.*:

  `11100010 10001001 10100000`

  (Only single-byte UTF-8 encodings have a leading 0-bit.)
- **Character boundaries** are simple to detect (even when placed at some arbitrary position in a UTF-8 byte stream).
- UTF-8 encoding does not affect (binary) sort order.
- Text processing software which was originally developed to work with the pervasive 7-bit ASCII encoding remains functional.
  This is especially true for the C programming language and its string (`char[]`) representation.

✎ C and UTF-8

Can you explain the last points made?

Torsten Grust (WSI) | Database-Supported XML Processors | Winter 2012/13 | 50

Advantages of UTF-8 encoding

N.B. UTF-8 enjoys a number of highly desirable properties:

- For a UTF-8 multi-byte sequence, the **length of the sequence** is equal to the number of leading 1-bits (in the first byte), *e.g.*:

  <u>111</u>00010 10001001 10100000

  (Only single-byte UTF-8 encodings have a leading 0-bit.)
- **Character boundaries** are simple to detect (even when placed at some arbitrary position in a UTF-8 byte stream).
- UTF-8 encoding does not affect (binary) sort order.
- Text processing software which was originally developed to work with the pervasive 7-bit ASCII encoding remains functional.
  This is especially true for the C programming language and its string (`char[]`) representation.

✎ C and UTF-8

Can you explain the last points made?

XML and Unicode

- A conforming XML parser is required to correctly process UTF-8 and UTF-16 encoded documents (The W3C XML Recommendation predates the UTF-32 definition).
- Documents that use a different encoding *must* announce so using the XML text declaration, *e.g.*

  `<?xml version="1.0" encoding="iso-8859-15"?>`
  or `<?xml version="1.0" encoding="utf-32"?>`
- Otherwise, an XML parser is encouraged to **guess** the encoding while reading the very first bytes of the input XML document:

Head of doc (bytes)	**Encoding guess**
`0x00 0x3C 0x00 0x3F`	UTF-16 (*big-endian*)
`0x3C 0x00 0x3F 0x00`	UTF-16 (*little-endian*)
`0x3C 0x3F 0x78 0x6D`	UTF-8 (or ASCII, `iso-8859-*`: *erroneous*)

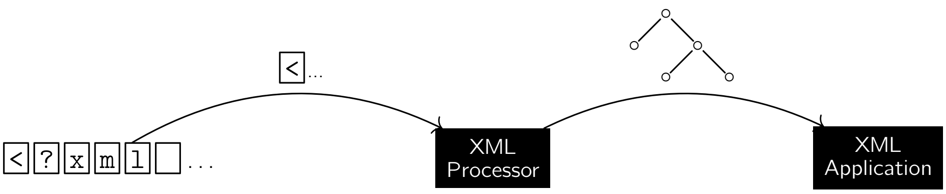
(Notice: `<` = `U+003C`, `?` = `U+003F`, `x` = `U+0078`, `m` = `U+006D`)

The XML Processing Model

- On the **physical** side, XML defines nothing but a **flat text format**, *i.e.*, defines a set of (UTF-8/16) character sequences being well-formed XML.
- Applications that want to analyse and transform XML data in any meaningful manner will find processing flat character sequences hard and inefficient.
- The **nesting** of XML elements and attributes, however, defines a **logical tree-like structure**.

XML Processors

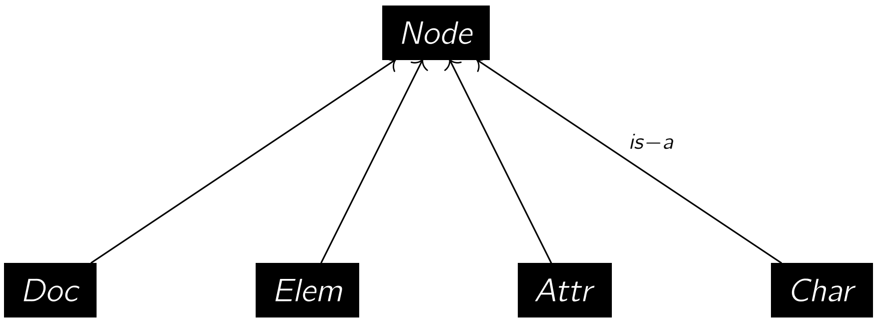
- Virtually all **XML applications operate on the logical tree view** which is provided to them through an **XML Processor** (*i.e.*, the XML parser):



- XML processors are widely available (*e.g.*, Apache's Xerces see `http://xml.apache.org/`).
- How is the XML processor supposed to **communicate the XML tree structure** to the application . . . ?

The XML Information Set

- Once an XML processor has checked its XML input document to be well-formed, it offers its application a set of document **properties** (functions).
- The application calls property functions and thus explores the input XML tree as needed.
- An XML document tree is built of different types of **node objects**:



- The set of properties of all document nodes is the document's **Information Set** (see `http://www.w3.org/TR/xml-infoset/`).

Node properties

Node Object Type	Property		Comment
Doc	*children*	:: *Doc* → *Elem*	root element
	base-uri	:: *Doc* → *String*	
	version	:: *Doc* → *String*	`<?xml version="1.0"?>`
Elem	*localname*	:: *Elem* → *String*	
	children	:: *Elem* → (*Node*)	$*^1$
	attributes	:: *Elem* → (*Attr*)	
	parent	:: *Elem* → *Node*	$*^2$
Attr	*localname*	:: *Attr* → *String*	
	value	:: *Attr* → *String*	
	owner	:: *Attr* → *Elem*	
Char	*code*	:: *Char* → *Unicode*	a single character
	parent	:: *Char* → *Elem*	

- Read symbol :: as "*has type*".
- For any node type τ, (τ) denotes an ordered **sequence** of type τ.

✎ Make sense of the types of the *Elem* properties *children* ($*^1$) and *parent* ($*^2$)!

Information set of a sample document

Document δ_0 (weather forecast)

```xml
<?xml version="1.0"?>
<forecast date="Thu, Oct 30">
  <condition>foggy</condition>
  <temperature unit="Celsius">-1</temperature>
</forecast>
```

$$
\begin{aligned}
\mathit{children}\ (\delta_0) &= \varepsilon_1 \\
\mathit{base\text{-}uri}\ (\delta_0) &= \texttt{"file:/..."} \\[6pt]
\mathit{version}\ (\delta_0) &= \texttt{"1.0"} \\
\mathit{localname}\ (\varepsilon_1) &= \texttt{"forecast"} \\
\mathit{children}\ (\varepsilon_1) &= (\varepsilon_3, \varepsilon_9) \\
\mathit{attributes}\ (\varepsilon_1) &= (\alpha_2) \\
\mathit{parent}\ (\varepsilon_1) &= \delta_0 \\
\mathit{localname}\ (\alpha_2) &= \texttt{"date"} \\
\mathit{value}\ (\alpha_2) &= \texttt{"Thu, Oct 30"} \\[6pt]
\mathit{localname}\ (\varepsilon_3) &= \texttt{"condition"} \\
\mathit{children}\ (\varepsilon_3) &= (\gamma_4, \gamma_5, \gamma_6, \gamma_7, \gamma_8) \\
\mathit{attributes}\ (\varepsilon_3) &= () \\
\mathit{parent}\ (\varepsilon_3) &= \varepsilon_1
\end{aligned}
$$

$$
\begin{aligned}
\mathit{code}\ (\gamma_4) &= \texttt{U+0066 'f'} \\
\mathit{parent}\ (\gamma_4) &= \varepsilon_3 \\
&\vdots \\
\mathit{code}\ (\gamma_8) &= \texttt{U+0079 'y'} \\
\mathit{parent}\ (\gamma_8) &= \varepsilon_3 \\
\mathit{localname}\ (\varepsilon_9) &= \texttt{"temperature"} \\
\mathit{children}\ (\varepsilon_9) &= (\gamma_{11}, \gamma_{12}) \\
\mathit{attributes}\ (\varepsilon_9) &= (\alpha_{10}) \\
\mathit{parent}\ (\varepsilon_9) &= \varepsilon_1 \\
&\vdots
\end{aligned}
$$

N.B. Node objects of type *Doc*, *Elem*, *Attr*, *Char* are denoted by δ_i, ε_i, α_i, γ_i, respectively (subscript i makes object identifiers unique).

Working with the Information Set

- The W3C has introduced the XML Information Set to aid the specification of further XML standards.
- We can nevertheless use it to write simple "programs" that explore the XML tree structure. The resulting code looks fairly similar to code we would program using the DOM (Document Object Model, see next chapter).

Example: Compute the list of **sibling** *Elem* nodes of given *Elem* ε (including ε):

$\underline{\mathit{siblings}\ (\varepsilon) :: \mathit{Elem} \rightarrow (\mathit{Elem})}$

Node ν;

$\nu \leftarrow \mathit{parent}\ (\varepsilon)$;
if $\nu = \delta_\square$ then
  // ν *is the Doc node, i.e.,* ε *is the root element*
  return (ε);
else
  return *children* (ν);

Another Example

Return the text **content** of a given *Doc* δ (the sequence of all Unicode characters δ contains):

$\underline{\mathit{content}\ (\delta) :: \mathit{Doc} \rightarrow (\mathit{Unicode})}$

return *collect* ((*children* (δ)));

$\underline{\mathit{collect}\ (\nu s) :: (\mathit{Node}) \rightarrow (\mathit{Unicode})}$

```
Node ν;
(Unicode) γs;

γs ← ();
foreach ν ∈ νs do
    if ν = γ□ then
        // we have found a Char node ...
        γs ← γs + (code (ν));
    else
        // otherwise ν must be an Elem node
        γs ← γs + collect (children (ν));
return γs;
```

- Example run: *content* (δ_0) = ('f','o','g','g','y','-','1').

"Querying" using the Information Set

- Having the XML Information Set in hand, we can analyse a given XML document in arbitrary ways, *e.g.*
  1. In a given document (comic strip), find **all** *Elem* nodes with local name `bubble` owning an *Attr* node with local name `speaker` and value `"Dilbert"`.
  2. List **all** `scene` *Elem* nodes **containing** a `bubble` spoken by `"Dogbert"` (*Attr* `speaker`).
  3. Starting in `panel` number 2 (`no` *Attr*), find **all** `bubble`s **following** those spoken by `"Alice"` (*Attr* `speaker`).
- Queries like these are quite common in XML applications. An XML standard exists (**XPath**) which allows to specify such **document path traversals** in a declarative manner:
  1. `//bubble[./@speaker = "Dilbert"]`
  2. `//bubble[@speaker = "Dogbert"]/../..`
  3. `//panel[@no = "2"]//bubble[@speaker = "Alice"]/following::bubble`

More XML node types . . .

- The XML standard defines a number of additional node types that may occur in well-formed documents (and thus in their XML Information Set).
- **CDATA** nodes (embed *unparsed* non-binary character data)

CDATA

```
<source>
  <![CDATA[ May use <, >, and & and
     anything else freely here ]]>
</source>
```

- **Comment** nodes (place comments in XML documents)

Comment

```
<proof>
  <!-- Beware! This has not been properly
       checked yet... -->
  ...
</proof>
```


XML Processing Model | More XML Node Types

More XML node types . . .

- The XML standard defines a number of additional node types that may occur in well-formed documents (and thus in their XML Information Set).
- **CDATA** nodes (embed *unparsed* non-binary character data)

CDATA

```
<source>
  <![CDATA[ May use <, >, and & and
     anything else freely here ]]>
</source>
```

- **Comment** nodes (place comments in XML documents)

Comment

```
<proof>
  <!-- Beware! This has not been properly
       checked yet... -->
  ...
</proof>
```

More XML node types . . .

- The XML standard defines a number of additional node types that may occur in well-formed documents (and thus in their XML Information Set).
- **CDATA** nodes (embed *unparsed* non-binary character data)

CDATA

```
<source>
  <![CDATA[ May use <, >, and & and
     anything else freely here ]]>
</source>
```

- **Comment** nodes (place comments in XML documents)

Comment

```
<proof>
  <!-- Beware! This has not been properly
       checked yet... -->
  ...
</proof>
```


. . . more XML node types

- **PI** nodes (embed *processing instructions* in XML documents)

PI

```
<em>
  <b>Result:</b>
  <?php sql ("SELECT * FROM ...") ...?>
</em>
```

- For a complete list of node types see the W3C XML Recommendation (`http://www.w3.org/TR/REC-xml`).

. . . more XML node types

- **PI** nodes (embed *processing instructions* in XML documents)

PI

```
<em>
  <b>Result:</b>
  <?php sql ("SELECT * FROM ...") ...?>
</em>
```

- For a complete list of node types see the W3C XML Recommendation (`http://www.w3.org/TR/REC-xml`).