5.2 Computing with XSLT

- **XSLT** is a declarative rule-based language
  - for a special purpose: XML transformation
  - Can we express procedural computations with XSLT?
  - What is the exact computational power of XSLT?
- We've seen some programming-like features:
  - iteration over selected source nodes (xsl:for-each)
  - conditional evaluation (xsl:if and xsl:choose)

---

A Real-Life Example

- We used LaTeX to format an XML article. For this, source table structures
  
  \[
  \begin{tabular}{c|c|c|c}
  \hline
  A & B & C \\
  \hline
  1 & 2 & 3 \\
  \hline
  \end{tabular}
  \]
  
  had to be mapped to corresponding LaTeX environments:
  
  \begin{verbatim}
  \begin{tabular}{|c|c|c|}
  \hline
  \end{tabular}
  \end{verbatim}

- How to do this?

---

Solution (1/2)

- Pass the count of columns to a named template which generates an appropriate number of 1's.

  \begin{verbatim}
  <xsl:template match="tgroup">
  \begin{tabular}{{|c|c|c|}}
  \hline
  \end{tabular}
  \end{verbatim}

---

Solution 2/2: Recursive gen-cols

\begin{verbatim}
<xsl:template name="gen-cols">
  <xsl:param name="cols" />
</xsl:template>
\end{verbatim}

---

Computational power of XSLT

- XSLT seems quite powerful, but how powerful is it?
  - Implementations may provide extension mechanisms, e.g., to call arbitrary Java methods
  - Are there limits to XSLT processing that we can do without extensions?
- We can show that any algorithmic computation can be simulated with XSLT
  - shown indirectly, through simulating Turing machines by XSLT
Turing machine

- Alan Turing 1936/37
- Formal model of algorithms
- Primitive but powerful to simulate any computation expressible in any algorithmic model (Church/Turing thesis)
- Turing machine
  - A finite set of states
  - Unlimited tape of cells for symbols, examined by a tape head

Control of a Turing machine

- Control defined by a transition function \( \sigma \):
  \( \sigma(q, a) = (q', b, d) \), where \( d \in \{ \text{left, right} \} \)
  - Meaning: with current state \( q \) and tape symbol \( a \),
    - Move to new state \( q' \)
    - Write new symbol \( b \) at the place of \( a' \)
    - Move tape head one step in direction \( d \)
- Such control can be simulated in XSLT with a recursive named-template: Call it transition

Overall structure of the simulation

```xml
<xsl:template name="transition">
  <!-- parameters and trace output omitted -->
  <xsl:choose>
    <xsl:when test="/state = "YES""> 
      <ACCEPT />
    </xsl:when>
    <!-- an xsl:when for simulating each defined single transition comes here ... -->
    <xsl:otherwise> 
      <REJECT />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Illustration of a Turing Machine

The "transition" template

- Parameters:
  - state: the current state
  - left: contents of the tape up to the tape head
  - right: contents of the tape starting at the cell pointed by the tape head
- Transition simulates a single transition step; calls itself with updated parameters

Updating the representation of the tape

- For each right-move \( \sigma(q, a) = (q', b, \text{right}) \), concatenate \( b \) at the end of \( \text{left} \) and drop the first character of \( \text{right} \)
- Left-moves \( \sigma(q_0, a) = (q_0, b, \text{left}) \) in a similar manner:
  - Drop the last character of \( \text{left} \), and concatenate it in front of \( \text{right} \) whose first character has been replaced by \( b \)
- Example: a TM for palindromes over alphabet \( \{ a, b \} \) (if ' used for denoting blank tape slots)
Simulating a single transition (1/2)

```xml
<!-- simga(mark,a) = (move_a,#,right) -->
<xsl:when test="state = 'mark' and substring(right, 1, 1) = '#'">
  <!-- First update the parameters -->
  <xsl:variable name="newstate" select="move_a"/>
  <xsl:variable name="newleft" select="concat(left, '#')"/>
  <xsl:variable name="newright" select="substring(right, 2)"/>
</xsl:when>
```

Simulating a single transition (2/2)

```xml
<!-- Then call 'transition' with new parameter -->
<xsl:call-template name="transition">
  <xsl:with-param name="state" select="newstate"/>
  <xsl:with-param name="left" select="newleft"/>
  <xsl:with-param name="right" select="newright"/>
</xsl:call-template>
</xsl:when>
```

Sample trace of the simulation

```
$ saxon dummy.xm taperlindr.xsl input= tape=a

<table>
<thead>
<tr>
<th>state</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>move_a</td>
</tr>
<tr>
<td>#</td>
<td>move_a</td>
</tr>
<tr>
<td>#</td>
<td>text_a</td>
</tr>
<tr>
<td>#</td>
<td>return</td>
</tr>
<tr>
<td>#</td>
<td>move_a</td>
</tr>
<tr>
<td>#</td>
<td>sort_b</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>
```

What does this mean?

- XSLT has full algorithmic power
  - (It is "Turing-complete")
  - Is this intentional?
    - Inconvenient as a general-purpose programming language
    - Impossible to recognize non-terminating transformations automatically
      - (=> the "haling problem" has no algorithmic solution)
    - Malicious parser could cause "denial-of-service" through non-terminating style sheets