5.2 Computing with XSLT

- XSLT is principally a declarative rule-based language
  - 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
  <tgroup cols="3">
  ...
  </tgroup>
  had to be mapped to corresponding LaTeX environments:
  \begin{tabular}{lll} %3 left-justified cols
  ...
  \end{tabular}
- How to do this?

Solution (1/2)

- Pass the column count to a named template which generates an appropriate number of 'l's:
  <xsl:template match="tgroup">
  \begin{tabular}{<xsl:call-template name="gen-cols">
  <xsl:with-param name="count" select=" @cols" />
  <xsl:with-param name="symb" select="'l'" />
  </xsl:call-template>}
  <xsl:apply-templates />
  \end{tabular}
</xsl:template>

Solution 2/2: Recursive gen-cols

- 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

Computational power of XSLT

Further programming-like features:
- limited variables (names bound to values):
  <xsl:for-each select="/name">
  <xsl:variable name="LAndF" select="concat(last, ', ', first)" />
  </xsl:for-each>
- explicitly callable named templates with parameters:
  <xsl:call-template name="process-name">
  <xsl:with-param name="base" select="'LAndF' />
  </xsl:call-template>
</xsl:for-each>
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 \( \sigma \):
  \( \sigma(q_1, a) = (q_2, b, d) \), where \( d \in \text{[left, right]} \)
  - Meaning: with current state \( q_1 \) and tape symbol \( a \)
    - move to new state \( q_2 \)
    - 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>
    <xsl:otherwise>
      <REJECT />
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Updating the representation of the tape

- For each right-move \( \sigma(q_1, a) = (q_2, b, \text{right}) \),
  concatenate \( b \) at the end of \( \$\text{left} \) and drop the first character of \( \$\text{right} \)
- Left-moves \( \sigma(q_1, a) = (q_2, b, \text{left}) \) 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\}; ('#' used for denoting blank tape slots)
Simulating a single transition (1/2)

```xml
<when test="$state='mark' and substring($right, 1, 1)='a'>
  <!-- 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)"/>
</when>
```

Simulating a single transition (2/2)

```xml
<call-template name="transition">
  <with-param name="state" select="$newstate"/>
  <with-param name="left" select="$newleft"/>
  <with-param name="right" select="$newright"/>
</call-template>
```

Sample trace of the simulation

```
$ saxon dummy.xml tm-palindr.xsl input-tape=aba
<state shown as state_left=tape-left state_right=tape-right
<state state='#[mark]aba#'/>
<state state='##[move_a]ba#'/>
<state state='##b[move_a]a#'/>
<state state='##ba[move_a]#'/>
<state state='##b[test_a]a#'/>
<state state='##[return]b##'/>
<state state='#[return]#b##'/>
<state state='##[mark]b##'/>
<state state='###[move_b]##'/>
<.state state='##[test_b]###'/>
<state state='#[YES]####'/>
<accept/>
```

What does this mean?

- XSLT has full algorithmic programming power
  - (It is "Turing-complete")
  - Is this intentional?
    - Inconvenient as a general-purpose programming language!
    - Impossible to recognise non-terminating transformations automatically
      - The "halting problem" has no algorithmic solution
      - Malicious hacker could cause "denial-of-service" through non-terminating style sheets