SAX—Simple API for XML

- **SAX**\(^7\) (*Simple API for XML*) is, unlike DOM, *not* a W3C standard, but has been developed jointly by members of the XML-DEV mailing list (*ca.* 1998).

- SAX processors use **constant space**, regardless of the XML input document size.
  - Communication between the SAX processor and the backend XML application does *not* involve an intermediate tree data structure.
  - Instead, the **SAX parser sends events** to the application whenever a certain piece of XML text has been recognized (*i.e.*, parsed).
  - The **backend acts on/ignores events** by populating a **callback function table**.

\(^7\)http://www.saxproject.org/
A SAX processor reads its input document **sequentially** and **once only**.

No memory of what the parser has seen so far is retained while parsing. As soon as a **significant bit of XML text** has been recognized, an **event** is sent.

The application is able to act on events **in parallel** with the parsing...
SAX Events

To meet the constant memory space requirement, SAX reports fine-grained parsing events for a document:

<table>
<thead>
<tr>
<th>Event</th>
<th>...reported when seen</th>
<th>Parameters sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>startDocument</td>
<td>&lt;?xml...?&gt;(^8)</td>
<td></td>
</tr>
<tr>
<td>endDocument</td>
<td>⟨EOF⟩</td>
<td></td>
</tr>
<tr>
<td>startElement</td>
<td>&lt;t a₁=v₁ ... aₙ=vₙ⟩</td>
<td>t, (a₁, v₁), ..., (aₙ, vₙ)</td>
</tr>
<tr>
<td>endElement</td>
<td>⟨/t⟩</td>
<td>t</td>
</tr>
<tr>
<td>characters</td>
<td>text content</td>
<td>Unicode buffer ptr, length</td>
</tr>
<tr>
<td>comment</td>
<td>⟨!—c—⟩</td>
<td>c</td>
</tr>
<tr>
<td>processingInstruction</td>
<td>⟨?t pi?⟩</td>
<td>t, pi</td>
</tr>
</tbody>
</table>

\(^8\text{N.B.}: \) Event startDocument is sent even if the optional XML text declaration should be missing.
dilbert.xml

```xml
<?xml encoding="utf-8"?>
<bubbles>
<!-- Dilbert looks stunned -->
<bubble speaker="phb" to="dilbert">
  Tell the truth, but do it in your usual engineering way so that no one understands you.
</bubble>
</bubbles>
```

<table>
<thead>
<tr>
<th>Event</th>
<th>Parameters sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>*1</td>
<td><code>startDocument</code></td>
</tr>
<tr>
<td>*2</td>
<td><code>startElement</code></td>
</tr>
<tr>
<td>*3</td>
<td><code>comment</code></td>
</tr>
<tr>
<td>*4</td>
<td><code>startElement</code></td>
</tr>
<tr>
<td>*5</td>
<td><code>characters</code></td>
</tr>
<tr>
<td>*6</td>
<td><code>endElement</code></td>
</tr>
<tr>
<td>*7</td>
<td><code>endElement</code></td>
</tr>
<tr>
<td>*8</td>
<td><code>endDocument</code></td>
</tr>
</tbody>
</table>

Events are reported in **document reading order** *1, *2, ..., *8.

**N.B.:** Some events suppressed (white space).
SAX Callbacks

To provide an efficient and tight **coupling** between the SAX **frontend** and the application **backend**, the SAX API employs function **callbacks**.¹¹

1. Before parsing starts, the application **registers function references** in a table in which each event has its own slot:

<table>
<thead>
<tr>
<th>Event</th>
<th>Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>startElement</td>
<td>SAX register(startElement, startElement ())</td>
</tr>
<tr>
<td>endElement</td>
<td>endElement ()</td>
</tr>
</tbody>
</table>

2. The application alone decides on the implementation of the functions it registers with the SAX parser.

3. **Reporting an event** ∀ᵢ then amounts to call the function (with parameters) registered in the appropriate table slot.

¹¹Much like in event-based GUI libraries.
In Java, populating the callback table is done via implementation of the SAX ContentHandler interface: a ContentHandler object represents the callback table, its methods (e.g., public void endDocument()) represent the table slots.

Example: Reimplement content.cc shown earlier for DOM (find all XML text nodes and print their content) using SAX (pseudo code):

```java
content (File f)
    // register the callback,
    // we ignore all other events
    SAXregister (characters, printText);
    SAXparse (f);
    return;

printText ((Unicode) buf, Int len)
    Int i;
    foreach i ∈ 1...len do
        print (buf[i]);
    return;
```
Looking closer, the **order** of SAX events reported for a document is determined by a **preorder traversal** of its document tree\(^\text{12}\):

\begin{itemize}
\item \textbf{N.B.}: An \textit{Elem} [Doc] node is associated with two SAX events, namely \textit{startElement} and \textit{endElement} [\textit{startDocument}, \textit{endDocument}].
\end{itemize}

\(^{12}\)Sequences of sibling \textit{Char} nodes have been collapsed into a single \textit{Text} node.
**Challenge**

- This **left-first depth-first** order of SAX events is well-defined, but appears to make it hard to answer certain queries about an XML document tree.

✎ Collect all direct children nodes of an *Elem* node.

In the example on the previous slide, suppose your application has just received the `startElement(t = "a")` event \( \star_2 \) (i.e., the parser has just parsed the opening element tag `<a>`).

With the remaining events \( \star_3 \ldots \star_{16} \) still to arrive, can your code detect all the immediate children of *Elem* node `a` (i.e., *Elem* nodes `b` and `c` as well as the *Comment* node)?
The previous question can be answered more generally:

SAX events are sufficient to **rebuild the complete XML document tree inside the application.** (Even if we most likely don’t want to.)

---

**SAX-based tree rebuilding strategy (sketch):**

1. **[startDocument]**
   - Initialize a **stack** $S$ of **node IDs** (e.g. $\in \mathbb{Z}$). **Push** first ID for this node.

2. **[startElement]**
   - Assign a **new ID** for this node. **Push** the ID onto $S$.\(^\text{13}\)

3. **[characters, comment, ...]**
   - Simply assign a new node ID.

4. **[endElement, endDocument]**
   - **Pop** $S$ (no new node created).

\(^\text{13}\) In callbacks ② and ③ we might wish to store further node details in a table or similar summary data structure.

---

Invariant: The **top of** $S$ holds the identifier of the current **parent node**.
SAX Callbacks

SAX callbacks to rebuild XML document tree:

- We maintain a summary table of the form

<table>
<thead>
<tr>
<th>ID</th>
<th>NodeType</th>
<th>Tag</th>
<th>Content</th>
<th>ParentID</th>
</tr>
</thead>
</table>

- `insert (id, type, t, c, pid)` inserts a row into this table.

- Maintain stack $S$ of node IDs, with operations `push(id)`, `pop()`, `top()`, and `empty()`.

```plaintext
startDocument ()
  id ← 0;
  S.empty();
  insert (id, Doc, [], [], []);
  S.push(id);
  return;

endDocument ()
  S.pop();
  return;
```
4. SAX

SAX and XML Trees

SAX Callbacks

**startElement** \( (t, (a_1, v_1), \ldots) \)

\[
\begin{align*}
  id & \leftarrow id + 1; \\
  \text{insert} & \ (id, \ Elem, t, \Box, S.\text{top}()); \\
  S.\text{push}(id); \\
  \text{return};
\end{align*}
\]

**endElement** \( (t) \)

\[
\begin{align*}
  & S.\text{pop}(); \\
  & \text{return};
\end{align*}
\]

**characters** \( (buf, len) \)

\[
\begin{align*}
  id & \leftarrow id + 1; \\
  \text{insert} & \ (id, \ Text, \Box, buf[1 \ldots len], S.\text{top}()); \\
  \text{return};
\end{align*}
\]

**comment** \( (c) \)

\[
\begin{align*}
  id & \leftarrow id + 1; \\
  \text{insert} & \ (id, \ Comment, \Box, c, S.\text{top}()); \\
  \text{return};
\end{align*}
\]
Run against the example given above, we end up with the following summary table:

<table>
<thead>
<tr>
<th>ID</th>
<th>NodeType</th>
<th>Tag</th>
<th>Content</th>
<th>ParentID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Doc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Elem</td>
<td>a</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Elem</td>
<td>b</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Text</td>
<td></td>
<td>&quot;foo&quot;</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Comment</td>
<td></td>
<td>&quot;sample&quot;</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Elem</td>
<td>c</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Elem</td>
<td>d</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>Text</td>
<td></td>
<td>&quot;bar&quot;</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>Elem</td>
<td>e</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>Text</td>
<td></td>
<td>&quot;baz&quot;</td>
<td>8</td>
</tr>
</tbody>
</table>

Since XML defines tree structures only, the ParentID column is all we need to recover the complete node hierarchy of the input document.

**Walking the XML node hierarchy?**

Explain how we may use the summary table to find the (a) *children*, (b) *siblings*, (c) *ancestors*, (d) *descendants* of a given node (identified by its ID).
Path queries are the core language feature of virtually all XML query languages proposed so far (e.g., XPath, XQuery, XSLT, ...).

To keep things simple for now, let a path query take one of two forms (the \( t_i \) represent tag names):

\[
//t_1/t_2/\ldots/t_n \quad \text{or} \quad //t_1/t_2/\ldots/t_{n-1}/\text{text()}
\]

**Semantics:**

A path query selects a set of *Elem nodes* [with text(): Text nodes] from a given XML document:

1. The selected nodes have tag name \( t_n \) [are Text nodes].
2. Selected nodes have a parent *Elem* node with tag name \( t_{n-1} \), which in turn has a parent node with tag name \( t_{n-2} \), which ... has a parent node with tag name \( t_1 \) (not necessarily the document root element).
Examples:

1. Retrieve all **scene nodes** from a DilbertML document:
   
   ```xml
   //panels/panel/scene
   ```

2. Retrieve all character **names** from a DilbertML document:

   ```xml
   //strip/characters/character/text()
   ```

Path Query Evaluation

The summary table discussed in the previous section obviously includes all necessary information to evaluate both types of path queries.

**Evaluating path queries using the materialized tree structure.**

Sketch a summary table based algorithm that can evaluate a path query. (Use ```//a/c/d/text()``` as an example.)

- Note that, although based on SAX, such a path query evaluator would probably **consume as much memory as a DOM-based implementation.**
SAX-based path query evaluation (sketch):

1. Preparation:
   - Represent path query `//t_1/t_2/.../t_{n-1}/text()` via the step array
     `path[0] = t_1, path[1] = t_2, ..., path[n-1] = text()`.
   - Maintain an array index `i = 0...n`, the current step in the path.
   - Maintain a stack `S` of index positions.

2. `[startDocument]`
   Empty stack `S`. We start with the first step.

3. `[startElement]`
   If the current step’s tag name `path[i]` and the reported tag name match, proceed to next step. Otherwise make a failure transition\(^{14}\). Remember how far we have come already: `push` the current step `i` onto `S`.

4. `[endElement]`
   The parser ascended to a parent element. Resume path traversal from where we have left earlier: `pop` old `i` from `S`.

5. `[characters]`
   If the current step `path[i] = text()` we have found a match. Otherwise do nothing.

\(^{14}\)This “Knuth-Morris-Pratt failure function” `fail[]` is to be explained in the tutorial.
SAX-based path query evaluation (given step array \( path[0 \ldots n − 1] \)):

\[
\begin{align*}
\text{startDocument} & ( \quad ) \\
& \text{i} \leftarrow 0; \\
& \text{S. empty}(); \\
& \text{return;}
\end{align*}
\]

\[
\begin{align*}
\text{startElement} & ( \, t, (a_1, \nu_1), \ldots \, ) \\
& \text{S. push}(i); \\
\text{while true do} \\
& \quad \text{if } \text{path}[i] = t \text{ then} \\
& \quad \quad \text{i} \leftarrow i + 1; \\
& \quad \quad \text{if } i = n \text{ then} \\
& \quad \quad \quad \quad \text{★Match★;} \\
& \quad \quad \quad \quad \text{i} \leftarrow \text{fail}[i]; \\
& \quad \quad \text{break;}
\end{align*}
\]

\[
\begin{align*}
\text{characters} & ( \, \text{buf}, \text{len} \, ) \\
& \text{if } \text{path}[i] = \text{text}() \text{ then} \\
& \quad \text{★Match★;} \\
& \text{return;}
\end{align*}
\]

\[
\begin{align*}
\text{endElement} & ( \, t \, ) \\
& \text{i} \leftarrow \text{S. pop}(); \\
& \text{return;}
\end{align*}
\]

These SAX callbacks

1. evaluate a path query while we receive events (stream processing), and

2. operate without building a summary data structure and can thus evaluate path queries on documents of arbitrary size.

N.B.:
4. SAX
SAX and Path Queries

Tracing SAX Events . . .

Is there a bound on the stack depth we need during the path query execution?

Path Query (length \( n = 4 \)): //a/c/d/text()
\( path[0]=a, \ path[1]=c, \ path[2]=d, \ path[3]=\text{text}() \)

```
1. startDocument()
   \( i = 0 \)
   \( S = \square \square \)

2. startElement(\( t = a \))
   \( i = 1 \)
   \( S = 0 \square \square \)

3. startElement(\( t = b \))
   \( i = 0 \)
   \( S = 1 \square \square \)

4. characters("foo", 3)
   \( i = 0 \)
   \( S = 10 \square \)

5. endElement(\( b \))
   \( i = 1 \)
   \( S = 0 \square \square \)

6. comment("sample")
   \( i = 1 \)
   \( S = 0 \square \square \)

7. startElement(\( c \))
   \( i = 2 \)
   \( S = 10 \square \)

8. startElement(\( d \))
   \( i = 3 \)
   \( S = 210 \square \)

9. characters("bar", 3)
   \( i = 3 \)
   \( S = 210 \square \) ⋆Match⋆

10. endElement(\( d \))
    \( i = 2 \)
    \( S = 10 \square \)

11. startElement(\( e \))
    \( i = 0 \)
    \( S = 210 \square \)

12. characters("baz", 3)
    \( i = 0 \)
    \( S = 210 \square \)

13. endElement(\( e \))
    \( i = 2 \)
    \( S = 10 \square \)

14. endElement(\( c \))
    \( i = 1 \)
    \( S = 10 \square \)

15. endElement(\( a \))
    \( i = 0 \)
    \( S = 0 \square \square \)

16. endDocument()
```
Final Remarks on SAX

- For an XML document fragment shown on the left, SAX might actually report the events indicated on the right:

<table>
<thead>
<tr>
<th>XML fragment</th>
<th>XML + SAX events</th>
</tr>
</thead>
</table>
| 1  
| 2 <affiliation>  
| 3 AT&T Labs  
| 3 </affiliation>  | 1 <affiliation>\*1  
| 2 AT\*2 & \*3 T Labs  
| 3 \*4</affiliation>\*5 |

\*1 startElement(affiliation)  
\*2 characters("\nAT", 5)  
\*3 characters("&", 1)  
\*4 characters("T\nLabs\n", 7)  
\*5 endElement(affiliation)

- White space is reported.
- Multiple characters events may be sent for text content (although adjacent).
  (Often SAX parsers break text on entities, but may even report each character on its own.)