Part V

SAX—Simple API for XML

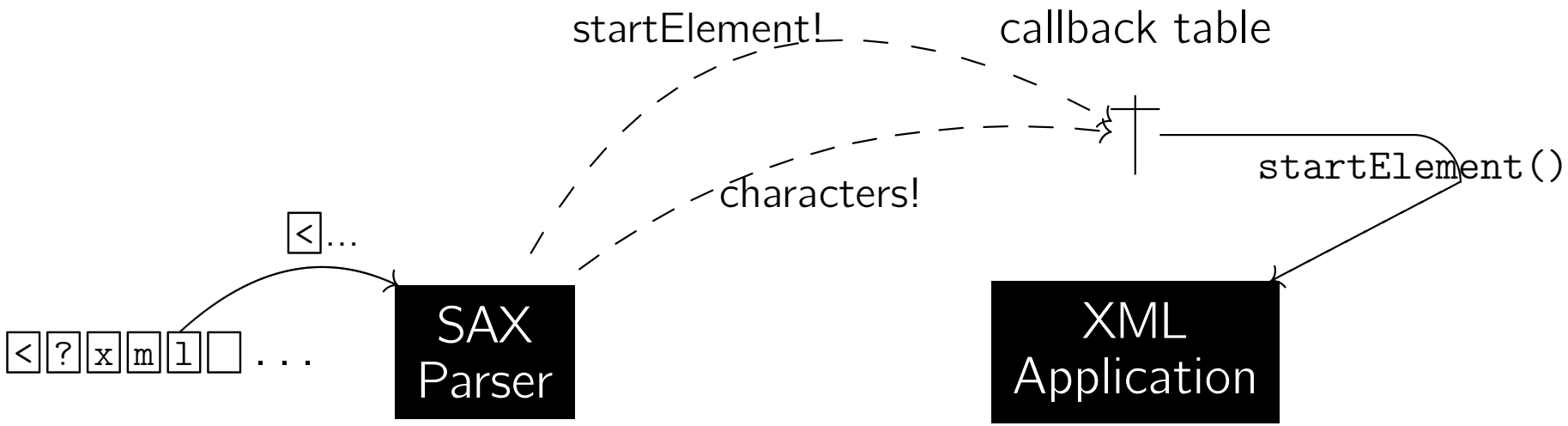
Outline of this part

SAX—Simple API for XML

- **SAX**[5] (**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 back-end 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 **back-end acts on/ignores events** by populating a **callback function table**.

[5] `http://www.saxproject.org/`

Sketch of SAX's mode of operations



- 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 progress.

SAX Events

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

Event	**. . . reported when seen**	**Parameters sent**
startDocument	<?xml. . . ?>[6]	
endDocument	⟨EOF⟩	
startElement	<t a_1=v_1 . . . a_n=v_n>	t, $(a_1, v_1), \ldots, (a_n, v_n)$
endElement	</t>	t
characters	*text content*	Unicode buffer ptr, length
comment	<!--c-->	c
processingInstruction	<?t pi?>	t, pi
	⋮	

[6] **N.B.**: Event *startDocument* is sent even if the optional XML text declaration should be missing.

dilbert.xml

```
1  <?xml encoding="utf-8"?> ⋆1
2  <bubbles> ⋆2
3    <!-- Dilbert looks stunned --> ⋆3
4    <bubble speaker="phb" to="dilbert"> ⋆4
5      Tell the truth, but do it in your usual engineering way
6      so that no one understands you. ⋆5
7    </bubble> ⋆6
8  </bubbles> ⋆7 ⋆8
```

Event[7] [8]		Parameters sent
$\star_1$	*startDocument*	
$\star_2$	*startElement*	t = "bubbles"
$\star_3$	*comment*	c = "␣Dilbert looks stunned␣"
$\star_4$	*startElement*	t = "bubble", ("speaker","phb"), ("to","dilbert")
$\star_5$	*characters*	*buf* = "Tell the...understands you.", *len* = 99
$\star_6$	*endElement*	t = "bubble"
$\star_7$	*endElement*	t = "bubbles"
$\star_8$	*endDocument*	

[7] Events are reported in **document reading order** $\star_1, \star_2, \ldots, \star_8$.

[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**:[9]
  1. Before parsing starts, the application **registers function references** in a table in which each event has its own slot:

Event	*Callback*
⋮	
startElement	?
endElement	?
⋮	

$\longrightarrow$ *SAX register*(*startElement*, `startElement ()`) *SAX register*(*endElement*, `endElement ()`)

Event	*Callback*
⋮	
startElement	`startElement ()`
endElement	`endElement ()`
⋮	

  2. The application alone decides on the implementation of the functions it registers with the SAX parser.
  3. **Reporting an event** $\star_i$ then amounts to call the function (with parameters) registered in the appropriate table slot.

[9] Much like in event-based GUI libraries.

Java SAX API

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):

content (*File f*)

// register the callback,
// we ignore all other events
SAX register (*characters*, *printText*);
SAX parse (*f*);
return;

printText ((*Unicode*) *buf*, *Int len*)

Int i;

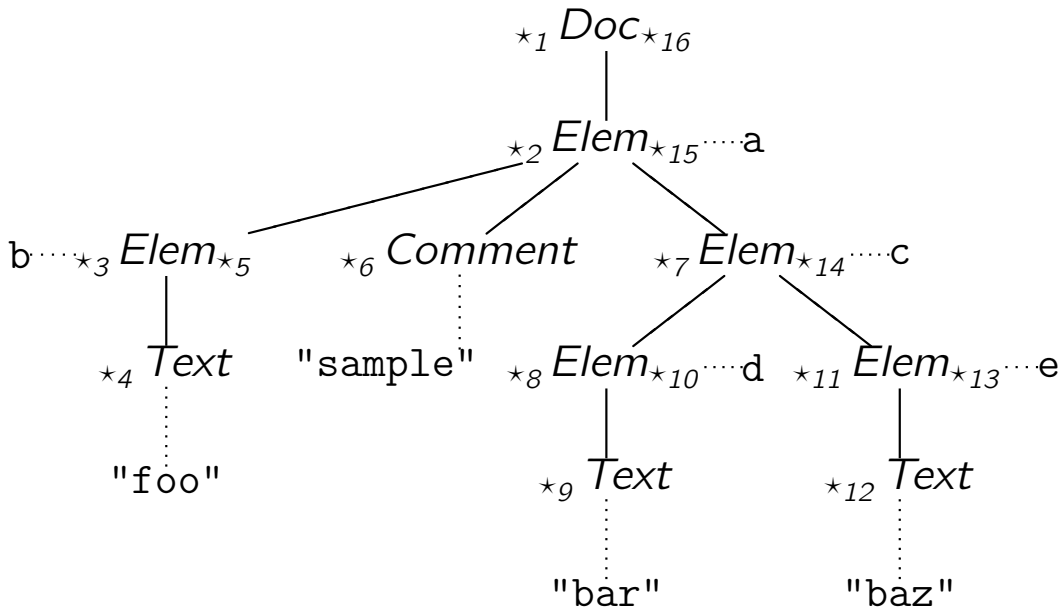
foreach $i \in 1 \ldots len$ **do**
  print (*buf*[*i*]);
return;

SAX and the XML Tree Structure

- Looking closer, the **order** of SAX events reported for a document is determined by a **preorder traversal** of its document tree[10]:

Sample XML document

```
1  ⋆1
2  <a>⋆2
3     <b>⋆3 foo⋆4 </b>⋆5
4     <!--sample-->⋆6
5     <c>⋆7
6       <d>⋆8 bar⋆9 </d>⋆10
7       <e>⋆11 baz⋆12 </e>⋆13
8     </c>⋆14
9  </a>⋆15 ⋆16
```



N.B.: An *Elem* [*Doc*] node is associated with two SAX events, namely *startElement* and *endElement* [*startDocument*, *endDocument*].

[10] Sequences of sibling *Char* nodes have been collapsed into a single *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.[11]
3. [*characters*, *comment*, ...]
   Simply assign a new node ID.
4. [*endElement*, *endDocument*]
   Pop S (no new node created).

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

[11] In callbacks ② and ③ we might wish to store further node details in a table or similar summary data structure.

SAX Callbacks

SAX callbacks to rebuild XML document tree:

- We maintain a summary table of the form

ID	NodeType	Tag	Content	ParentID

- *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 $empty()$.

startDocument ()

$id \leftarrow 0$;
$S.empty()$;
insert $(id, Doc, \square, \square, \square)$;
$S.push(id)$;
return;

endDocument ()

$S.pop()$;
return;

SAX Callbacks

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

$id \leftarrow id + 1;$
insert $(id, Elem, t, \square, S.top());$
$S.push(id);$
return;

endElement (t)

$S.pop();$
return;

characters (buf, len)

$id \leftarrow id + 1;$
insert $(id, Text, \square, buf[1 \ldots len], S.top());$
return;

comment (c)

$id \leftarrow id + 1;$
insert $(id, Comment, \square, c, S.top());$
return;

- Run against the example given above, we end up with the following summary table:

ID	NodeType	Tag	Content	ParentID
0	*Doc*	□	□	□
1	*Elem*	`a`	□	0
2	*Elem*	`b`	□	1
3	*Text*	□	`"foo"`	2
4	*Comment*	□	`"sample"`	1
5	*Elem*	`c`	□	1
6	*Elem*	`d`	□	5
7	*Text*	□	`"bar"`	6
8	*Elem*	`e`	□	5
9	*Text*	□	`"baz"`	8

- 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).

Final Remarks on SAX

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

XML fragment

```
1  <affiliation>
2    AT&amp;T Labs
3  </affiliation>
```

XML + SAX events

```
1  <affiliation>⋆1
2    AT⋆2&amp;⋆3T Labs
3  ⋆4</affiliation>⋆5
```

$\star_1$	*startElement*(`affiliation`)
$\star_2$	*characters*(`"\n␣␣AT"`, 5)
$\star_3$	*characters*(`"&"`, 1)
$\star_4$	*characters*(`"T␣Labs\n"`, 7)
$\star_5$	*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.)