Navigating Large XML Documents on Small Devices

Roger Peppe
Vita Nuova
April 2002

ABSTRACT

Browsing eBooks on platforms with limited memory presents an interesting problem: how can memory usage be bounded despite the need to view documents that may be much larger than the available memory. A simple interface to an XML parser enables this whilst retaining much of the ease of access afforded by XML parsers that read all of a document into memory at once.

Introduction

The Open Ebook Publication Structure was devised by the Open Ebook Forum in order to ''provide a specification for representing the content of electronic books''. It is based on many existing standards, notably XML and HTML. An Open eBook publication consists of a set of documents bound together with an Open eBook package file which enumerates all the documents, pictures and other items that make up the book

The underlying document format is essentially HTML compatible, which is where the first problem arises: HTML was not designed to make it easy to view partial sections of a document. Conventionally an entire HTML document is read in at once and rendered onto the device. When viewing an eBook on a limited-memory device, however, this may not be possible; books tend to be fairly large. For such a device, the ideal format would keep the book itself in non-volatile storage (e.g. flash or disk) and make it possible for reader to seek to an arbitrary position in the book and render what it finds there.

This is not possible in an HTML or XML document, as the arbitrarily nested nature of the format means that every position in the document has some unknown surrounding context, which cannot be discovered without reading sequentially through the document from the beginning.

SAX and DOM

There are two conventional programming interfaces to an XML parser. A SAX parser provides a stream of XML entities, leaving it up to the application to maintain the context. It is not possible to rewind the stream, except, perhaps, to the beginning. Using a SAX parser is fairly straightforward, but awkward: the stream-like nature of the interface does not map well to the tree-like structure that is XML. A DOM parser reads a whole document into an internal data structure representation, so a program can treat it exactly as a tree. This also enables a program to access parts of the document in an arbitrary order. The DOM approach is all very well for small documents, but for large documents the memory usage can rapidly grow to exceed the available memory capacity. For eBook documents, this is unacceptable.

A different approach

The XML parser used in the eBook browser is akin to a SAX parser, in that only a little of the XML structure is held in memory at one time. The first significant difference is that the XML entities returned are taken from one level of the tree - if the program does not wish to see the contents of a particular XML tag, it is trivial to skip over. The second significant difference is that random access is possible. This possibility comes from the observation that if we have visited a part of the document we can record the context that we found there and restore it later if necessary. In this scheme, if we wish to return later to a part of a document that we are currently at, we can create a ''mark'', a token that holds the current context; at some later time we can use that mark to return to this position.

The eBook browser uses this technique to enable random access to the document on a page-by-page basis. Moreover a mark can be written to external storage, thus allowing an external ''index'' into the document so it is not always necessary to read the entire document from the start in order to jump to a particular page in that document.

The programming interface

The interface is implemented by a module named `Xml`, which provides a `Parser` adt which gives access to the contents of an XML document. Xml items are represented by an `Item` pick adt with one branch of the pick corresponding to each type of item that might be encountered.

The interface to the parser looks like this:

```
open: fn(f: string, warning: chan of (Locator, string)): (ref Parser, string);
Parser: adt {
    next:       fn(p: self ref Parser): ref Item;
    down:   fn(p: self ref Parser);
    up:     fn(p: self ref Parser);
    mark:   fn(p: self ref Parser): ref Mark;
    atmark: fn(p: self ref Parser, m: ref Mark): int;
    goto:   fn(p: self ref Parser, m: ref Mark);
    str2mark:   fn(p: self ref Parser, s: string): ref Mark;
};
```

To start parsing an XML document, it must first be `open`ed; `warning` is a channel on which non- fatalerror messages will be sent if they are encountered during the parsing of the document. It can be nil, in which case warnings are ignored. If the document is opened successfully, a new `Parser` adt, say *p*, is returned. Calling *p*`.next` returns the next XML item at the current level of the tree. If there are no more items in the current branch at the current level, it returns `nil`. When a `Tag` item is returned, *p*`.down` can be used to descend ''into'' that tag; subsequent calls of *p*`.next` will return XML items contained within the tag, and *p*`.up` returns to the previous level.

An `Item` is a pick adt:

```
Item: adt {
    fileoffset: int;
    pick {
    Tag =>
        name:   string;
        attrs:      Attributes;
    Text =>
        ch:     string;
        ws1, ws2: int;
    Process =>
        target: string;
        data:       string;
    Doctype =>
        name:   string;
        public: int;
        params: list of string;
    Stylesheet =>
        attrs:      Attributes;
    Error =>
        loc:        Locator;
        msg:        string;
    }
};
```

`Item.Tag` represents a XML tag, empty or not. The XML fragments ''`<tag></tag>`'' and ''`<tag />`'' look identical from the point of view of this interface. A `Text` item holds text found in between tags, with adjacent whitespaces merged and whitespace at the beginning and end of the text elided. `Ws1` and `ws2` are non- zeroif there was originally whitespace at the beginning or end of the text respectively. `Process` represents an XML processing request, as found between ''`<?....?>`'' delimiters. `Doctype` and `Stylesheet` are items found in an XML document's prolog, the former representing a ''`<!DOCTYPE...>`'' document type declaration, and the latter an XML stylesheet processing request.

When most applications are processing documents, they will wish to ignore all items other than `Tag` and `Text`. To this end, it is conventional to define a ''front- end''function to return desired items, discard others, and take an appropriate action when an error is encountered. Here's an example:

```
nextitem(p: ref Parser): ref Item
{
    while ((gi := p.next()) != nil) {
        pick i := gi {
        Error =>
            sys->print("error at %s:%d: %s0,
                i.loc.systemid, i.loc.line, i.msg);
            exit;
        Process =>
            ;   # ignore
        Stylesheet  =>
            ;   # ignore
        Doctype =>
            ;   # ignore
        * =>
            return gi;
        }
    }
    return nil;
}
```

When `nextitem` encounters an error, it exits; it might instead handle the error another way, say by raising an exception to be caught at the outermost level of the parsing code.

A small example

Suppose we have an XML document that contains some data that we would like to extract, ignoring the rest of the document. For this example we will assume that the data is held within `<data>` tags, which contain zero or more `<item>` tags, holding the actual data as text within them. Tags that we do not recognize we choose to ignore. So for example, given the following XML document:

```
<metadata>
    <a>hello</a>
    <b>goodbye</b>
</metadata>
<data>
    <item>one</item>
    <item>two</item>
    <item>three</item>
</data>
<data>
    <item>four</item>
</data>
```

we wish to extract all the data items, but ignore everything inside the `<metadata>` tag. First, let us define another little convenience function to get the next XML tag, ignoring extraneous items:

```
nexttag(p: ref Parser): ref Item.Tag
{
    while ((gi := nextitem(p)) != nil) {
        pick i := gi {
        Tag =>
            return i;
        }
    }
    return nil;
}
```

Assuming that the document has already been opened, the following function scans through the document, looking for top level `<data>` tags, and ignoring others:

```
document(p: ref Parser)
{
    while ((i := nexttag(p)) != nil) {
        if (i.name == "data") {
            p.down();
            data(p);
            p.up();
        }
    }
}
```

The function to parse a `<data>` tag is almost as straightforward; it scans for `<item>` tags and extracts any textual data contained therein:

```
data(p: ref Parser)
{
    while ((i := nexttag(p)) != nil) {
        if (i.name == "item") {
            p.down();
            if ((gni := p.next()) != nil) {
                pick ni := gni {
                Text =>
                    sys->print("item data: %s0, ni.ch);
                }
            }
            p.up();
        }
    }
}
```

The above program is all very well and works fine, but suppose that the document that we are parsing is very large, with data items scattered through its length, and that we wish to access those items in an order that is not necessarily that in which they appear in the document. This is quite straightforward; every time we see a data item, we record the current position with a mark. Assuming the global declaration:

```
marks: list of ref Mark;
```

the `document` function might become:

```
document(p: ref Parser)
{
    while ((i := nexttag(p)) != nil) {
        if (i.name == "data") {
            p.down();
            marks = p.mark() :: marks;
            p.up();
        }
    }
}
```

At some later time, we can access the data items arbitrarily, for instance:

```
    for (m := marks; m != nil; m = tl m) {
        p.goto(hd m);
        data(p);
    }
```

If we wish to store the data item marks in some external index (in a file, perhaps), the `Mark` adt provides a `str` function which returns a string representation of the mark. `Parser`'s `str2mark` function can later be used to recover the mark. Care must be taken that the document it refers to has not been changed, otherwise it is likely that the mark will be invalid.

The eBook implementation

The Open eBook reader software uses the primitives described above to maintain display- page- based access to arbitrarily large documents while trying to bound memory usage. Unfortunately it is difficult to unconditionally bound memory usage, given that any element in an XML document may be arbitrarily large. For

instance a perfectly legal document might have 100MB of continuous text containing no tags whatsoever. The described interface would attempt to put all this text in one single item, rapidly running out of memory! Similar types of problems can occur when gathering the items necessary to format a particular tag. For instance, to format the first row of a table, it is necessary to lay out the entire table to determine the column widths.

I chose to make the simplifying assumption that top-level items within the document would be small enough to fit into memory. From the point of view of the display module, the document looks like a simple sequence of items, one after another. One item might cover more than one page, in which case a different part of it will be displayed on each of those pages.

One difficulty is that the displayed size of an item depends on many factors, such as stylesheet parameters, size of installed fonts, etc. When a document is read, the page index must have been created from the same document with the same parameters. It is difficult in general to enumerate all the relevant parameters; they would need to be stored inside, or alongside the index; any change would invalidate the index. Instead of doing this, as the document is being displayed, the eBook display program constantly checks to see if the results it is getting from the index match with the results it is getting when actually laying out the document. If the results differ, the index is remade; the discrepancy will hopefully not be noticed by the user!