Lab 1: Parsing XML in C

Due date: Tuesday, October 1, 11:59pm.

Lab Assignment

This is your first team assignment. It consists of two parts: synergistic activities and programming.

The purpose of the programming part of the assignment is for you to become comfortable with parsing and working with XML files in C.

Synergistic Activities

This lab includes the following team-building activities.

Team name. Come up with a team name.

Team repository and wiki page setup. Set your team up for work using the CSC 468 github repository. We will discuss the specific details for how the repository should be set up, but essentially, each team needs to have the following:

- Workspace. Part of the repository where the team will upload/maintain project-related code.
- Team Wiki Page. Will contain team information and links to documents produced throughout the project.

The wiki page should contain the team name and the list of team members.

ArferDB logo. Come up and create a drawing/rendering of ArferDB logo. Put an image of your logo on your team's Wiki page.

XML Parser

The CentOS install of `gcc` comes with `libxml`, a powerful XML/XPath/XSLT parser, that complies with a large number of WWW Consortium's XML-related recommendations, as well as with most of the Document Object Model (DOM).

The documentation for `libxml` can be found at http://www.xmlsoft.org/.

The Tasks

The programming assignment has dual goals of getting you comfortable with parsing XML, and preparing XML documents for storage as a collection of parent/child pairs.

You will create two programs. Both programs shall behave in the same way, take same types of inputs, and on the same inputs, produce the same output. However, the will be different in terms of the use of different `libxml` features.

Implementation 1: Using DOM/XML tree

Using `libxml`, write a program that performs as described below.

R0. Basics. Name the C file containing the `main()` function for your assignment `xmlBreakdown.c`. Your program shall take as input one command-line argument, a name of an XML file.

R1. libxml use. Your program shall use the `libxml` parser to parse the input XML file into an XML tree or a DOM object (your pick).

R2. Traversal. Once the XML tree is built, your program shall traverse it and report all *parent/child context pairs*.

A *parent/child context pair* is defined as follows:

- **Empty nodes.** If x is an empty XML node, then

$$x/$$

is the parent/child context pair corresponding to it.

- **Leaf nodes.** Let x be a leaf XML node with content is C. Then the corresponding parent/child context pair is

$$x/C$$

- **Root node.** Let r be the root node of the XML document. Then, the corresponding parent/child context pair is

$$/r$$

- **Parent/Child structure.** Let x be an XML node and let y be one of its children. Then, the corresponding parent/child context pair is

$$x/y$$

- **Attributes.** Let x be an XML node, and let a be its attribute. Then, the corresponding parent/child context pair is

$$x/a$$

- **Attribute values.** Let a be an attribute of some XML node, and let S be its value. Then, the corresponding parent/child context pair is

$$a/S$$

The XML tree shall be traversed in depth-first traversal order, and parent/child context pairs must be reported as soon as they are detected.

R3. Output. Your program shall output the following information:

- First line of output shall contain the name of the XML file being parsed.
- Second line shall be left blank.
- Starting with line three, report all discovered parent/child context pairs, one pair per line.
- After the last parent/child context pair is reported, skip one line.
- The last line of the output shall contain the total number of discovered parent/child context pairs.

R4. No Mixed XML. Your program shall assume that the input XML document has not mixed content, i.e., that all whitespace between XML elements is ignorable.

For example, in the following XML fragment:

```
<a>
  <b>1</b>
  <d>2</d>
</a>
```

XML element `a` has two child nodes: XML elements `b` and `d`. There are **no (!)** text nodes between `a` and `b`, between `b` and `d` and between `d` and the end of `a`.

Example. Consider the following simple XML file `f.xml`.

```
<?xml version="1.0"?>
<a>
 <v>1</v>
 <d/>
 <x s="1">true</x>
</a>
```

Executing your program with this file as input shall yield the following output:

```
f.xml

/a
```

```
a/v
v/"1"
a/d
d/
a/x
x/s
s/"1"
x/"true"

9
```

Implementation 2: Using SAX

The second program, named `xmlBreakdownSAX.c` shall satisfy the same requirements as `xmlBreakdown.c`, except requirement **R1** shall be replaced with the following requirement **R1+**.

R1+. `libxml` use. Use `libxml`'s SAX parser, and process each SAX call as they come.

The work and the output of the two programs shall coincide.

Why

`xmlBreakdown.c` is probably easier to write. Why bother with SAX? The real reason lies in the memory footprint of `xmlBreakdown.c`. The XML tree/DOM tree data structure created will be as big as the XML document. For *very large XML documents*, the resulting structure may not fit main memory. Also, building XML tree is slow, and since no traversal is performed until the tree is ready, tree construction effectively blocks useful work.

SAX parsers, on the other hand, require minimal footprint. In order to process XML documents using SAX parsers you must maintain a stack, whose depth will be roughly equal to the depth of the XML tree. For wide and shallow XML documents (Big Data encoded as XML), the size of the stack at each moment is bound by a small constant. Therefore you can process an XML document of ANY size. Also, SAX parsers allow for immediate output of results, and make pipelining straightforward and convenient.

Research

You will have built two different programs that perform the same computations. Time to compare their efficiency.

Using test XML documents provided to you by the instructor, as well as any other XML data found on-line, synthesized, or otherwise legally procured, *develop and execute a performance study* that compares the performance of `xmlBreakdown` and `xmlBreakdownSAX` programs.

Your study must test both programs on sizeable inputs. For the purpose of conducting the study, you can exclude the time it takes to print each individual line of output - that is, you can comment all `printf` statements in both programs before running them and computing their performance.

Submission Instructions

The submission is in two parts: `handin` and wiki.

Submit the following files using `handin`:

- `xmlBreakdown.c` and all other supporting `.c` and `.h` files.
- `xmlBreakdownSAX.c` and all other supporting `.c` and `.h` files.
- `README` file containing team name, the list of team members and compilation/running instructions if necessary. For each team member, indicate their contribution to the assignment[1]

Use the following `handin` command:

```
$ handin dekhtyar lab01-468 <files>
```

Submit the following to your team's wiki page:

- Your team's ArferDB logo. The higher the quality of the graphic, the better.
- A research paper documenting your performance study. The paper must be uploaded in the PDF format. I welcome using LaTeX for typesetting, but will accept a Word doc converted to PDF. The paper must be formatted as an academic research paper and contain a title, a list of authors with affiliations (Department of Computer Science, Cal Poly, San Luis Obispo, and your emails), a short abstract, a body of the paper broken into Introduction, Experimental Design, Results and Analysis/Conclusion sections, with a bibliography, *if needed* also included. The paper does not have to be large (2-3 pages of text plus whatever graphs/figures/visuals you provide in the Results section).

Good Luck!

[1] Each team gets the same grade, but I would like to know how each of you contributed to this assignment.