# CS6795 Labs and Assignments Documentation

***Release 1.0***

**Reuben Peter-Paul**

October 03, 2011

# CONTENTS

CHAPTER

ONE

# ASSIGNMENT 0

**Todo**

Using only a pencil and the left half of a paper, construct a random, non-trivial well-formed XML element with tag names x, y, z and sub(...sub)elements as follows: Write an x/y/z start-tag such as <x>, pronouncing it “angle, x, angle”; leave plenty of space and write, vertically below, the matching end-tag </x>, pronouncing it “angle, slash, x, angle” (with practice, you can also pronounce the ‘x-colored brackets’ <x> as “start-x” and </x> as “end-x”). Then fill in, indented by two blanks, another x/y/z start-tag such as <y>, pronouncing it “angle, y, angle” (“start-y”); leave some space and write, again vertically below, the matching end-tag </y>, pronouncing it “angle, slash, y, angle” (“end-y”). If there is more space left below the current subelement, proceed with the next subelement vertically below it; otherwise, proceed by filling in the space between two other pairs of matching tags, indented by two further blanks. Continue in this way to fill in the space between matching pairs of tags, repeatingly using tag names from the set {x, y, z}. However, instead of adding more tag pairs, you may also fill in natural-language phrases between matching pairs of tags.

## 1.1 Parts A, B

**Todo**

- Count the number of subelements of your generated XML element, including the main element in the sum. For each tag name x, y, z give the number of subelements using it.
- Write a Prolog term on the right half of your paper such that an XML tag pair like <x> ... </x> becomes a Prolog structure x( ... ). Align each constructor and its opening parenthesis such as x( with the corresponding start-tag <x>; align each matching closing parenthesis ) with the corresponding end-tag </x>. Put an XML natural-language phrase into double quotes (”) before its use in a Prolog structure.

## 1.2 Part C

**Todo**

Draw the node-labeled, (left-to-right-)ordered tree for which the XML element and equivalent Prolog structure are just two linearized representations. Hint: The vertical subelement/substructure extension corresponds to subtree breadth; the horizontal subelement/substructure indention corresponds to subtree depth.

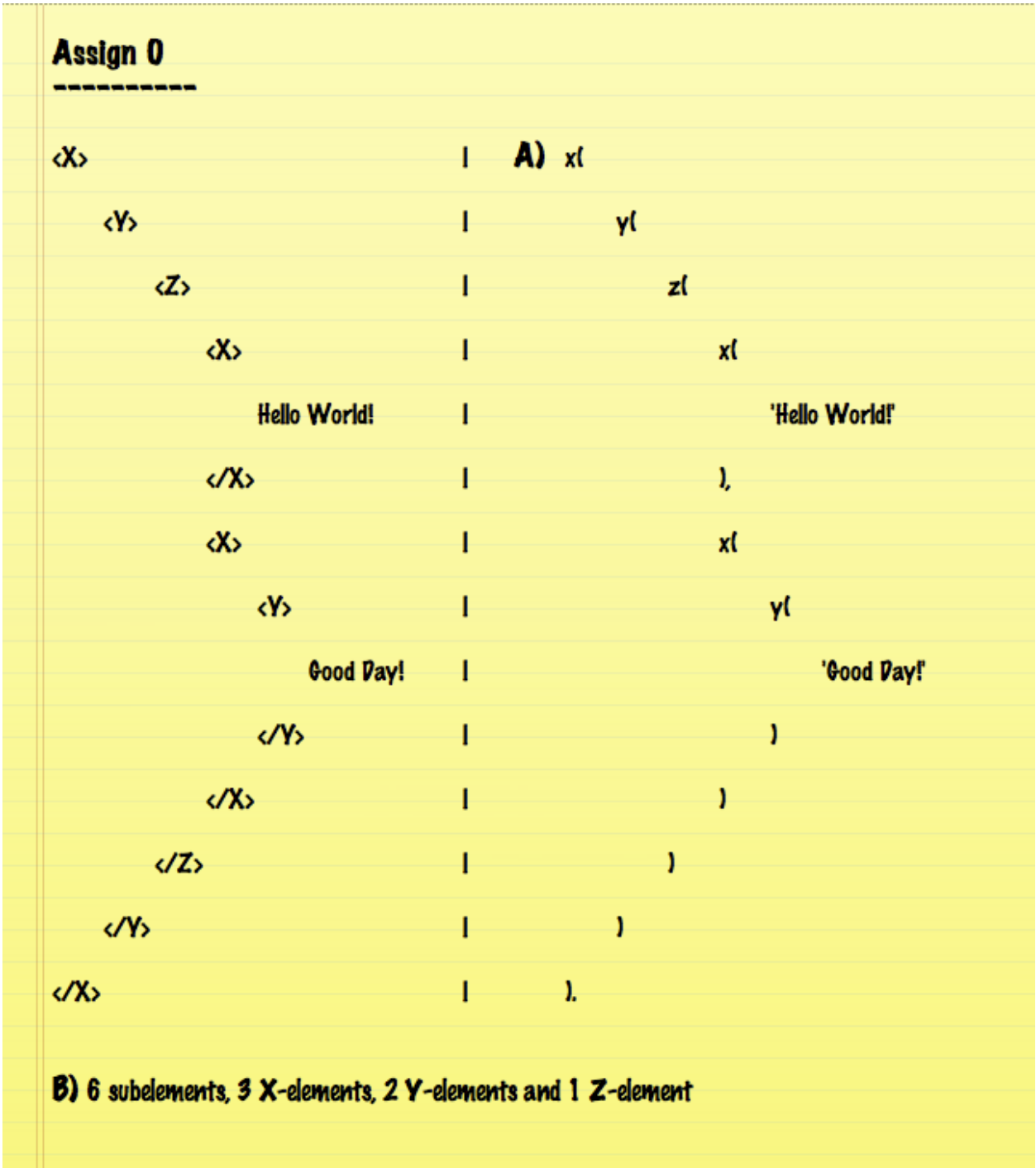
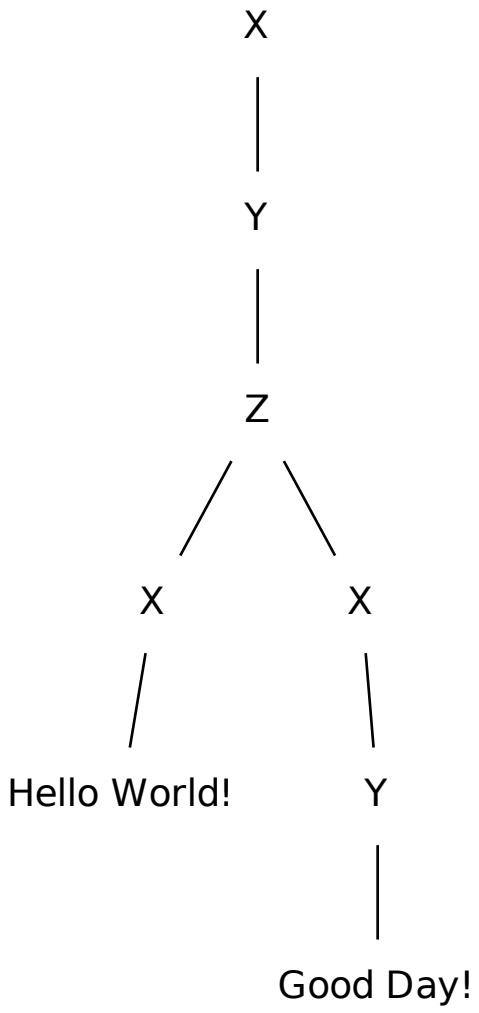



Figure 1.1: *Left:* Construction of a non-trivial *well-formed* XML instance using tag names `x`, `y`, and `z`. *Right:* Prolog term *pretty-printed* to align with the XML tag elements on the left. *Bottom:* Answer to part **A** of assignment 0.




Visualization of the XML isntance tree structure, *rendered using Graphviz*.

## 1.3 Part D

**Todo**

List the notational (dis)advantages of the XML and Prolog representations.

The notational advantages of xml representations are generally well known among programmers and are many (cf. [JSON]):

- XML is human readable.

- XML can be used as an exchange/interchange format to enable users to mover thier data between similar applications.
- XML provides structure to data so taht it is richer in information.
- XML is easily processed because of the structure of the data is simple and standard.
- There is a wide range of reusable software available to programmers to handle XML so they don't have to re-invent code.
- XML seperates the presentation of data from the structure of that data.
- Many views of the one data.
- Self-describing data.
- Complete integration of all traditional databases and formats.
- Internationalization.
- Open and extensible.
- XML is widely adopted by the computer industry.

The notational disadvantages of xml representations are also generally well know to programmers (cf. [Tolf11]):

- DOM is too specialized.
- Can be cumbersome and inefficient.
- Does not map well to data types of most programming languages.

The notational advantages of prolog representations are similar to those of XML in that they are human readable, they provide a structure to the data so that it is richer in information.

The notational disadvantages of prolog are inherent to all dynamically checked programming languages, e.g.: type checking done at run-time, only one datatype (`term`) (cf. [Wp1]).

## 1.4 Part E

---

**Todo**

Can anything be logically wrong with the legal Prolog structures corresponding to arbitrary XML elements that use repeated tags from {x, y, z}? Hint: Consider ways in which not only XML but also Prolog is "less formal" than logic (types, modes, arities, ...).

---

XML and Prolog are less formal and their type specifications are much more relaxed than formal logics. According to [Brac04] (chapter 2, p. 25) the aim of formal logics is to build up a *Logical Model* such that the set of possible interpretations is made more narrow so as to rule out more and more unintended interpretations. Ultimately, logical consequene itself will tend toward "'truth in the intended interpretation"'. The *document object mode (DOM)* implemented by XML does not provide semantics for "logical implications/inference/entailment" the relationships between elements are structural only, and while a term can be expressed in *Prolog* that mimics the structure of an XML document queries against such terms are seem to be *meaningless* and *useless* since the inner terms are inaccessbile to the `top loop`.

Prolog REPL Sample

```
?- ['Assign0.pl'].
% Assign0.pl compiled 0.00 sec, 1,768 bytes
true.

?- listing.

a(b(c('hello world'))).

x(y(z(x('hello world'), x(y(hi))))).
true.

?- c(X).
ERROR: toplevel: Undefined procedure: c/1 (DWIM could not correct goal)
?- z(X).
ERROR: toplevel: Undefined procedure: z/1 (DWIM could not correct goal)
```

CHAPTER

TWO

# LAB 1 - CREATE YOUR OWN XML DTD, XSD, AND RNC OR RNG

**Todo**

Consider these examples of XML documents for clause sets consisting of zero or more facts f (Prolog: `f.`) and zero or more 'backward' rules c <- p (Prolog: `c :- p.`), in any order ("myurl" will be replaced as explained below):

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE clauses SYSTEM "myurl">
 <clauses>
    <fact> f </fact>
    <rule> <conc> c </conc> <prem> p </prem> </rule>
 </clauses>

<?xml version="1.0" standalone="no"?>
<!DOCTYPE clauses SYSTEM "myurl">
<clauses>
    <rule> <conc> c1 </conc> <prem> p1 </prem> </rule>
    <fact> f1 </fact>
    <rule> <conc> c2 </conc> <prem> p2 </prem> </rule>
    <fact> f2 </fact>
    <fact> f3 </fact>
</clauses>
```

Inductively complete this XML DTD (overwrite the "..." lines) for such clause sets:

```
<!ELEMENT clauses   (.................)>
<!ELEMENT rule   (.................)>
<!ELEMENT fact    (#PCDATA)>
<!ELEMENT ........    (.................)>
<!ELEMENT ........    (.................)>
```

## 2.1 DTD

```
<!ELEMENT clauses (fact|rule)*>
<!ELEMENT rule                (conc,prem)>
<!ELEMENT fact                (#PCDATA)>
<!ELEMENT conc                (#PCDATA)>
<!ELEMENT prem                (#PCDATA)>
```

## 2.2 XSD

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="clauses">
    <xs:complexType>
      <xs:sequence>
        <xs:choice maxOccurs="unbounded">
          <xs:element ref="rule"/>
          <xs:element ref="fact"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="rule">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="conc"/>
        <xs:element ref="prem"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="fact" type="xs:string"/>
  <xs:element name="conc" type="xs:string"/>
  <xs:element name="prem" type="xs:string"/>
</xs:schema>
```

## 2.3 RelaxNG

Compact syntax:

```
default namespace = ""

start = clauses
clauses = element clauses {
  (element rule {
    (element conc {xsd:string},
    element prem {xsd:string})}
  | (element fact {xsd:string}))*}
```

---

**Note:** I used trang to transform RelaxNG compact syntax into xml-based syntax.

---

XML syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar ns="" xmlns="http://relaxng.org/ns/structure/1.0" datatypeLibrary="http://www.w3.org/2001/XM
  <start>
    <ref name="clauses"/>
  </start>
  <define name="clauses">
    <element name="clauses">
      <zeroOrMore>
        <choice>
          <element name="rule">
```

```
            <element name="conc">
              <data type="string"/>
            </element>
            <element name="prem">
              <data type="string"/>
            </element>
          </element>
          <element name="fact">
            <data type="string"/>
          </element>
        </choice>
      </zeroOrMore>
    </element>
  </define>
</grammar>
```

## 2.4 Misc

---

**Note:** To validate the above sample instances I used xmllint - command line XML tool to parse and type-check/validate:

```
$ xmllint --noout --schema http://reubenpeterpaul.github.com/lab1/XSD/clauses.xsd clauses-instance.xm
$ xmllint --noout --dtdvalid http://reubenpeterpaul.github.com/lab1/DTD/clauses.dtd clauses-instance.
$ trang clauses.rnc clauses.rng
$ xmllint --noout --relaxng http://reubenpeterpaul.github.com/lab1/DTD/clauses.rng clauses-instance.x
```

---

---

**Note:** I also used my own custom XML catalog file:

```
<?xml version="1.0"?>
<!DOCTYPE catalog
PUBLIC "-//OASIS/DTD Entity Resolution XML Catalog V1.0//EN"
"http://www.oasis-open.org/committees/entity/release/1.0/catalog.dtd">
<catalog
xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">

...

  <system
  systemId="http://reubenpeterpaul.github.com/cs6795/lab1/DTD/clauses.dtd"
  uri="/home/peter-paulr/.laboratory/cs6795/lab1/clauses.dtd"
  />

  <system
  systemId="http://reubenpeterpaul.github.com/cs6795/lab1/XSD/clauses.xsd"
  uri="/home/peter-paulr/.laboratory/cs6795/lab1/clauses.xsd"
  />

  <system
  systemId="http://reubenpeterpaul.github.com/cs6795/lab1/RNG/clauses.rng"
  uri="/home/peter-paulr/.laboratory/cs6795/lab1/clauses.rng"
  />
</catalog>
```

---

CHAPTER

THREE

# ASSIGNMENT 1

## 3.1 Part A-0

**Todo**

Do the missing leaf data for passengers cause any (XML-level) issue?

No the the missing leaf data (`#PCDATA`) does not affect the xml well-formedness.

## 3.2 Part A-1

**Todo**

Does the repeatedly occurring `appellation`:

```
appellation('Main St').     appellation('Fredericton').
```

label cause a problem with respect to the unambiguous representation of all parts of the taxi-ride information? Hint: Is it even necessary to check whether every leaf of the tree - ordered from 37 to 12.50 - are uniquely denoted by the "path name" of node labels leading to it from the taxi-ride root? What about any other repeatedly occurring label?

To determine potential ambiguity introduced by the repeated use of the `appellation` tag-label consider the following prolog system:

```
appellation('Main St').
appellation('Fredericton').
first('Peter').
mid('C.').
lastn('Jones').
numbern(37).
numbern(12).
province('NB').

name(X,Y,Z) :- first(X), mid(Y), lastn(Z).
driver(W,X,Y,Z) :- numbern(W), name(X,Y,Z).
municipality(X,Y) :- appellation(X), province(Y).
street(X,Y) :- number(X), appellation(Y).
```

Backtracking in prolog against the query `street(X,Y)` yields the following results:

```
?- ['testA2'].
% testA2 compiled 0.00 sec, 5,624 bytes
true.

?- street(X,Y).
X = 37,
Y = 'Main St' ;
X = 37,
Y = 'Fredericton' ;
X = 12,
Y = 'Main St' ;
X = 12,
Y = 'Fredericton'.
```

This illustrates by example that the repeated use of `appellation` and also `number` results in adding ambiguity to the representation of `taxi-ride`. This problem can be dealt with via introduction of complex types, or denoting the leaf nodes with the path of node-labels from the root `taxi-ride`, e.g.:

```
street(numbern(12), appellation('Main St')).
municipality(appellation('Fredericton'), province('NB')).
```

## 3.3 Part A-2

**Todo**

What if the nodes labeled 'appellation' would have been labeled 'name', too?

Again let us consider a prolog system:

```
name('Main St').
name('Fredericton').
first('Peter').
mid('C.').
lastn('Jones').
numbern(37).
numbern(12).
province('NB').

name(X,Y,Z) :- first(X), mid(Y), lastn(Z).
driver(W,X,Y,Z) :- numbern(W), name(X,Y,Z).
municipality(X,Y) :- name(X), province(Y).
street(X,Y) :- number(X), name(Y).
```

Backtracaking in prolog REPL against the query `street` and `driver` yields the following results:

```
?- consult('testA2b').
% testA2b compiled 0.00 sec, 5,096 bytes
true.


?- street(X,Y).
X = 37,
Y = 'Main St' ;
X = 37,
Y = 'Fredericton' ;
X = 12,
```

```
Y = 'Main St' ;
X = 12,
Y = 'Fredericton'.

?- driver(W,X,Y,Z).
W = 37,
X = 'Peter',
Y = 'C.',
Z = 'Jones'.
W = 12,
X = 'Peter',
Y = 'C.',
Z = 'Jones'.
```

The result is the same as above (*Part A-1*). `Node-labelled` paths are still required to denotate the `name` tags used by `street` and `municipality` nodes. While the rule for `name` satisfies the complex conditions for separating the parts of an individuals name.

## 3.4 Part A-3 - Relabeling of tree in Prolog

```
carnumber(37).
firstname('Peter').
middlename('C.').
lastname('Jones').
date('9/27').
streetnumber(12).
streetname('Main St').
municipalityname('Fredericton').
provincename('NB').
fare(12.50).

taxiride(W,X,Y,Z,D,E,F,G,H,I) :-
  driver(W,X,Y,Z),
  passengers,
  transportinformation(D,E,F,G,H,I).

driver(W,X,Y,Z) :- carnumber(W), fullname(X,Y,Z).

fullname(X,Y,Z) :- firstname(X), middlename(Y), lastname(Z).

transportinformation(D,E,F,G,H,I) :- date(D), destination(E,F,G,H),
fare(I).

destination(E,F,G,H) :- street(E,F), municipality(G,H).

street(X,Y) :- streetnumber(X), streetname(Y).

municipality(X,Y) :- municipalityname(X), provincename(Y).
```

## 3.5 Part B-3 - XML instance of relabeling

```
<taxiride>
  <driver>
```

```
    <carnumber>37</carnumber>
    <fullname>
      <firstname>Peter</firstname>
      <middlename>C.</middlename>
      <lastname>Jones</lastname>
    </fullname>
  </driver>
  <passengers />
  <transportinformation>
    <date>9/27</date>
    <detination>
      <street>
        <streetnumber>12</streetnumber>
        <streetname>Main St</streetname>
      </street>
      <municipality>
        <municipalityname>Fredericton</municipalityname>
        <provincename>NB</provincename>
      </municipality>
    </destination>
    <fare>12.50</fare>
  </transportinformation>
</taxiride>
```

## 3.6 Part C - Prolog equivalent term of *Part B-3 - XML instance of relabeling*

```
taxiride(
  driver(
    carnumber(37),
    fullname(
      firstname('Peter'),
      middlename('C.'),
      lastname('Jones'))),
  passengers,
  transportinformation(
    date('9/27'),
    destination(
      street(
        streetnumber(12),
        streetname('Main St')),
      municipality(
        municipalityname('Fredericton'),
        provincename('NB'))),
    fare(12.50))).
```

# BIBLIOGRAPHY


[JSON] JSON the Fat Free Alternative to XML, Introducing JSON, 21 September 2011, <http://www.json.org/xml.html>.

[Tolf11] Why XML is bad for representing arbitrary data, Home Page of Fredrik Tolf, 21 September 2011, <http://dolda2000.com/~fredrik/doc/xmlds>.

[Wp1] Wikipedia: Prolog, Wikipedia.org, 21 September 2011, <http://en.wikipedia.org/wiki/Prolog>.

[Brac04] Brachman R.J., and Levesque H.J., *Knowledge Representation and Reasoning*. San Francisco, CA: Elsevier, 2004.