CS2 Current Technologies note 4

Document Type Descriptors

Peter Buneman

4.1 Introduction

A Document Type Descriptor (DTD) describes the *structure* of an XML document. In writing queries/stylesheets for XML we have made certain assumptions about the structure. A DTD can express some of these assumptions.

There is some relationship between a DTD and the type system of a programming language. For example, in Java, when you write `x.f` the class/type of `x` tells you (and the Java compiler) that `x` had a data member called `f`. Similarly, you can use a DTD to tell you that a given kind of element has a child element with a tag named `f`. However (except for a few experimental systems) the DTD does not tell XPath, XSLT, XQuery, or anything else that processes the XML document about the type. It is up to you – the programmer – to check that the program is sensible with respect to the DTD.

There are other things that distinguish a DTD from a conventional type system. The DTD is used to specify the *linear* structure of documents. When you write a Java class, the order in which you declare the methods and data members is, I believe, unimportant. However the order in which things occur in a conventional document is profoundly important.

DTDs also allow us to specify a rather primitive kind of "pointer" which allows elements of a document to reference other elements in that document.

However, DTDs are rather weak in other respects. There is only one base type, which is PCDATA or text. You cannot for example constrain the contents of some element to be an integer or the textual representation of an integer. The pointers mentioned above cannot be "typed". You cannot constrain a pointer to point to an element with a given tag name or structure. You cannot describe any notion of inheritance between types.

Another important thing to remember is that we type elements by their tag name. In programming languages they are determined by some form of scoping or context. For example, the `title` member of a `book` class in Java could have a completely different structure to the `title` member of a `person` class. In XML specified by a DTD, all `title` elements must have the same structure. There are "work-arounds" for this in DTDs, but they are not satisfactory.

Some of these deficits have been corrected in XML Schema. XML Schema can express a rather large number of constraints. However it suffers from being too complicated, and the interaction between the constraints you can specify in it is poorly understood. If you are courageous enough

to study XML schema, you will in any case have to understand how DTDs work, so studying them first is a good idea.

4.2 DTDs and regular expressions

I have a 25 year old format for addresses which is easily converted into XML. If this were done, one entry in the address book would look like this.

```
⟨person⟩
    ⟨name⟩ McNeil, John ⟨/name⟩          must exist
    ⟨greet⟩ Dr. John McNeil ⟨/greet⟩     optional
    ⟨addr⟩ 1234 Huron Street ⟨/addr⟩     as many address lines as needed
    ⟨addr⟩ Rome, OH 98765 ⟨/addr⟩
    ⟨tel⟩ (321) 786 2543 ⟨/tel⟩          0 or more tel and faxes in any order
    ⟨fax⟩ (123) 456 7890 ⟨/fax⟩
    ⟨tel⟩ (321) 198 7654 ⟨/tel⟩
    ⟨email⟩ jm@abc.com ⟨/email⟩          0 or more email addresses
⟨/person⟩
```

The specification of this structure is via a *regular expression* with a somewhat changed syntax. For example:

`name`	to specify a `name` element
`greet?`	to specify an optional (0 or 1) `greet` elements
`name,greet?`	to specify a name followed by an optional `greet`
`addr*`	to specify 0 or more `address` lines
`tel | fax`	a `tel` or a `fax` element
`(tel | fax)*`	0 or more repeats of `tel` or `fax`
`email*`	0 or more email elements

So the whole structure of a person entry is specified by

```
name, greet?, addr*, (tel | fax)*, email*
```

You should be able to describe an equivalent deterministic automaton. The W3C DTD standard describes a restriction on DTDs – essentially a one-"character" look-ahead – that makes them easy to parse. We do not need to worry about this restriction now.

In a DTD we specify that a `person` element has this structure by writing:

```
⟨!ELEMENT person (name, greet?, addr*, (fax|tel)*, email*)⟩
```

The whole DTD for the address book becomes:

```
⟨!DOCTYPE address [
⟨!ELEMENT addressbook (person*)⟩
⟨!ELEMENT person (name, greet?, addr*, (fax|tel)*, email*)⟩
⟨!ELEMENT name (#PCDATA)⟩
⟨!ELEMENT greet (#PCDATA)⟩
⟨!ELEMENT addr (#PCDATA)⟩
⟨!ELEMENT tel (#PCDATA)⟩
⟨!ELEMENT fax (#PCDATA)⟩
⟨!ELEMENT email (#PCDATA)⟩
]⟩
```

Representing tabular data. In the coming lectures we are going to study relational databases. The underlying idea is that we represent data as tables. For example, we might have two tables to represent projects and employees:

Projects:

title	budget	manager
XML types	100000	Jane
...	...	...

Employees:

name	empid	age
Joe	1234	34
...	...	...

Now a common use for XML (in fact its *raison d'être*) is to ship data around, so we should be able to come up with a good way of turning these tables into XML. Unfortunately there are *several* ways. We could send a mixture of employees and projects:

```
⟨!DOCTYPE db [
⟨!ELEMENT  db (project | employee)*⟩
⟨!ELEMENT  project (title, budget, managedBy)⟩
⟨!ELEMENT  employee (name, empid, age)⟩
⟨!ELEMENT  title #PCDATA⟩
...
]⟩
```

We could group them (this may be closest in spirit to the tabular form)

```
⟨!DOCTYPE db [
⟨!ELEMENT  db (projects,employees)⟩
⟨!ELEMENT  projects (project*)⟩
⟨!ELEMENT  employees (employee*)⟩
⟨!ELEMENT  project (title, budget, manager)⟩
⟨!ELEMENT  employee (name, empid, age)⟩
...
]⟩
```

We could remove all the grouping tags:

```
⟨!DOCTYPE db [
⟨!ELEMENT  db )((name, empid, age)|(title, budget, manager))*)⟩
```

```
...
]⟩
```

All of these are possible, and each has its advantages.

Recursive DTDs These are surprisingly common. The current course project has one; even the DTD for Shakespeare's plays is recursive. Here is another (flawed) example:

```
⟨!DOCTYPE  genealogy [
⟨!ELEMENT  genealogy (person*)⟩
⟨!ELEMENT  person (
name,
dateOfBirth,
person,          // mother
person           // father
)⟩
]⟩
```

It should be obvious what is intended, but can you see what is wrong with this? And how would you fix it?

A problem we encounter when dealing with transmitting traditional data is that the order in which we send, say, the fields of a tuple (a row of a table or a record) is immaterial. If we want to say that an employee element contains `name`, `age` and `empid` elements in *some order* we have to say:

```
⟨!ELEMENT employee (
(name, age, empid)
| (age, empid, name)
| (empid, name, age)
...
)⟩
```

Tables with 50 columns are common! What does one do short of saying ⟨`!ELEMENT employee ANY`⟩, which allows the `employee` tag to have anything you like in it. Related to this is the fact that you cannot say (with a DTD) that an element should have *at least* a specified set of child elements. This is needed for subtyping, which is common in relational databases. These issues are fixed to some extent in XML schema.

It is common to find real DTDs going off the rails. This was taken from a repository of published DTDs:

```
⟨!ELEMENT PARTNER (NAME?, ONETIME?, PARTNRID?, PARTNRTYPE?, SYNCIND?, ACTIVE?, CURRENCY?,
DESCRIPTN?, DUNSNUMBER?, GLENTITYS?, NAME*, PARENTID?, PARTNRIDX?, PARTNRRATG?,
PARTNRROLE?, PAYMETHOD?, TAXEXEMPT?, TAXID?, TERMID?, USERAREA?, ADDRESS*, CONTACT*)⟩
```

What is wrong with this?

4.3 IDs and IDREFs – "pointer integrity"

This is a rather feeble mechanism, but we should mention it briefly. The idea is that an `ID` attribute is a unique identifier or *key* for an element and an `IDREF` type is a reference or "pointer" to a key. Look at the following example:

```
⟨family⟩
    ⟨person id="jane" mother="mary" father="john"⟩
        ⟨name⟩ Jane Doe ⟨/name⟩     ⟨/person⟩
    ⟨person id="john" children="jane jack"⟩
        ⟨name⟩ John Doe ⟨/name⟩
    ⟨/person⟩
    ⟨person id="mary" children="jane jack"⟩
        ⟨name⟩ Mary Doe ⟨/name⟩
    ⟨/person⟩
    ⟨person id="jack" mother="mary" father="john"⟩
        ⟨name⟩ Jack Doe ⟨/name⟩
    ⟨/person⟩
⟨/family⟩
```

The DTD that constrains these attributes is:

```
⟨!DOCTYPE family [
   ⟨!ELEMENT  family (person)*⟩
   ⟨!ELEMENT  person (name)⟩
   ⟨!ELEMENT  name (#PCDATA)⟩
   ⟨!ATTLIST person
      id       ID     #REQUIRED
      mother   IDREF  #IMPLIED
      father   IDREF  #IMPLIED
      children IDREFS #IMPLIED⟩
]⟩
```

- If an attribute is declared as `ID` the associated values must all be *distinct* (no confusion).
- If an attribute is declared as `IDREF` the associated value *must exist* as the value of some `ID` attribute (no "dangling pointers").
- Similarly for all the values of an `IDREFS` attribute

Note that `ID` and `IDREF` attributes are *not typed*. We can't require `ID` attribute to point to a particular kind of attribute.

Some important terms. An XML document is

- *well formed* if it "parses" (it observes the nested tags rule, attributes are unique, etc.) and
- *valid* if it satisfies a DTD. It is better to say say "valid with respect to ...".