Database Theory: Exercise Sheet 2

Dr G.M. Bierman
gmb@cl.cam.ac.uk

February 2, 2004

Part I
Complex value model

Throughout the lecture course we have made use of a simple database schema concerning films and cinemas. In lecture 6 we realised that there can be redundancy in the database and we learnt about a richer model of data, the complex value model. Assume that we replace the cinema database schema from the notes with the following complex value schema.

```
Movies   : <Title: dom, Director: {dom}, Actor: {dom}>
Location : <Cinema: dom, Town: dom, Contacts: {<Number:dom>}>
Guide    : <Cinema: dom, Title: dom, Times: {dom}>
```

Write the following queries in ALGcv. (You may used derived operators provided that you give their definitions.)

1. Directors of film called Magnolia
2. Names of movies directed by Almodovar
3. Names of movies where Stallone is both an actor and a director
4. Tel numbers of cinemas in Cambridge
5. Times of films being shown at the Cambridge Picturehouse
6. Times of films being shown where Anderson is the director
7. Names of actors in all films currently being shown in Cambridge
8. Names of actors in films being shown at 21:00

Part II
Schema and XML

1 Background

In lecture 8 I motivated semi-structured data (SSD) with the need for handling data that does not have a fixed schema. This is nice, but there are many circumstances when we *will* have a

schema (e.g. data exchange) and so we would like to be able to both express the schema in some language, and also check that the data adheres to the schema.

For SSD the big problem is dealing with the constraining nature of schema, with the intended flexibility of the model. For example, consider the following XML:

```
<IMDB>
  <Movies>
    <Movie>
     <title>Magnolia</title>
     <year>1999</year>
     <stars>
       <star>Tom Cruise</star>
     </stars>
    </Movie>
  <Movies>
</IMDB>
```

A simple schema for this might be:

```
schema IMDB  ::= <Movies> of Movie;
schema Movie ::= <Movie> of title*year*stars;
schema title ::= <title> of string;
schema year  ::= <year> of integer;
schema stars ::= <stars> of star;
schema star  ::= <star> of string;
```

But this isn't very flexible. Here are some problems:

1. We intended to be able to store *many* movies in the database. Thus we'd like to be able to store a **sequence** of `Movie` data. A well-formed piece of XML data should match this schema irrespective of the size of the sequence.

2. Sometimes we don't know a piece of data, e.g. we may not know the `year`. We'd still want the XML to match the schema. Thus we need **optional** pieces of data.

3. What do we mean by `integer`? Every machine has a different notion. Let's make a simplifying assumption and only assume one datatype: the string!

This leads us essentially to a **DTD** (Document Type Definition), which is a standard for defining schema for XML documents.[1] Although it looks easy, we'll see that there are complications lurking. Here is a possible DTD for the data given earlier:

```
<!DOCTYPE IMDB [
 <!ELEMENT Movies (Movie)*>
 <!ELEMENT Movie  ((title,stars)|(title,year,stars))>
 <!ELEMENT title  (#PCDATA)>
 <!ELEMENT year   (#PCDATA)>
 <!ELEMENT stars  (star)+>
 <!ELEMENT star   ((name)|(name,age))>
 <!ELEMENT name   (#PCDATA)>
 <!ELEMENT age    (#PCDATA)>
]>
```

[1] You saw these in the IB Databases course. Re-read those notes if you need further details.

Notice the sequencing (`*` for zero or more, `+` for one or more), optional definitions (`|`), and the string datatype (`#PCDATA`).

2 Exercises

1. Consider the data:

```
<foo>
  <pop>Britney</pop>
  <pop>Christina</pop>
</foo>
```

   (a) Does this data validate against the following DTD?

```
<!ELEMENT foo (pop+)>
<!ELEMENT pop (#PCDATA)>
```

   (b) Does this data validate against the following DTD?

```
<!ELEMENT foo (pop*)>
<!ELEMENT pop (#PCDATA)>
```

   (c) Does this data validate against the following DTD?

```
<!ELEMENT foo (pop)>
<!ELEMENT pop (#PCDATA)>
```

   (d) Does this data validate against the following DTD?

```
<!ELEMENT foo (pop,pop+)>
<!ELEMENT pop (#PCDATA)>
```

   (e) Does this data validate against the following DTD?

```
<!ELEMENT foo (pop,pop*)>
<!ELEMENT pop (#PCDATA)>
```

   (f) Does this data validate against the following DTD?

```
<!ELEMENT foo ((pop,pop*)|(pop,pop+))>
<!ELEMENT pop (#PCDATA)>
```

   (g) Does this data validate against the following DTD?

```
<!ELEMENT foo ((pop,pop*)|(pop,pop,pop*))>
<!ELEMENT pop (#PCDATA)>
```

2. Assume a file `doc.xml` whose data satisfies the DTD:

```
<!ELEMENT root (elm*)>
<!ELEMENT elm  (#PCDATA)>
```

   Consider the function XQuery `f`

```
f=
<result>
 FOR $x in "doc.xml"/root/elm RETURN <a>$x/text()</a>
 FOR $x in "doc.xml"/root/elm RETURN <b>$x/text()</b>
 FOR $x in "doc.xml"/root/elm RETURN <c>$x/text()</c>
</result>
```

(a) Describe that this function does.

(b) What DTD might we infer for the resulting XML? (Stick strictly to valid DTDs only.)

(c) How might you extend DTDs to give a more precise DTD?

(d) Consider the following possible output DTD for the query `f`

```
<!ELEMENT result (((a,a)*, (b,b)*, (c,c)*) |
                  ((a,a)*,a (b,b)*,b,(c,c)*,c))>
```

i. What sort of data does this DTD describe?

ii. Does `f` satisfy this DTD?

iii. Compare this DTD with your inferred DTD from earlier. (If you view DTDs as regular languages, is there any relationship between the two?)

iv. What might this imply?