Exam

Web Data Management
Master Parisien de Recherche en Informatique

March 1st, 2019

This is the final exam for the Web Data Management class, which will determine 50% of your grade for this class (the other 50% being given by the project). The exam consists of 3 independent exercises + 1 bonus question. **You must write your answer to both exercise 3 and the bonus question on a *separate sheet* of paper.** You can choose to answer the questions in English or in French, as you prefer.

Write your name clearly on the top right of every sheet used for your exam answers, and number every page.

You are provided with an XPath "cheat sheet" which gives you a summary about the syntax of XPath (but which you probably shouldn't take at face value; see the Bonus question at the end). You are additionally allowed **one A4 sheet** (i.e., two pages, one on each side) with the content of your choice. You may not use any other written material.

The exam is **strictly personal**: any communication or influence between students, or use of outside help, is prohibited. No electronic devices such as calculators, computers, or mobile phones, are permitted. Any violation of the rules may result in a grade of 0 and/or disciplinary action.

Exercise 1: Compiling DTDs to tree automata (11 points)

In the entire exercise, we will consider XML documents that contain no textual content or entities, and have no document type declaration, comments, or processing instructions. The documents only use tag names in a finite fixed alphabet Σ. Further, each tag is either empty or contains exactly two child tags. In other words, the DOM representation of an XML document is always a full binary tree with node labels in Σ, which we call a Σ-tree. For now, we also assume that there are no attribute nodes (we will revisit this assumption further on).

Remember that a *DTD* can be used to describe the schema of an XML document. We will study a simplified version of the DTD language called *SimpleDTD* that only allows two kinds of declarations:

- First, declarations of the form:

  ```
  <!ELEMENT eltname (E)>
  ```

  where `eltname` is a name in Σ, and where E is a non-empty *deterministic* regular expression (regexp) built with concatenation (`,`) disjunction (`|`), element names (from Σ), and parentheses. This declaration specifies that the sequence of children of the element must satisfy the regexp. Remember that a *deterministic* regexp is one where, for every position in the regexp, for every element name in Σ, the next position in the regexp is uniquely defined.

- Second, declarations of the form:

  ```
  <!ELEMENT eltname EMPTY>
  ```

  where `eltname` is a name in Σ. This declaration specifies that the element must be empty.

A *SimpleDTD* is a sequence of such declarations with exactly one declaration overall per element name.

Question 1 (1 point). Consider the alphabet $\Sigma_1 = \{a, b, c\}$ and the following constraints:

- elements labeled b and elements labeled c have no children
- elements labeled a have either two children labeled b or two children labeled c.

(a) Write a SimpleDTD Δ_1 that expresses these constraints.

Answer.

```
<!ELEMENT a (bb|cc)>
<!ELEMENT b EMPTY>
<!ELEMENT c EMPTY>
```

(b) Now, we also allow elements labeled a to have one first child labeled b and a second child labeled c, in addition to what Δ_1 allows. Write a SimpleDTD Δ_1' that expresses these constraints.

Answer.

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

Note that the intuitive solution that follows is wrong because the regexp is not deterministic:

```
<!ELEMENT a (bb|bc|cc)>
<!ELEMENT b EMPTY>
<!ELEMENT c EMPTY>
```

Our goal will be to translate DTD variants to tree automata. A *bottom-up deterministic finite tree automaton on Σ-trees* (or *Σ-TA*) is a tuple $A = (Q, F, \iota, \delta)$ where Q is a finite set of *states*, $F \subseteq Q$ is the set of *final states*, $\iota : \Sigma \to Q$ is the *initialization function*, and $\delta : Q \times Q \times \Sigma \to Q$ is the *transition function*. The *run* of A on a Σ-tree T is defined as the unique function ρ that maps each node of T to Q such that:

- for each leaf n of T with label $l \in \Sigma$, we have $\rho(n) := \iota(l)$;
- for each internal node n of T with label $l \in \Sigma$ and children n_1 and n_2, we have $\rho(n) := \delta(\rho(n_1), \rho(n_2), l)$.

We say that the run ρ is *accepting* if $\rho(r) \in F$ where r is the root of T, and we say that A *accepts* T if the run of A on T is accepting.

Question 2 (2 points). Give an algorithm which, given a SimpleDTD Δ on an alphabet Σ, constructs a Σ-TA A_Δ such that for every Σ-tree T, the automaton A_Δ accepts T if and only if T satisfies Δ. Argue for the correctness of your algorithm, and discuss its complexity as a function of Σ and Δ.

Answer. The state space of the automaton is $Q = \Sigma \sqcup \{\bot\}$. The final states of the automaton are $F := Q \setminus \{\bot\}$. The initial function ι is the function defined as follows: for every label $l \in \Sigma$:

- If Δ contains a declaration allowing l to be empty, then $\iota(l) := l$;

- Otherwise, $\iota(l) := \bot$.

The transition function δ is defined as follows: for any states $q_1, q_2 \in Q$, for any label $l \in \Sigma$:

- If $q_1 = \bot$ or $q_2 = \bot$, then $\delta(q_1, q_2, l) := \bot$;
- Otherwise, if l has a declaration such that the sequence $q_1 q_2$ (of two letters of Σ) matches the regexp, then $\delta(q_1, q_2, l) := l$;
- Otherwise, $\delta(q_1, q_2, l) := \bot$.

We now argue for correctness by showing the following inductive claim: for every Σ tree T, letting ρ be the unique run of A on T and r be the root of T, then:

- Either T does not satisfy the DTD, and $\rho(r) = \bot$;
- Or T satisfies the DTD, and $\rho(r)$ is the label of r.

Let us prove this invariant by induction:

- Initialization: if T consists of a single node n, then
  - If T does not satisfy the DTD, then it must be because n has a label l which is not allowed to be empty, and then we have $\iota(l) = \bot$ hence $\rho(n) = \bot$;
  - Otherwise, if T satisfies the DTD, then n must have a label l which can be empty, and then we indeed have $\iota(l) = l$ hence $\rho(n) = l$.
- Induction:
  - Either T does not satisfy the DTD, and then there are two cases:
    - Either one of the subtrees T_1 and T_2 does not satisfy the DTD. Assume that it is T_1; the other case is symmetric. Then, by induction, we know that $\rho_1(n_1) = \bot$ for n_1 the root of T_1 and ρ_1 the unique run of A on T_1. Then the definition of δ ensures that, whatever the label of the root r of T and whatever the run over T_2, we have $\rho(r) = \bot$.
    - Or T_1 and T_2 satisfy the DTD but the root r of T has children n_1 and n_2 whose labels l_1 and l_2 do not match the declaration for the label l of r in the DTD. In this case, we know by induction that $\rho(n_1) = l_1$ and $\rho(n_2) = l_2$. The definition of δ then ensures that we have $\rho(r) = \bot$.
  - Or T satisfies the DTD. This means that the two subtrees T_1 and T_2 of T must also satisfy the DTD, so that by induction we know that, letting n_1 and n_2 be their respective roots and l_1 and l_2 be their respective labels, we have $\rho(n_1) = l_1$ and $\rho(n_2) = l_2$. Now, letting r be the root of T, the word $l_1 l_2$ must satisfy the declaration for the label l of r. This means that, by definition of δ, we have $\rho(r) = l$.

We have thus established correctness by induction.

In terms of complexity, the bottleneck is the definition of δ. There, we can simply enumerate all labels $l \in \Sigma$, all pairs of child states $l_1, l_2 \in \Sigma$, and determine if $l_1 l_2$ is accepted by the declaration of l. As the regexp for the declaration of l is deterministic, this can be done in linear time in the expression. Hence, overall we are in $O(|\Sigma|^3 \times |\Delta|)$, where $|\Delta|$ is the longest size of a declaration in Δ. Nevertheless, this is polynomial in Δ.

Question 3 (1 point). For a fixed SimpleDTD Δ, what is the complexity, given an input Σ-tree T, to determine whether T satisfies Δ? Justify your answer.

Answer. We first apply the process of question 2 to compute the tree automaton A_Δ, the complexity of which is constant as it is independent from T. Now, given the tree T, we test whether A_Δ accepts T, which can be done in linear time in T. Hence, we can perform verification in linear time in T.

Question 4 (1 point). Consider the alphabet $\Sigma_4 = \{a, b, c, d\}$ and the following constraint: "for every a-labeled node n, there must be both a b-labeled node and a c-labeled node that are descendants of n".

Write an XPath query that returns precisely the a-labeled nodes that do *not* satisfy the constraint. (You may refer to the provided XPath cheat sheet.)

Answer. `//a[not(.//b) or not(.//c)]`

Question 5 (1 point). Design a Σ_4-TA that accepts exactly the trees that satisfy the constraint of question 4, or prove that no such tree automaton exists.

Answer. We can design such an automaton. The state space is $Q := \{\emptyset, \{b\}, \{c\}, \{b, c\}, \bot\}$. The final states are $F := Q \setminus \{\bot\}$. The initialization function maps a to $\bot$, b to $\{b\}$, c to $\{c\}$, and d to $\emptyset$. The transition function is defined as follows: for every $q_1, q_2 \in Q$, for every label $l \in \Sigma$:

- if one of q_1 or q_2 is $\bot$, then $\delta(q_1, q_2, l)$ is $\bot$;
- otherwise, if l is a and $q_1 \cup q_2$ is not $\{b, c\}$, then $\delta(q_1, q_2, l)$ is $\bot$.
- otherwise, if l is b or c, then $\delta(q_1, q_2, l)$ is $(q_1 \cup q_2 \cup \{l\}) \cap \{b, c\}$.

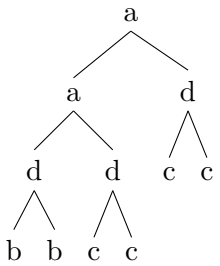
To show correctness we argue that, on every Γ-tree T, the state to which the root node is mapped by the unique run of the automaton on T is:

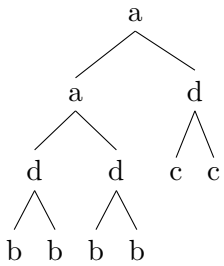
- $\bot$ if there is a violation
- otherwise, the subset of $\{b, c\}$ corresponding to the node labels that occur in T

This is shown by a straightforward induction.

Question 6 (1 point). Design a DTD that accepts exactly the trees that satisfy the constraint of question 4, or prove that no such DTD exists.

Answer. There is no such DTD. To see why, assume by contradiction that such a DTD Δ exists, and consider the following two trees:





The first tree satisfies the constraint. Hence, the declaration for b and c must be `<!ELEMENT b EMPTY>` and `<!ELEMENT c EMPTY>`, the declaration for d must allow at least the sequence of children bb and cc, and the declaration for a must allow at least the sequence of children dd and ad. Now, in the second tree, all elements are used according to these declarations, hence the second tree is accepted by the DTD. Yet, the leftmost a of the tree violates the constraint, so the tree should not be accepted, a contradiction.

We now add *attributes* to our XML language. The finite tree alphabet Σ will now be partitioned into three kind of labels: *element names* (as previously), *attribute names*, and the two special labels *attr* and *child* (to be explained).

We keep our assumptions on XML documents from the beginning of the exercise, except elements may now have attributes, whose name must be an attribute name in Σ, and whose value is *always the empty string*. We recall that well-formed XML documents cannot have two attributes with the same name.

We study another simplified version of the DTD language called *AttrDTD* that only allows two kinds of declarations (the `ELEMENT` declarations are no longer allowed, but we implicitly assume that there are no constraints on the children nodes allowed on a given element):

- First, declarations of the form

  ```
  <!ATTLIST eltname attname "" #IMPLIED>
  ```

  which specifies that element `eltname` (an element name in Σ) *may* have an attribute named `attname` (an attribute name in Σ),

- Second, declarations of the form

  ```
  <!ATTLIST eltname attname "" #REQUIRED>
  ```

  which specifies that element `eltname` (an element name in Σ) *must* have an attribute named `attname` (an attribute name in Σ).

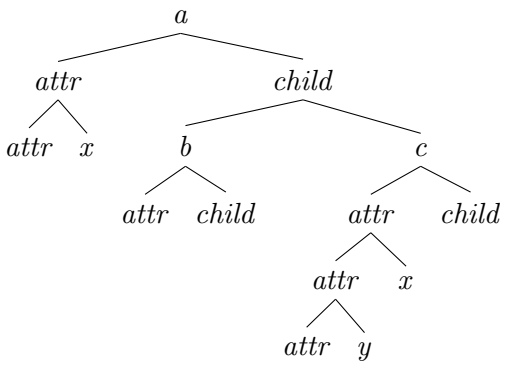
An *AttrDTD* is a sequence of such declarations where there is at most one declaration per pair of an element name and an attribute name. (However, each element name may have zero, one, or many declarations of either type.)

We now explain how XML documents with attributes are represented as Σ-trees. Each element node n of the XML document with label l (an element name in Σ) is coded by a node n' with label l in the tree, which has two children whose respective labels are the special labels *attr* and *child*, and whose children are defined as follows:

- The *attr* node is a leaf if n has no attributes. Otherwise if n has $k > 0$ attributes then the *attr* node is the root of a chain of $k+1$ *attr* nodes: for $1 \leq i \leq k$, the i-th *attr* node has as right child a node whose label is that of the i-th attribute (in the order in which they appear in the textual representation of the XML file) and has as left child the $(i+1)$-th *attr* node; note that the $(k+1)$-th *attr* node is a leaf.
- The children of the *child* node code the children of n (and if n has no children then the *child* node is a leaf).

Here is an example on the alphabet $\Sigma = \{a, b, c, x, y, attr, child\}$:

```
<a x="">
  <b></b>
  <c x="" y=""></c>
</a>
```



We now denote by Σ-*tree* the trees whose nodes carry a label in Σ and which have been constructed in this way.

Question 7 (2 points). Give an algorithm which, given an AttrDTD Δ on an alphabet Σ, constructs a Σ-TA A_Δ such that for every Σ-tree T, the automaton A_Δ accepts T if and only if T satisfies the DTD. Argue for the correctness of your algorithm, and discuss its complexity as a function of Σ and Δ.

Answer. Let Σ' be the set of labels that are attribute names. The automaton state is $Q := \{\top, \bot\} \cup 2^{\Sigma'}$, and the final states are $F := Q \setminus \{\bot\}$. The initialization function maps every label $l \in \Sigma'$ to $\{l\}$ and all other labels to $\top$. The transition function is defined as follows for every state q_1, q_2, and label l:

- if one of q_1 or q_2 is $\bot$ then $\delta(q_1, q_2, l)$ is $\bot$
- otherwise, if l is *attr* then $\delta(q_1, q_2, l) := q_1 \cup q_2$ (the invariant will imply that in this case q_1 and q_2 cannot be $\top$)
- otherwise, if l is an element label, then the invariant will imply that in this case q_1 is a set, and then:
  - if the set q_1 is a legal set of attributes for an element labeled l according to the DTD, then $\delta(q_1, q_2, l) := \top$
  - else, $\delta(q_1, q_2, l) := \bot$
- otherwise (if l is *child*) then $\delta(q_1, q_2, l) := \top$

We argue for correctness by first observing that, on trees T denoting attributes (i.e., a path of *attr* nodes with attribute values as children), the run of the automaton associates each *attr* node to the set of attribute values that appear in the tree: this is immediate by induction. Note that it ensures that the transitions defined above cover all possible cases on a Σ-tree. Now, we claim that on a Σ-tree every node is mapped to either $\top$ or $\bot$ depending on whether or not there is a violation. Indeed, trees consisting of a single element node satisfy the constraint, and in other trees the values $\top$ and $\bot$ are correctly propagated. This establishes correctness.

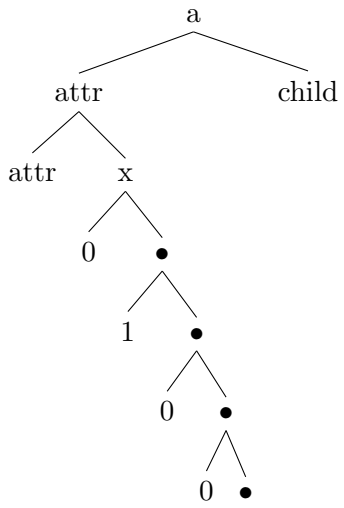
We can check in linear time in Δ whether an attribute set is legal, but the complexity is exponential in Σ as each subset needs to have its own state.

Question 8 (1 point). Assume now that we want to extend our tree representation to XML documents where attributes can carry values. We want to allow arbitrary values to appear in the XML document, however we still want the tree representation to be on a fixed alphabet (i.e., Σ should be finite, it should not contain infinitely many possible attribute values). We also want trees to remain binary and full. Present a possible way to extend our tree model to represent attribute values subject to these constraints.

Answer. As attribute values will be representable by a sequence of bits, we can add to our alphabet Σ the special values 0, 1, and $\bullet$ (to be used as a filler to make the trees binary). We then code the value of attributes as a chain of 0 and 1 values under the attribute. For instance, the following element, where we directly write the attribute value in binary:

```
<a x="0100"/>
```

would be coded as:



Question 9 (1 point). We extend the *AttrDTD* language to add attribute declarations of the form:

```
<!ATTLIST eltname attname ID #REQUIRED>
```

which defines a mandatory attribute `attname` for element `eltname` which must be an XML ID (i.e., it must consist of only lowercase and uppercase ASCII letters, digits, '`-`', '`.`', '`:`', or '`_`', and must start by a letter, by '`:`', or by '`_`') and must be unique (i.e., the value used for an occurrence of an attribute declared with `ID` cannot be used as the value of another occurrence of an attribute declared with `ID`).

Explain how to extend the algorithm of question 7 to produce automata (with your extended encoding in question 8) that can handle AttrDTD featuring such declarations, or explain why it is not possible to do so.

Answer. It is not possible to do this with a finite automaton. In our coding of question 8, the problem is that, as the automaton has finitely many states, it can only remember a finite quantity of information while reading an ID attribute value. Hence, if the automaton fails when it sees the same ID attribute value twice, it must also fail erroneously in cases where all values are unique, because there must be two values that will yield the same state.

A similar phenomenon should occur no matter how attribute values are represented.

Exercise 2: Translating XPath to SPARQL (4 points)

The goal of this open-ended exercise is to convert XPath queries on trees to SPARQL queries on RDF graphs. For brevity, all URIs will be in an `m` namespace; you do not need to worry about how it is declared. You may refer to the provided cheat sheet for information about the syntax of XPath. For SPARQL, recall the following important features of the language:

- Basic queries: `SELECT ?x WHERE { ... }`
- The body contains statements of the form `subject predicate object .` where the subject, predicate and object may be URIs (e.g., `<m:42>`) or variables (e.g., `?foo`). The object may also be a literal (e.g., `"42"`).
- The predicate can also be a *property path*, i.e., a regular expression on paths, with the most common operators being disjunction ('`|`'), concatenation ('`/`'), Kleene star ('`*`'), one or more occurrences ('`+`'), and the inverse relation (e.g., '`^<m:foo>`' is the inverse relation of `<m:foo>`).
- The `{ ... } UNION { ... }` operator can be used to union the results of two queries.
- The `FILTER EXISTS { ... }` operator can be used to filter the preceding results based on the existence of matches to a subquery. There is also `FILTER NOT EXISTS`.
- We can add `OFFSET x LIMIT y` at the end of a `SELECT` query to return only y results starting at the x-th result.
- There is support for aggregates, e.g., `SELECT ?x (COUNT(?y) AS ?count) WHERE { ... }`.

We work with XML documents having no document type declaration, no comments, no processing instructions, no entities, no attributes, and no textual content. To convert an XML tree to an RDF graph, we generate one numerical identifier per element in the XML document order (starting at 0), and put them in the `m` namespace. We use three special relation names:

- `m:l`, where the subjects are the nodes of the tree, and the object for each subject is a literal containing the node label;
- `m:c`, where the subjects are the internal nodes of the XML tree, and the objects for a given subject are the children of that node;

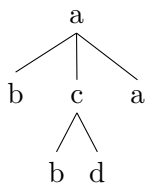
- `m:n`, where the subjects are the nodes of the XML tree that are not the last child of their parent, and the object for a given subject is the next sibling of that node.

Here is an example, where for brevity we write the list of RDF facts in three columns:

XML document

```
<a>
  <b></b>
  <c>
    <b></b>
    <d></d>
  </c>
  <a></a>
</a>
```

XML tree



RDF representation (NTriples format)

```
<m:0> <m:l> "a" .     <m:0> <m:c> <m:1> .     <m:1> <m:n> <m:2> .
<m:1> <m:l> "b" .     <m:0> <m:c> <m:2> .     <m:2> <m:n> <m:5> .
<m:2> <m:l> "c" .     <m:0> <m:c> <m:5> .     <m:3> <m:n> <m:4> .
<m:3> <m:l> "b" .     <m:2> <m:c> <m:3> .
<m:4> <m:l> "d" .     <m:2> <m:c> <m:4> .
<m:5> <m:l> "a" .
```

Question 1 (1 point). Translate the XPath query

`a//b[../c]`

to an equivalent SPARQL query of the form `SELECT ?x WHERE { ... }`, i.e., the RDF nodes `?x` returned by the query should be exactly the RDF nodes corresponding to the tree nodes returned by the XPath query.

Answer.

```
SELECT ?x WHERE {
  ?a <m:l> "a" .
  ?a <m:c>* ?x .
  ?x <m:l> "b" .
  FILTER EXISTS {
    ?c <m:c> ?x .
    ?c <m:l> "c" .
  }
}
```

Question 2 (3 points). Explain more generally how to translate XPath queries to SPARQL. In particular, you should mention the following features (the list is not exhaustive):

- Finding the root of the document
- Axes such as `descendant`, `following-sibling`, or `following`
- Predicates, nested predicates
- `position()`, `last()`, `count()`

Answer.

- The root of the document will always be `<m:0>` by definition. Otherwise, it can be identified as the node that has no parents:

```
SELECT ?x WHERE {
  ?x <m:l> ?l .
  FILTER NOT EXISTS {
    ?y <m:c> ?x .
  }
}
```

- To find the descendants of `<m:0>`:

```
SELECT ?x WHERE {
  <m:0> <m:c>* ?x .
}
```

- To find the following siblings of `<m:1>`:

```
SELECT ?x WHERE {
  <m:1> <m:n>* ?x .
}
```

- To find the next nodes in document order, it's a bit more subtle: those are the descendants of a strict next sibling of a parent (requiring the reverse relation of `m:c`)

```
SELECT ?x ?y WHERE {
  ?x <m:l> ?xx .
  ?y <m:l> ?yy .
  ?x ^<m:c>*/<m:n>*/<m:c>* ?y .
}
```

- Predicates are implemented with `FILTER EXISTS` (which can be nested in the case where there are nested predicates, and concatenated if we want to impose multiple predicates)
- Position, last, count, etc., can be implemented (in simple cases) using `LIMIT` and `OFFSET`
- Successive steps can be handled successively in the SPARQL query with consistent naming. For instance, doing the example from question 1 again:

```
SELECT ?s1 WHERE {
  # == first step: produce ?s0 ==

  # node test
  ?s0 <m:l> "a" .

  # == second step: produce ?s1 ==

  # axis: descendant
  ?s0 <m:c>* ?s1 .

  # node test
  ?s1 <m:l> "b" .

  # predicate
  FILTER EXISTS {
    # axis:parent
    ?s1 ^<m:c> ?s2 .
    # node test
    ?s2 <m:l> "c" .
  }

  # other predicates on ?s1, if any, would go here

  # subsequent steps, if any, would go here
}
```

Please write your answer to Exercise 3 and the bonus question on a separate sheet of paper.

Exercise 3: The Deep Web (5 points)

This exercise is fairly open ended. No fixed answer is expected, but you will be judged on the originality, feasibility, and soundness of your proposals.

Sources from the deep Web (Web forms, Web services) typically expose some relational data, as is traditional in the data integration setting, with a twist: it is not possible to access the data without providing some input to the form or service. For example, a directory service such as *PagesBlanches* may expose relational data of the form `Contact`(`Last`,`First`,`City`,`Phone`) but only if a form is submitted with input consisting of both a `Last` name and a `City`. This is called an *access method* and it is denoted:

$$\texttt{Contact}^{\text{ioio}}(\texttt{Last}, \texttt{First}, \texttt{City}, \texttt{Phone})$$

where 'i' stands for *input* and 'o' for *output*: if both a `Last` name and a `City` are provided as *input*, a list of matching records will provided, including the value of the `First` name and `Phone` number for each record.

There are two common approaches to exploiting deep Web data:

- the *surfacing* approach (also called *extensional*, *siphoning*) where one aims to retrieve and store *all* content hidden behind the deep Web interface, in the spirit of data warehousing;
- the *intensional* approach where one aims to rewrite user queries at runtime to make use of the deep Web query interfaces, in the spirit of the mediator approach to data integration.

Question 1 (2 points). Assume we want to follow the surfacing approach. Propose a pragmatic strategy to retrieve as much content as possible from a directory deep Web source featuring the access method above. You need to describe in detail the architecture of a system that siphons the deep Web source: what content is submitted to the Web form? where to find input data? when to stop? how to assess how much data is left?

We now focus on the intensional approach. Assume we have a bunch of relational sources, each with one or several access methods that constrain access to the data. We also assume we have a Local-As-View description of each source as a conjunctive query over a global, mediator schema. Now consider a user query Q.

Question 2 (1 point). Give an example with at least two different sources where Q could be rewritten using at least two local sources with standard data integration techniques if all access methods had no input attribute, but where such a rewriting is not possible when access methods have input attributes.

Question 3 (2 points). Propose an approach to decide whether there exists a rewriting of Q in terms of local sources, that respects the access methods. Can standard data integration techniques, such as the Bucket or Inverse Rules algorithm, be adapted to this setting?

Bonus Question (+2 points)

The XPath cheat sheet provided was found on the Internet, and as most things found on the Internet, it contains (at least) one major error. This is an error with respect to the semantics of XPath. You get 2 bonus points if you identify this error and provide an example XML document and query that illustrate why this is an error.