Tree Pattern Evaluation using SAX

Friso Abcouwer - 4019873 Matthijs van Dorth - 1265911

June 1, 2014

1 Introduction

In this report we will show how we implemented an algorithm for evaluating tree-pattern queries using SAX. SAX (Simple API for XML) is a way to parse XML documents using a stream of data, in contrast to DOM (Document Object Model), which is a complete representation of the XML file as a tree.

Enclosed with this report are an executable jar file, as well as the source code of the jar. The parser we created takes as input a query and an XML file and can display the result of the query in various formats. Alternatively, users can also define their own query trees in Java. An example query is given in Listing 1, and an example XML file in Listing 2.

Given an XML file such as the one given in the book and in Listing 2, the query will return all email and last name nodes of the person nodes that have an email and last name node. The result is a table or an XML file with these values.

2 The Structure of the Algorithm

The algorithm that we created has several parts. First there is the PatternNode class that represents a single node in a XML document, such as the node with a name "email". Each node can have a value such as "a@work". When executing a query a TPEStack is created. A TPEStack consists of a PatternNode and a list of child TPEStacks. The TPEStack for Listing 1 can be constructed with the Java code shown in Listing 3.

The TPEStack is used by the StackEval algorithm to create a Match object when an element-name matches the name of the PatternNode. The most important methods of this StackEval algorithm are `startElement`; which is called everytime a opening tag is encountered, `endElement`; which is called everytime a closing tag is found and `characters`; which is called when text nodes are found.

These Match objects keep a record of the TPEStack that was matched against this object and all the child Match object children that still have to be matched against a PatternNode.

As tree nodes are processed, they are saved into Result objects. This object has an `id`, `parentId`, `name`, `value` and `depth` at which the element was found, and can be sorted by ID. When the XML file has been parsed entirely, the desired Results are saved into a ResultList object that is able to print out the results as a table or as XML. As such, the StackEval class corresponds to the first assignment from the book (evaluating C-TP tree patterns), and together with the ResultList class corresponds to the second assignment (computing result tuples in the form of tables or XML).

3 Wildcards, Optional Nodes and Value Predicates

We proceeded with extending the algorithm further by allowing wildcards into the queries. We gave the PatternNode boolean fields `wildcard` and `optional`, and apapted the `startElement` and `endElement` methods to take into account situations when a wildcard or optional node was found. The way the results had to be printed had to be altered, however, since we needed to retrieve the name of the node in case of printing it as XML or print it with a wildcard symbol *, when printing it in a table, and `null` values had to be used with optional nodes where applicable. Matching on value predicates turned out to be relatively simple to implement: if the tree pattern Stack had a node with the required value, we could check the nodes in the input to see if they matched, and if not, remove the corresponding Matches from the list.

4 The Final Algorithm

A somewhat simplified version of the algorithm's code is presented in Listings 4 through 8.

After the InputHandler parses the query into a tree pattern, the StackEval class is initialized as shown in Listing 4. The helper method verifyTopMatch() will be used later: it verifies that a TPEStack has matches and that its top match has an 'open' state.

The `startElement` method is shown in Listing 5. Though it is similar to the pseudocode given in the book, we had to switch to using *rawName* rather than *localName*. Since spar is not always defined, and its top Match might not always be defined either, we had to add checks for this as well.

The `characters` method is shown in Listing 6. For each element, a Result object is created. In the case that the node has a text value, this is put into the result object as well as into nodeStrings, which maps pre numbers to values (and was also used in startElement). The appending in case of multiple calls is also important: we found that without it, it was not possible to correctly handle Strings with newlines or tabs.

Listing 7 shows the `endElement` method. Compared to the one in the book, it adds housekeeping of Result attributes, as well as the handling of optional nodes, value predicates and matches and stacks that might not be defined.

Finally, the results are collected, sorted and presented, as shown in Listing 8.

5 Input & Output

5.1 Printing the result as XML

The result of the query can be printed out as valid XML. This is done in the ResultList class: the elements' opening tags are printed in order and the closing tags are placed on a stack for later printing. When the next element to be examined has a lower depth than the current one, the closing tags are popped from the stack and printed. The depth of each result is used to properly indent the text.

5.2 Creating a TPEStack from a Query

The InputParser is an object that is created from a query String and is able to create a TPEStack corresponding to the query. Our parser does not support the Let clause, but the other clauses, For, In, Where, and Return, are supported. The InputHandler class of our code accepts a query String and separates it into these four clauses. Going through them in sequence, PatternNodes and TPEStacks are created to construct a tree pattern based on the input query. Finally, the root node of the generated tree pattern is returned.

6 Testing the Parser

Using several different XML files, we tested the parser with different queries. For example, the query shown in Listing 1 was tested against the XML file shown in Listing 2. The result can be shown using a table as was done in the book [1] or as XML, shown in Listing 9.

As can be seen, only the values of email and last are printed out. Because the person with the last name "Lang" does not have an email adress, only the last name is printed out. In the table this would have resulted in a `null` value.

To test our algorithm, we created two classes: `TestParse.java`, in which tree patterns are constructed by hand to test the algorithm, and `TestQueries.java`, which tests the five queries presented in the book, along with a few other queries, and outputs the results in table and XML form. Most of the queries we tested the algorithm with performed correctly, with one exception: query *q5* from the book. We believe this is due to the wildcard node between two named nodes, *person/*/last*. We were unfortunately unable to correct this error in time for the deadline.

7 Future Improvements

Though our parser is functional, we believe there is still room for improvement. Some examples of this are the following:

- **Wildcards**: As stated above, queries with wildcards currently do not work if they are embedded between named nodes, for example as *person/*/last*.
- **Documentation**: Though we believe our code to be readable without too many problems, the StackEval algorithm is quite extensive: more comments and other documentation would certainly help make it more accessible.
- **Memory Use**: In developing the algorithm, we have not paid much attention to the memory use of the algorithm. For example, printing results is done with a recursive method, which could very well cause problems as input size increases.
- **Input Parsing**: input queries are currently not checked for correctness. Checking the queries for correctness could help detect and correct common typos or mistakes.
- **Error Handling**: There is only limited error handling in this implementation, which reduces user-friendliness.

- **XML Formatting**: XML indentation is now done based on the depth of elements in the original tree. This can lead to elements being indented several tabs ahead of the element above them in the result.

References

[1] Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, et al. *Web data management*. Cambridge University Press, 2012.

Listings

Listing 1: Example Query

```
1 for $p in //person  [email]
2                     [name/last]
3 return (    $p//email,
4             $p/name/last)
```

Listing 2: people.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <person>
    <email>m@home</email>
    <name>
      <first>Mary</first>
      <last>Jones</last>
    </name>
  </person>
  <person>
    <name>
      <first>Bob</first>
      <last>Lang</last>
    </name>
  </person>
  <person>
    <email>@home</email>
    <email>@work</email>
    <name>
      <first>Al</first>
      <last>Hart</last>
    </name>
  </person>
</people>
```

Listing 3: Construction of a TPEStack for Query 1

```
public static TPEStack personStack() {
    PatternNode person = new PatternNode("person");
    PatternNode email = new PatternNode("email");
    PatternNode name = new PatternNode("name");
    PatternNode last = new PatternNode("last");

    TPEStack personStack = new TPEStack(person, null);
    TPEStack nameStack = new TPEStack(name, personStack);
    TPEStack lastStack = new TPEStack(last, personStack);
    TPEStack emailStack = new TPEStack(email, personStack);

    personStack.addChildStack(nameStack);
    personStack.addChildStack(emailStack);
    personStack.addChildStack(nameStack);
    nameStack.addChildStack(lastStack);

    person.addChild(name);
    person.addChild(email);
    name.addChild(last);

    return personStack;
  }
```

Listing 4: "Initialization"

```
1  Main(String query, String fileName){
2    rootNode <- new InputHandler(query).parseQuery();
3    s <- new StackEval(rootNode);
4    XMLReader.parse(fileName) using s as ContentHandler;
5  }
6  StackEval{
7    int currentPre = 1;
8    Stack<Integer> preOfOpenNodes = new Stack<Integer>();
9    Map<Integer, String> nodeStrings = new HashMap<Integer, String>();
10   Map<Integer, Match> resultsMap = new HashMap<Integer, Match>();
11
12   StackEval(PatternNode root){
13     this.rootStack <- new TPEStack(root);
14     Initialize TPEStacks of descendants of root;
15   }
16
17   startDocument(){
18     //EMPTY: initializing the necessary stacks is performed in the constructor
19   }
20   ...
21 }
22
23 TPEStack{
24   ...
25   public boolean verifyTopMatch() {
26     if (!matches.isEmpty())
27       return (matches.peek().getState() == 1);
28     else
29       return false;
30   }
31   ...
32 }
```

Listing 5: ” startElement”

```
1
2 startElement(String nameSpaceURI, String localName, String rawName, Attributes
      attributes){
3   for (TPEStack s : rootStack.getDescendantStacks()) {
4     PatternNode p = s.getPatternNode();
5     TPEStack spar = s.getSpar();
6     if (rawName.equals(p.getName()) || p.isWildcard()) {
7       if (spar == null) {
8         Match m = new Match(currentPre, null, s);
9         resultsMap.put(currentPre, m);
10        s.push(m);
11      } else if (spar.top() != null && spar.top().getState() == 1) {
12        Match m = new Match(currentPre, spar.top(), s);
13        spar.top().addChild(s.getPatternNode(), m);
14        // create a match satisfying the ancestor conditions of
15        // query node s.p
16        resultsMap.put(currentPre, m);
17        s.push(m);
18      }
19    }
20  }
21  // Attributes part
22  for (int i = 0; i < attributes.getLength(); i++) {
23    // similarly look for query nodes possibly matched
24    // by the attributes of the currently started element
25    for (TPEStack s : rootStack.getDescendantStacks()) {
26      PatternNode p = s.getPatternNode();
27      TPEStack spar = s.getSpar();
28      if (attributes.getLocalName(i).equals(p.getName())
29          && (spar == null || spar.verifyTopMatch())) {
30        Match ma;
31        if (spar == null) {
32          ma = new Match(currentPre, null, s);
33        } else {
34          ma = new Match(currentPre, spar.top(), s);
35          spar.top().addChild(p, ma);
36        }
37        nodeStrings.put(currentPre, attributes.getValue(i));
38        ma.close();
39        s.push(ma);
40      }
41    }
42  }
43  preOfOpenNodes.push(currentPre);
44  currentPre++;
45 }
```

Listing 6: "characters"

```
1 public void characters(char[] ch, int start, int length) throws SAXException {
2   String str = new String(ch, start, length).trim();
3   int last = preOfOpenNodes.lastElement();
4
5   Result r1 = results.getResult(last);
6   if (r1 != null) {
7     r1.setValue(r1.getValue() + str); // Append in case of multiple
8                       // calls to characters
9   } else {
10    r1 = new Result(last, -1, null, str, preOfOpenNodes.size());
11    results.add(r1);
12  }
13
14  if (str.length() > 0) {
15    if (nodeStrings.containsKey(last))
16      nodeStrings.put(last, nodeStrings.get(last) + " " + str);
17    else
18      nodeStrings.put(last, str);
19  }
20 }
```

Listing 7: "endElement"

```
endElement(String nameSpaceURI, String localName, String rawName) {
// we need to find out if the element ending now corresponded
  // to matches in some stacks
  // first, get the pre number of the element that ends now:
  int preOflastOpen = preOfOpenNodes.pop();

  // set this element's parent
  if (!preOfOpenNodes.isEmpty()) {
    results.getResult(preOflastOpen).setParentId(preOfOpenNodes.peek());
    results.getResult(preOflastOpen).setName(rawName);
  }

  // now look for Match objects having this pre number:
  for (TPEStack s : rootStack.getDescendantStacks()) {
    PatternNode p = s.getPatternNode();
    // Only check last 2 if s.top() is not null
    if ((p.getName().equals(rawName) || p.isWildcard())
        && s.verifyTopMatch() && s.top().getPre() == preOflastOpen) {
      // all descendants of this Match have been traversed by now.
      Match m = s.top();
      Result r1 = results.getResult(m.getPre());
      if (r1 != null) {
        r1.setName(rawName);
        if (p.isQueried()) {
          r1.setQueried(true);
        }
      }
      m.close();

      // Check for the value of the node
      String value = m.getSt().getPatternNode().getValue();
      if (!value.isEmpty()
          && !value.equals(nodeStrings.get(m.getPre()))) {
        resultsMap.remove(m.getPre());
        remove(m, s);

        if (r1 != null) {
          r1.setQueried(false);
        }
        if (m.getParent() != null) {
          m.getParent().removeChild(s.getPatternNode(), m);
        }
      }

      // check if m has child matches for all children of its pattern
      // node
      for (PatternNode pChild : p.getChildren()) {
        // pChild is a child of the query node for which m was
        // created
        if (!pChild.isOptional()
            && (m.getChildren().get(pChild) == null || m
                .getChildren().get(pChild).size() == 0)) {
          // m lacks a child Match for the pattern node pChild
          // we remove m from its Stack, detach it from its parent
          remove(m, s);
          if (r1 != null) {
            r1.setQueried(false);
          }
          resultsMap.remove(m.getPre());
          if (m.getParent() != null) {
            m.getParent().removeChild(s.getPatternNode(), m);
          }
        }
```

Listing 8: "Final Part"

```
endDocument(){

  finalResults <- new ResultList();

    for(Integer i : resultsMap.keySet()){
      if(results.get(i) exists and is Queried){
        add results.get(i) to finalResults;
      }
    }

  finalResults.sortByID();

  print result of the query in the form of a table and/or XML file;

}
```

Listing 9: The result of running Query 1 on people.XML

```
<?xml version="1.0" encoding="UTF-8"?>
<results>
    <person>
      <email>m@home</email>
      <name>
        <last>Jones</last>
      </name>
      <name>
        <last>Lang</last>
      </name>
    </person>
    <person>
      <email>@home</email>
      <email>@work</email>
      <name>
        <last>Hart</last>
      </name>
    </person>
</results>
```