String processing and XML

Menyhárt, László

menyhart@elte.hu
ELTE IK

Abstract. In my opinion the real problems are very important in teaching. In my new idea I would like to use XML to teach string processing. In this article I present my approach and the worked out curriculum. I tested it in two groups on ELTE, and I did an assessment. It is also analyzed here.

1. Introduction

The subject of my PhD is XML. I am teaching in the Eötvös Loránd Science University (ELTE) where I had some courses about string processing. I could see that the base algorithms are not too interesting for the students. That is why I tried to create some new practises. I chose the topic from my favourite XML. The SAX parsing is a simple sequential string processing which can be good to make presentation the importance of this course. This real problem is solved by a lot of implementation and we can create a simplistic application, too.

I worked out the curriculum which is presented here, and I had chance to test it in two groups where the students filled out a survey.

This article and its appendixes are available on [1] URL.

2. Curriculum

I worked out a new matter to a lesson with practises based on my idea. Now I present the lesson plan, and the handout which is good for the teacher and the students too.

Lesson plan

- Introduction (Booting and filling the first part of the survey) 6 minutes
- XML review 8 minutes
    - HTML. What is difference in XML? What does it do?
    - String file:readable, processing able, …
    - Well-formed
        - Now we do another restriction (easier to handle): “Very well”-formed
- Processing XML 8 minutes
    - DOM
        - tree data-structure

    - SAX (more details)
        - startDocument, endElement, startElement, endElement, findCharacters
        - others
- Reviewing about SAX_template 8 minutes
    - Downloading and unzipping
    - Running SAX_example
    - Reviewing surface, types, variables and functions of SAX_template

Now it is the tierce of the lesson.

- 1. practise 11 minutes
    - Implementation of SAX parsing
        - Designing of "event-functions": How can we create a table?
        - Coding into the SAX_template
- 2. practise 9 minutes
    - Implementation of getAttributeValue
        - Theorem of Linear searching
        - Coding into the SAX_template
- 3. a) practise 10 minutes
    - Implementation of parseXML
        - Talking about algorithms

Now it is the second tierce of the lesson.

- 4. a) practises (6) 12 minutes
    - Implementation of co-procedures
        - Talking about algorithms
- 4. b) practises (5) 10 minutes
    - Implementation of co-procedures
        - Coding into the SAX_template
- 3. b) practise 6 minutes
    - Implementation of parseXML
        - Coding into the SAX_template
- Fill the second part of the survey 2 minutes
- Homework
    - 0. Please, end the coding
    - 1. Rewrite SAX_handler to generate list instead of table.
    - 2. Implement readCDATASection
    - 3. How can we generalize the parseXML procedure? (not very-well formed!)

Handout

Today we introduce to XML, the SAX processing and we create an own implementation in a template application.

XML

XML is an abbreviation. It comes from eXtensible Markup Language. It stores the data in a text file. It contains tags, attributes and the data. Its structure is hierarchical. It is built from 1996 based on SGML. The first recommendation was published in 1998. The version 1.0 is used. ([2])

Goals at creations

- Straightforwardly usable over the Internet
- Support by wide variety of applications
- Easy managing
- Human-legible and reasonably clear
- Easy to create
- Free to use

Syntax

Example

```
<?xml version="1.0"?>
<!-- This is a comment -->
<example>
  <teacher executive="true" filled="false"/>
  <teacher executive="true">
    <etrid>ZSLABCD</etrid>
    <name>Zsako, Laszlo</name>
    <email>zsako@ludens.elte.hu</email>
  </teacher>
  <teacher>
    <etrid>MELEAET</etrid>
    <name>Menyhart, Laszlo</name>
    <email>menyhart@inf.elte.hu</email>
  </teacher>
</example>
```

The first line contains the declaration of XML which is contains the used version and maybe the character encoding, but it is missing now.
The second line is a comment.
The third line is the root, the tag “<example>" can be found.
It is followed by an empty element “<teacher>” where there are two attributes too.
In the next line an opening tag can be seen with an attribute.
The next three lines nodes stand with string content (“etrid”, “name”, “email”).
There is the closing tag for “teacher” in the ninth line.
A new complete “teacher” node goes after it.
Finally the document is closed by the root’s closing tag.

Opening tags

The opening tag stands the name of element and the attributes if there are some.

Closing tags

Against of HTML closing tags are required here. There cannot be opening tags without it.

The name of closing tags must correspond with the name of opening tags in view of characters. Case-sensitivity is important.

The descent is very important, too. The order of closing tags are not important in HTML but the build of XML is based on descending. So the second line is the correct.

```
<b><i>Text in bold and italic style. WRONG!</b></i>
<b><i>Text in bold and italic style. CORRECT!</i></b>
```

One root

Every XML document contains one (and only one) root element. It can have child elements which can have descendant, too.

```
<root>
   <child>
      <descendant>...</descendant>
   </child>
</root>
```

Attributes

Attributes consist of name-value pairs. Between the attributes must be a space. Values are in brackets. So only the second example is correct.

```
<element id=0> - WRONG
<element id="0"> - CORRECT
```

Comments

Syntax of comment is the same as in HTML.

```
<!-- This is a comment. -->
```

Well-formed document

An XML document is well-formed if it corresponds to these rules.

Valid document

An XML document is valid if it is well-formed and its structure is defined and observed. We can define the rules of the structure by two possibilities.

DTD

Document Type Definition. The structure of XML document is defined by DTD. It is a special definition, which can be contained in the XML as an inline definition in a <!DOCTYPE …> node declaration or XML can refer to an outer file. It contains only the names of elements and attributes; location, count and order of occurrence.

XSD

XML Schema Definition (XSD) is more than DTD. It can define the structure of XML document and the types of data by XML format. It specifies the elements, attributes, descent of XML document and it can define new types, enumerations, etc. It can define more information than DTD. It supports types and namespaces. So it is usable for testing syntax and semantics, too.

Special characters

There are characters which have special feature for helping the parsing. For example “<” character means the beginning of the tags. That is why “<” is required in the data instead of it. There are more characters that must be replaced by fore-defined ENTITY strings. The ENTITY starts with the “&” character and ends with the “;”.

Entity	Replaced character	Description
<	<	left angle bracket (less then)
>	>	right angle bracket (greater than)
&	&	ampersand
'	’	apostrophe
"	”	quotation mark
ő	ő	usable is the character codes, too

Table 1: Entities

CDATA section

Sometimes data must contain special characters or we do not want to replace the characters. for example storing the name of “K&H Bank”. We are able to indicate these sections, which is unnecessary to parse. There are not other elements. We can create a CDATA section (character data) where any data can appear except one string, the end-marker three characters.

```
<![CDATA[
```

Previous line contains the start characters. Any not parsed string can be coming here for example “<” and “&” characters. The end characters are these:

```
]]>
```

Tag and element or node

The other denomination of element is (in a tree). It is coming from the DOM (see later).

An element is build by an opening tag, a closing tag and the content. The opening tag consist of a “<” (left angle bracket) character, the name of element, the possible list of attributes (in `name=”value”` format) and “>” (right angle bracket) character. The closing tag consist of a “<” (left angle bracket) and a “/” (per) characters, the name of element and “>” (right angle bracket) character. The CONTENT can be text or other descent elements. Sometimes the text and other elements come together. It is possible in mixed element, but it is not liked, because the parsing will be more difficult.

```
<tagname attribute=”value”>CONTENT</tagname>
```

If the element does not have content (`<tagname></tagname>`) it can be written briefly. The opening tag contains a "`/`" (per) character just before the ">" (right angle bracket) character (`<tagname/>`). Of course attributes can appear here, too.

"Very well"-formed

Now we define a new, the "very-well"-formed idiom. If the documents are very-well formed the parsing will be easier. We would not like to regard the needless spaces, tabulators, and so on. The document must be well-formed and must have some other property:

1. Let there be just one space between the attributes
2. There are not mixed elements.
3. In the opening tag of an empty element the "/" (per) character must follow immediately the last attribute.
4. In the closing tag "/" (per) character follows immediately the "<" (left angle bracket) character.

XML processing

DOM

Document Object Model considers the XML document as a tree structure. ([3]) The whole document is loaded into the memory and it is stored in a tree data-structure. Reaching a node more times and programming is faster because the tree is in the memory. But it requires a lot of memory.

SAX

Simple API for XML looks the XML document as a sequential text file. ([4]) So if you reach an element and you need a previous one, you must read the file again from the beginning. In this case the work is slower. It does not require much memory because it reads only an element in one time. Almost every programming language has libraries for implementing it. Today we implement it! The solution is creating a general parsing processor which calls back the regular own functions to manage the data. It can be achieved in more format: over interfaces, with object oriented abstract class, events, ... Now we use the easiest one with overwriting the functions in an include file.

It defines the following events, basic methods:

Event	When	Parameters
startDocument	It runs on the beginning of the document	none
endDocument	It runs on the end of the document	none
startElement	It runs at the reading of opening tag	name of element , list of attribute, [namespace]
endElement	It runs at the reading of closing tag	element neve, [namespace]
characters	It runs at reading of character data	Text (read characters)

Table 2: Basic functions of SAX

There are some other, but we do not deal with them:

- processingInstruction
- ignorableWhitespaces
- skippedEntity

Moreover we can create or overwrite own event functions: for example to handle comments.

SAX_template

Download

http://xml.inf.elte.hu/2009_10_2/szovegfeldolgozas_xml/SAX_template.zip

http://xml.inf.elte.hu/

Education / 2009/2010 Spring semester / String processing – XML (Oktatás / 2009/2010 Tavaszi félév / Szövegfeldolgozás – XML)

SAX_template.zip

Files

Name of file	Description
unit1.pas	Mainprogram to the surface. Do not modify!
saxunit.pas	It contains the general parsing procedure and the co-functions.
SAX_handler.inc	It contains the event-functions (see 5 above)

Table 3: Important files

Surface

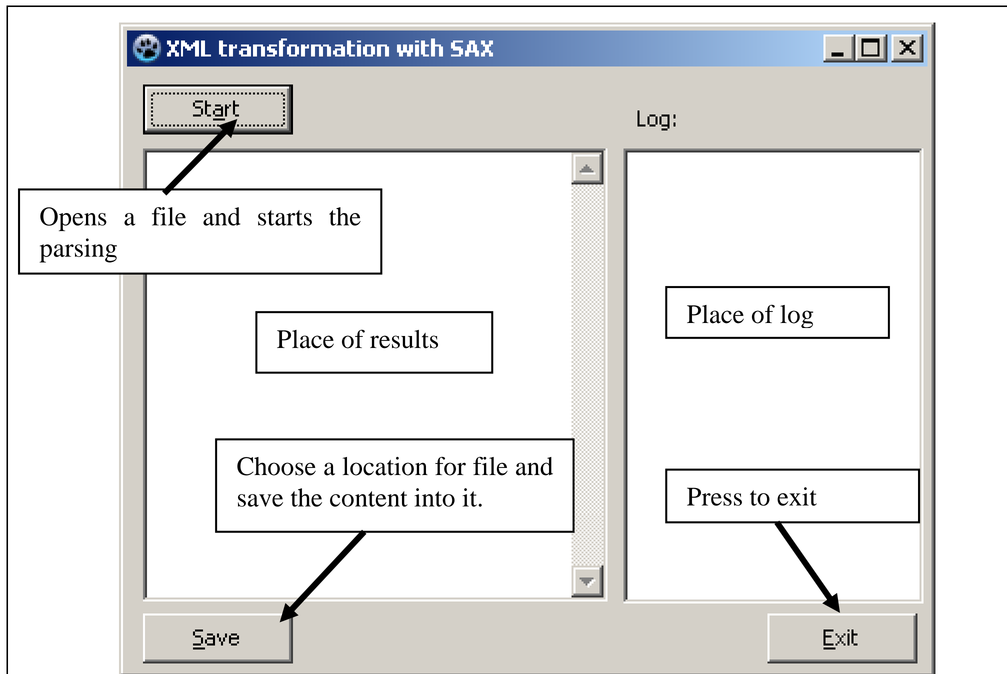

Figure 1: Surface of the template application

Types

We defined a new type for storing the name and value of attribute.

```
TAttribute=Record
  name:String
  value: String
end
```

An element can have more attributes, so we store them in an array.

```
TAttributeArray=Array(1..100:TAttribute)
```

We manage the count to use it as a list (almost).

```
TAttributeList= Record
  count:integer;
  item:TAttributeArray;
end
```

Variables

The main variables in the SAX class are the generated text, the log and the text file.

```
parsedStr,log:String;
f_xml:TextFile;
```

Co-variables

We need some other variables to read forward a character or to collect the characters.

```
ch:Character;
s1,s:String;
```

We manage the name and value of attribute detached and in a list, too.

```
ename,value: String;
att_list:TAttributeList;
```

We must find the position of "=" (equality) character.

```
p: integer;
```

Loop variable.

```
i: integer;
```

Notify the end of the tag or node.

```
endoftag,endofnode:Boolean;
```

Procedures and functions

We need getter/setter functions to reach the main variables.

```
setString(fs: String);
setLog(fs: String);
getString: String;
getLog: String;
```

We want to append string to the main variables.

```
addString(fs: String);
addLog(fs: String);
```

The next procedure makes the general XML processing.

```
parseXML(fname: String);
```

The previous procedure needs the following co-functions.

```
readForwardACharacter: String;
readToLess: String;
readToGreater: String;
readAWord(endoftag: Boolean;endofnode: Boolean): String;
readCDATASection: String;
readAttributesTo(att_list:TAttributeList; endofnode: Boolean);
```

The next 5 procedure will be overwritten as event-functions. These are in SAX_handler.inc file.

```
startDocument;
endDocument;
startElement(ename: String;attr_list:TAttributeList);
endElement(ename: String);
findCharacters(value: String);
```

The following is a co-function for parsing the attributes in the opening tag.

```
getAttributeValue(attr_list:TAttributeList;aname: String):
String;
```

Algorithms

Implementation of SAX processing (SAX_handler.inc)

startElement(ename:String;attr_list:TAttributeList)

It runs at the reading of opening tag.

```
switch
  in case of ename='example'
    //Beginning of table
    addString(' <table border="1" cellpadding="0"
cellspacing="0">');
    addString('  <tr><td>ETR id</td><td>Name</td><td>E-
mail</td></tr>');
  in case of ename='teacher'
    //Beginning of the row
    line:='  <tr';
    //if there is an attribute, we read the value of
"executive"
    if attr_list.count>0 then
      value:=getAttributeValue(attr_list,'executive');
      //If the value is "true", the class of style will be
"gray" and it will appear with gray background colour.
      if value='true' then
        line:=line+' class="gray"';
      end
    end;
    line:=line+'>';
    addString(line);
  in case of ((ename='etrid') or (ename='name') or
(ename='email'))
    addString('   <td>');
end;
```

endElement(ename: String)

It runs at the reading of closing tag.

```
switch
  in case of ename='example'
    //end of the table
    addString(' </table>');
  in case of ename='teacher'
    //end of the row
    addString('  </tr>');
  in case of ((ename='etrid') or (ename='name') or
(ename='email'))
    addString('   </td>');
end;
```

Implementation of getAttributeValue (saxunit.pas)

getAttributeValue(attr_list:TAttributeList;aname: String): String

We use linear searching to find the value to the “aname” in the list of attribute.

```
value:='';
i:=1;
while ((i<=attr_list.count) and
(attr_list.item[i].name<>aname))
  i:=i+1;
end;
if (i<=attr_list.count) then
  value:=attr_list.item[i].value;
end;
getAttributeValue:=value;
```

Implementation of parseXML (saxunit.pas)

parseXML(fname: String);

This is the general parsing procedure which reads sequentially the XML file. The name of file is the parameter. It starts with opening the file, calling the startDocument procedure and ends with closing the file and calling the endDocument procedure. At first we find – read the characters to – the beginning of the first tag (the first “<” (left angle bracket) character). The processing starts here and it repeats to the end of the file. It reads the first character by which it makes a junction and process the opening tag. Then it reads the data till the next “<” (left angle bracket) character. There are four (4) possibilities at the junction.

1. The “/” (per) character means the closing tag. It must read the name and call the endElement procedure.
2. The “?” (question-mark) character means processing instruction. It must read to the “>” (right angle bracket) character but now we do not handle it.
3. The “!” (exclamation mark) character means special element. In this time it must read the next character, too, and make a junction again with four (4) possibilities.
    a. The “-“ (hyphen) character shows comment. It reads to the “>” (right angle bracket) character but now we do not handle it as in the second case.
    b. The “D” character means document type definition. It reads to the “>” (right angle bracket) character again and we do not handle it as before.
    c. The “[“ (opening square bracket) character begins the CDATA section. It is closed by “]]>” (closing square bracket, closing square bracket, right angle bracket) characters. It must read the content of section and process it as characters.
    d. Otherwise it is unknown, so it must read to the “>” (right angle bracket) character but now we do not handle it as before.
4. Other character means opening tag. It must read the name of tag where the first character has already been read before. And it must read the attributes if the tag has not been ended yet (“endoftag”). If the tag contains a “/” (per) character in the end it means it is the closing tag, too. It is an empty element, and endElement must be called.

```
//Clear the result string before the new processing.
```

```
setString('');
OpenFile(f_xml,fname);
//start of the document
startDocument;
//Read content of file to the first ”<” (left angle bracket)
character.
s:=readToLess;
while (not end_of_file(f_xml))
  s:=readForwardACharacter;
  switch
    in case of s='/'
      //end of element
      ename:=readAWord(endoftag,endofnode);
      endElement(ename);
    in case of s='?'
      //Processing instruction - skip
      s:=readToGreater;
    in case of s='!'
      //Special element
      s:=readForwardACharacter;
      switch
        in case of s='-'
          //Comment (<!--...-->)'
          s:=readToGreater;
        in case of s='D'
          //DTD (<!DOCTYPE...>)'
          s:=readToGreater;
        in case of s='['
          //CDATA (<![CDATA[...]]>);
          s:=readCDATASection;
          if s<>'' then
            findCharacters(s);
          end
        otherwise
          //Unknown special element
          s:=readToGreater;
      end;
    otherwise
      //start of element, s contains the first letter of the
name!
      ename:=readAWord(endoftag,endofnode);
      ename:=s+ename;
      att_list.count:=0;
      if not endoftag then
        readAttributesTo(att_list,endofnode);
      end
      startElement(ename,att_list);
      //If the opening tag contains a "/" (per) character at
the end it is an empty element and it must be closed
      if endofnode then
        //End of element
        endElement(ename);
      end
```

```
    end;
    //Read the characters to the next (opening or closing) tag
  which starts with “<” (left angle bracket) character. The trim
  function cut the needless characters from the beginning and
  the end of it.
    s:=trim(readToLess);
    if s<>'' then
      findCharacters(s);
    end;
  end;
  //End of document
  endDocument;
  CloseFile(f_xml);
```

Implementation of co-functions (saxunit.pas)

readForwardACharacter: String;

It reads a character (reading forward) and returns with it as a string.

```
  KaraktertOlvas(f_xml,ch);
  readForwardACharacter:=ch;
```

readToLess: String;

It reads a character and append to the collected string while it does not find “<” (left angle bracket) character. The last read character is skipped because it is the “<” (left angle bracket) character.

```
  s:='';
  s1:=readForwardACharacter;
  while ((not end_of_file(f_xml)) and (s1<>'<'))
    s:=s+s1;
    s1:=readForwardACharacter;
  end;
  readToLess:=s;
```

readToGreater: String;

It reads a character and append to the collected string while it does not find “>” (right angle bracket) character. The last read character is skipped because it is the “>” (right angle bracket) character.

```
  s:='';
  s1:=readForwardACharacter;
  while ((not end_of_file(f_xml)) and (s1<>'>'))
    s:=s+s1;
    s1:=readForwardACharacter;
  end;
  readToGreater:=s;
```

readAWord(var endoftag: Boolean;var endofnode: Boolean): String;

It reads a character and append to the collected string while it does not find “>” (right angle bracket), “/” (per) or “ “ (space) character. The last read character must be listened because it shows the end of the tag or node. It is dropped because it is the “>” (right angle bracket) character. This algorithm is based on the SzóOlvas ([5]) it is extended with the two markers parameter.

```
endofnode:=FALSE;
s:='';
s1:=readForwardACharacter;
while ((not end_of_file(f_xml)) and (s1<>'>') and (s1<>' ')
and (s1<>'/'))
    s:=s+s1;
    s1:=readForwardACharacter;
end;
if s1='/' then
    endofnode:=true;
    s1:=readToGreater;
    endoftag:=true;
else
  endoftag:=(s1='>');
end;
readAWord:=s;
```

readCDATASection: String;

It resembles to the previous readAWord but it must read forward 3 characters and listens to the end-mark “]]>”.

```
Homework!
```

readAttributesTo(var att_list:TAttributeList;var endofnode: Boolean);

It reads words with using a post-test loop. A word is an attribute which is cut to name and value by the “=” (character) and these are stored in the list of attributes. It transmits the value of “endofnode” parameter.

```
do
//Reading a word.
  s:=readAWorld(endoftag,endofnode);
//Increment the count of the attributes
  att_list.count:=att_list.count+1;
//Look for the position of "=" (equality) character
  p:=Pos('=',s);
//Copy the name of attribute from the string. Do not copy the
"=" (equality) character.
  att_list.item[att_list.count].name:=CopyAPartOfString(s,1,p-
1);
//Copy the value of attribute from the string. Do not copy the
""" (quote) characters.
  att_list.item[att_list.count].value:=CopyAPartOfString(s,
p+2, length(s)-p-2);
while not (end_of_file(f_xml) or endoftag);
```

3. Survey and opinions

I conducted two lessons on ELTE. They are in minor BsC informatics teacher course. The name of the lesson is “Programozási Alapismeretek (M1,M2)” ([6]). These two lessons were on 4th March 2010. between 10:15 and 11:45 and between 12:15 and 13:45.

The first group did not get the handout ([7]) but I gave it to the second group. Independently from this I think the second group was more active and more inquisitive. The first group was not able to answer the question: “Which tags must be written to define a table? (table, tr, td)”. They knowledge of HTML is too high (3) or they were not so active and they hated the unknown, non-regular teacher.

This matter is enough for two (2) lessons.

On these lessons I did not have time to code the algorithms. In the first group we could talk about the algorithms without implementation (4.b); 3.b)). In the second group 2-3 people’s teams were coding the functions but we did not able to build together. Every team made one function and they did not see the other solutions.

It is confirmed by our measuring, too. My colleagues and I recorded the minutes at the checkpoints in the lesson plan. The next table shows the proposed and the real time in the first part of the lesson:

Title	Proposed time (minutes)	Real time (minutes)
Introduction	6	5
XML review	8	15
Processing XML	8	5
Reviewing about SAX_template	8	15
1. practise	11	20
2. practise	9	20
Total	**50**	**80**

Table 4: Elapsed times on first part of the lesson

The leisurely tempo wants to cut here the lesson.

Regarding to another idea it can be change the order. On the first coding lesson students implement the functions based on the given algorithms and only on the second lesson they can understand the motive (XML processing).

So the next two lesson plans can be applied:

Title	Time (minutes)
Introduction	5
XML review	15
Processing XML	10
Reviewing about SAX_template	15
1. practise	25
2. practise	20
Introduction	5
3.a)	20
4.a)	20
4.b)	20
3.b)	25

Table 5 a): Lessons plan A

Title	Time (minutes)
Introduction	5
3.a)	20
4.a)	20
4.b)	20
3.b)	25
Introduction	5
XML review	15
Processing XML	10
Reviewing about SAX_template	15
1. practise	25
2. practise	20

Table 5 b): Lessons plan B

I asked the students to fill out the following survey.

Classify, your knowledge … or, how do you like …	**1 insufficient not …**	**2 sufficient**	**3 medium pass for**	**4 good**	**5 excelent very …**
Questions to the beginning of the lesson:					
Your mark in “Webfejlesztés 1” (Leave it blank if you do not have)					
Knowledge of HTML (according to you)					
Knowledge of XML before the lesson					
knowledge of XML parsing before the lesson					
Knowledge of Pascal programming language					
Knowledge of Lazarus IDE					
Questions to the end of the lesson:					
Your attention on the lesson					
Your knowledge in this theme at the end of the lesson. (How do you feel?)					
… topic of the lesson					
… order of the lesson					

… trained of the teacher					
… presentation of the teacher					
Rate the template application as appearance					
Rate the template application as useable					
Rate the template application as implementation					
…1. SAX parsing (create “event-functions”)					
…2.implementation of getAttributeValue (linear searching theorem)					
…3.implementation of parseXML (complex task, requiring more thinking / thought-provoking)					
…4.implementation of co-functions (easier, “more precisely” task)					
What do you like on the lesson?					
What do NOT you like on the lesson?					

Table 6: The survey

The first part of the survey was filled out on the beginning of the lesson. The second part was filled out on the end of the lesson.

I attached the result, the 5_results_hu.xlsx file into the [1].

I present here my conclusions only:

- Their knowledge about HTML is medium (3)
- They have not XML knowledge (1.175)
- Pascal programming language is known moderately (2.8)
- Lazarus IDE knowledge is low (1.85)
- Their attention was arrested well (3.8)
- They learned something (2.5)
- The lesson
  - The topic is good (4,05)
  - The order of the lesson is good (3.9)
  - My preparedness was good (4.65)
  - My presentation was good (4.15)

    - These questions were rated well by the second group (0.7). They could work from the handout. Maybe it was better.
- The template application is good (4.01). It liked more in the second group, too.
- The awarding of SAX parsing was better with 0.5 in the second group (3.3).
- The implementation of getAttributeValue was medium (3.05). They did not know the algorithm of linear searching. It was only coding exercise.
- The last two exercises were not so good (3). We could only talking about these but we did not have time to try them.
- So medium HTML knowledge is enough for understanding XML and do transformation to HTML.

I regret but I forgot to make a vote about the next question: "What do you think this course let be on the next semester or not? (Y/N)"

4. Conclusion

In this paper I presented a new idea for teaching string processing, a lesson plan is available and I analysed the survey that was filled in the two test-lessons. I compressed and uploaded the presentation, the lesson plan, the survey, the handout, the results and the applications to the URL [1].

The curriculum is too much for 90 minutes, but it is complete in this format. That is why I suggest 2 lessons with 90-90 minutes. The reception was good. So it can require complementary work in the future.

Referencies

1. Menyhárt, László: *String Processing and XML*, INFODIDACT 2010, Szombathely (2010) http://xml.inf.elte.hu/konferencia/2010/INFODIDACT/StringProcessing&XML.zip
2. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C (2008) http://www.w3.org/TR/xml/
3. *Document Object Model (DOM) Level 1 Specification*, W3C (1998) http://www.w3.org/TR/REC-DOM-Level-1/
4. *Simple API for XML* (2004) http://www.saxproject.org/
5. Pap, Gáborné; Szlávi, Péter; Zsakó, László: *mikrológia 14, Módszeres programozás: Szövegfeldolgozás* (1995) 4. kiadás
6. Programozási alapismeretek (M1,M2) (2010) http://progalapm.elte.hu