Structured Document Processing using Lex and Yacc

U. Timoshkina, Y. Bogoyavlenskiy, M. Penttonen

Report

ISBN 951-781-260-4
ISSN 0787-6416

UNIVERSITY OF KUOPIO

Department of Computer Science
and Applied Mathematics

P.O.Box 1627, FIN-70211 Kuopio, FINLAND

Structured Documents Processing Using Lex and Yacc

Uljana Timoshkina, Yury Bogoyavlenskiy* Martti Penttonen†

Abstract

This report studies the applicability of the general purpose programming language compiler tools and web browsers for processing structured documents. In particular, the scanner generator `lex` and the parser generator `yacc` together with the compiler `gcc` are used for transforming the document structure, while the browser `Netscape` is used for browsing and creating the contents of the structured documents. As an application, a work plan is developed as a structured document. It is assumed that the reader is familiar with the `C` programming language.

1 Introduction

Processing and using structured documents are likely to become widely used in many software projects. Structured documents are used for processing work documentation, for representing data, as intermediate format for data exchange between applications, for encoding messages and remote procedure calls in transport protocols. Nowadays, there are a number of technologies for processing structured documents. A large part of this methodology is developed around W3C and, therefore, these technologies are standardized and well supported.

A *structured document* is a document containing both its logical structure and meaningful data provided by the user. One of the ways to add logical structure to documents is to use markup. Markup means adding tags to electronic data to specify style or add structure to the data. SGML and XML [3, 4] utilize tags in document instances for the description of the logical structure of the document. The start tags and end tags are used to separate logical units of the document content. Start tags can contain additional information for application to process the documents. This kind of information are called *attributes*, and represented by pairs of name and value.

Example 1 (A structured document.)

```
<!-- Example of database fragment -->
<?xml version="1.0" encoding="ISO-8859-1" ?>
<db>
<person idnum="1234"><last>Kilpelainen</last><first>Pekka</first></person>
<person idnum="5678"><last>Penttonen</last><first>Martti</first></person>
<person idnum="9012"><last>Kuikka</last><first>Eila</first></person>
<person idnum="3456"><last>Leinonen</last><first>Paula</first></person>
<person idnum="3011"><last>Bogoyavlenskiy</last><first>Yury</first>
    <status>visitor</status></person>
<person idnum="3012"><last>Timoshkina</last><first>Uljana</first>
    <status>visitor</status></person>
</db>
```

Example 2 (The DTD G_1 describing the fragment of database.)

```
<!-- XML database fragment DTD -->
```

*Petrozavodsk State University, Department of Computer Science, Lenin's Street 33, 185640 Petrozavodsk, Russia, email {uljatim,ybgv}@cs.karelia.ru

†University of Kuopio, Department of Computer Science and Applied Mathematics, P.O.Box 1627, 70211 Kuopio, Finland, email penttonen@cs.uku.fi

```
<!ELEMENT db (person)* >
<!ELEMENT person (last,first,status?) >
<!ATTLIST person idnum ID #REQUIRED >
<!ELEMENT last (#PCDATA) >
<!ELEMENT first (#PCDATA) >
<!ELEMENT status (#PCDATA) >
```

The processing of structured documents includes creating, browsing, transforming, checking syntactic correctness, and passing structured documents. It is possible to use even a text editor (emacs) or a specific editor (SYNDOC [7]) for creating structured documents. In the first case, user should add both logical structure and data. In the second case, the editor writes tags and checks logical structure in accordance with user's actions and the user can pay more attention to meaningful data. A text editor can be used for browsing structured documents, but it is not convenient because of mixed markup and data. The standard ways to browse structured documents are to use style sheets with a web browser or an algorithm to transform structured documents into HTML pages.

Transformation of structured documents is a more complex task: one format is used as input for a translator and another is used as output. Transformations of structured documents can be done in different ways and with different tools. For this aim interfaces such as XPath and XSLT can be used. Besides, there are projects for the transformation of structured documents using tree transducers for the implementation changes in structure [5, 6].

The processing (parsing, checking and passing) of structured documents can also be implemented as programmed manipulation of structured documents, for example with SAX and DOM APIs. The tasks of building a parse tree and passing the document content are simple because of the complete parenthesization of the XML markup in input documents. The task of checking of syntax is only a little more difficult to implement: pulling the entities together, checking the well-formedness, checking the validity with written DTD or a Scheme (DTD is a sort of a high-level grammar definition).

The aim of this work is to use existing compiler tools such as `lex`, `yacc` [2, 8], together with a Web browser and C language compiler `gcc`, for processing structured documents, to test the usability of these tools for structured document processing. Netscape is used for creating and browsing the contents of the structured documents, `lex` and `yacc` are used for developing the translators, are transforming from one format to another or processing content data.

The basic scheme of this work is represented in Figure 1, where it is possible to see the above mentioned tools and their usage.

In Figure 1, for each document type definition `MyDTD` two files `MyDTD.l` for `lex` and `MyDTD.y` for `yacc` are needed. Furthermore, additional files can be required (but are not necessary) for processing the document type. These files contain additional functions and procedures which can be used only for this DTD or can be used as common part for processing different kinds of DTDs. The output of `lex`, `yacc` and `gcc` block is an executable program `MyTranslator` processing structured documents conforming `MyDTD`. The input of `MyTranslator` may be, for example, an XML tagged document `MyDoc`, and the output, for example, an HTML representation of the document. `MyDoc` can be created directly by a text editor or by an `Interface part` as it is shown in the picture. A user can browse and edit the contents of `MyDoc` by a Web browser. When `MyTranslator` is run it translates `MyDoc` conforming `MyDTD` into `Changed MyDoc` conforming `MyDTD` or another DTD.

The rest of the report consists of the description of `lex` and `yacc` tools (section 2), examples of using `lex` (subsection 2.1) and `yacc` (subsection 2.2) for processing structured documents. Subsection 2.3 is devoted to the idea of using Netscape to implement the interface parts, which allow to browse and to create data content of structured documents. There is the brief description in section 3 of the real system of processing work documentation of the Department of Computer Science of the Petrozavodsk State University. At last, in the Conclusion, we discuss advantages and disadvantages of the studied approach for processing structured documents.

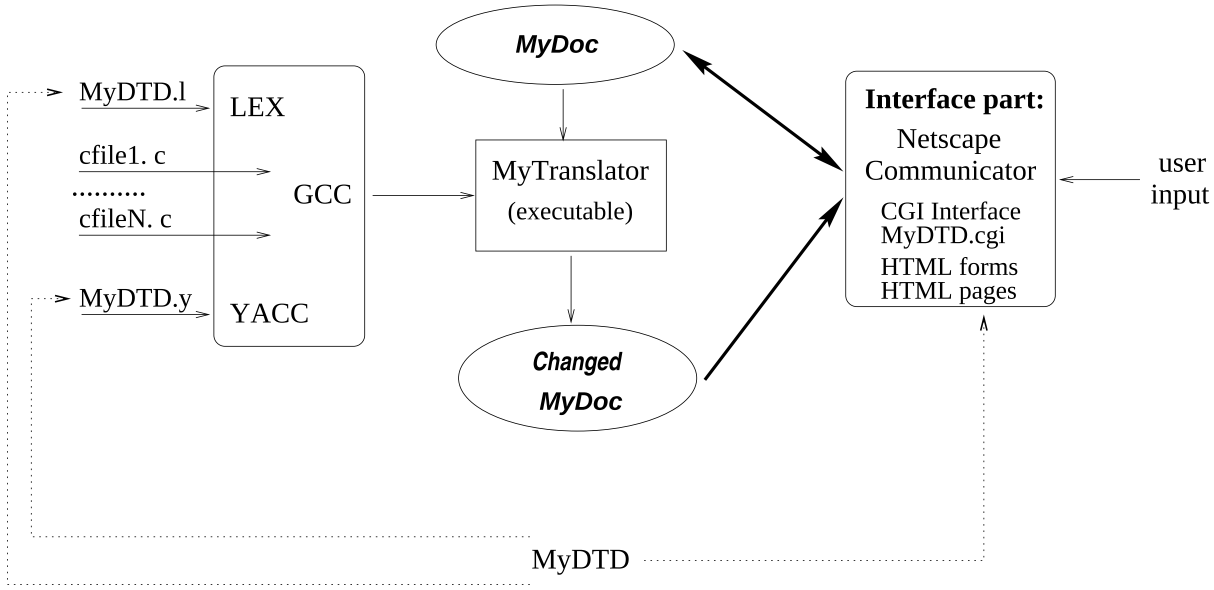

Figure 1: The basic scheme of the implementation

2 About tools

The methodological basis of structured documents is the theory of formal languages and automata [1, 10, 11]. Markup languages are usually defined by extended context-free grammars. The extended Backus-Naur Form (BNF) of context free grammar is suitable and widely used for defining the structure of the documents. Extended Backus-Naur Form of context-free grammar allows to use regular expressions to present the syntax more compactly and clearly. There is a well-known theorem stating, that extended and ordinary context free grammars generate the same set of languages [12]. So, both EBNF and BNF suit for describing logical structures of documents.

Documents are considered as syntactical structures that are characterized, recognized and manipulated by devices such as grammars, automata and transducers. Document processing can be considered as a sequence of transformations from one form to another. Every document has its own structure and this structure can be described by a context-free grammar. It allows to perform these transformations using translators of corresponding context-free languages.

For programming language translations there have been developed tools that essentially simplify design and implementation of the translators, like `lex` (lexical analyzer generator) and `yacc` (yet another compiler-compiler)[8]. As a matter of fact, the tools are high level CASE tools. `Lex` is a tool for automatic generation of lexical analyzers (scanners) based on a given specification, consisting of lexical rules in form of regular expressions that describe the tokens of the language. `Yacc` is a tool for automatically generating a parser, based on a given specification, which is an LALR(1) [1] grammar with actions. LALR(1) grammars are expressive enough to cover a wide class of programming languages and documents.

`Lex` and `yacc` use specifications, describing the *scanner* and *parser* to be generated. These specifications consist of three sections: *definitions*, *rules*, and *user code*. The `lex` rules section contains regular expressions and C code. When a generated scanner is run, it analyzes its input for occurrences of tokens matching the regular expressions. Whenever a token matches a subexpression, the scanner executes the corresponding C code. The `yacc` rules section contains grammar rules and C code (actions). Parser generated with `yacc` tries to derive the start symbol from input sequence using *shifts* and *reductions* corresponding to the defined grammar rules. Whenever it applies grammar rules it also performs the corresponding C code.

Parser, generated with `yacc`, uses the scanner, generated with *lex*, as a subroutine for reading the input document. This can significantly reduce the complexity of the parser, because the scanner can perform a lot of minor processing and hide unnecessary details from the parser. Actions of `lex` and yacc specifications provide a way to implement the semantic analysis of documents that is a principal component of any translation and it is its most complex part.

2.1 Using lex

lex specification describes the patterns of the tokens of a language and the actions necessary to perform when input string matches any of the patterns. A specification with patterns and actions is input for lex as a text file and it generates a C-verbatim. This file is compiled and linked (with gcc) to an executable program of scanner. This scanner is used for processing structured documents without checking its syntactic correctness. In a typical situation the scanner is used as a subroutine for a parser, generated with yacc .

Example 3 (Scanner for transforming an XML document directly to a HTML document.) *The database fragment specified in Example 1 is transformed to the HTML table below:*

```
<html>
<head>
<title>Example</title>
</head>
<body>
<table>
<caption><font size="14">Contents of DB</font></caption>
<tr><th>Last name</th><th>First name</th><th>Number</th></tr>
<tr><td>Kilpelainen</td><td>Pekka</td><td>"1234"</td></tr>
<tr><td>Penttonen</td><td>Martti</td><td>"5678"</td></tr>
<tr><td>Kuikka</td><td>Eila</td><td>"9012"</td></tr>
<tr><td>Leinonen</td><td>Paula</td><td>"3456"</td></tr>
<tr><td>Bogoyavlenskiy</td><td>Yury</td><td>"3011"</td></tr>
<tr><td>Timoshkina</td><td>Uljana</td><td>"3012"</td></tr>
</table>
</body>
</html>
```

Assuming that the document of Example 1 is syntactically correct, we can transform it to HTML form by the following lex *specification:*

```
%{ //begin of the section of definitions
#include <string.h>
#include <stdlib.h>
#include <alloca.h>

#define ECHO (void)fwrite(yytext,yyleng,1,stdout)

//description of work variables

char lastname[80];  //for storing last name
char firstname[80]; //for storing first name
char idnum[20]; //for storing attribute
char * tmpstr;

int flag; //flag variable
int i; //for loops

%}
```

```
    //declaration of simple name definition to simplify the scanner
    //specification using regular expressions

STROKA  [a-zA-Z]+
NUMBER [0-9]+
NOTHING [ \t\n]+

    //declaration of start conditions

%s PERSONATTR
//end of the section of definitions
%% //begin of the section of rules

{NOTHING} //Find spaces, tabs, end-of-line codes

"<?".+"?>" //Find processing instruction


"<db>" { fprintf(yyout,"<html>\n<head>\n");
  fprintf(yyout,"<title>Example</title>");
  fprintf(yyout,"\n</head>\n");
  fprintf(yyout,"<body>\n<table>\n");
  fprintf(yyout,"<caption><font size=\"14\">");
  fprintf(yyout,"Contents of DB");
  fprintf(yyout,"</font></caption>\n");
  fprintf(yyout,"<tr><th>Last name</th>");
  fprintf(yyout,"<th>First name</th>");
  fprintf(yyout,"<th>Number</th></tr>\n");
}
/*Find root element. Start to scanner input sequence.
  It is necessary to create html-page, table and header
  for table.
*/


"</db>" { fprintf(yyout,"</table>\n</body>\n</html>");
}
/*Find end of root element. End of scanning input sequence.
  This means time for closing table and html-page.
*/


"<first>" { tmpstr=firstname;
}
/*Find start tag for FIRST element.
  Remember that we are in FIRST element.
*/


"</first>" { tmpstr=NULL;
}
/*Find end tag for FIRST element.
  We leave FIRST element.
*/


"<last>" { tmpstr=lastname;
}
/*Find start tag for LAST element.
```

```
  Remember that we are in LAST element.
*/

"</last>" { tmpstr=NULL;
}
/*Find end tag for LAST element.
  We leave LAST element.
*/

"</person>" { fprintf(yyout,"<tr><td>%s</td>",lastname);
  fprintf(yyout,"<td>%s</td>",firstname);
  fprintf(yyout,"<td>%s</td></tr>\n",idnum);
  *(lastname)='\0';
  *(firstname)='\0';
  *(idnum)='\0';
}
/*Find end tag for PERSON element.
  Output corresponding row of table.
*/


"<person" { BEGIN(PERSONATTR);
}
/*Find start tag for PERSON element.
  Prepare to read attributes.
*/


<PERSONATTR>{STROKA}"="\"{NUMBER}\" { flag=-1;
//Find a value of idnum attribute.
  for(i=0;i<yyleng;i++)
  {
   if (*(yytext+i)=='=')
      { flag=0;}
   else if (flag>-1)
   { *(idnum+flag)=*(yytext+i);
    flag++;
   };
  };
  *(idnum+flag)='\0';
}
/*Find attribute for PERSON element.
  Store its value in idnum variable.
*/

<PERSONATTR>">" { BEGIN(INITIAL);
}
/*Processing of start tag of PERSON element is over.
*/


{STROKA} { if (!(tmpstr==NULL)) strcpy(tmpstr,yytext);
}
/*Find text data. Copy it in corresponding string.
*/

    //end of the section of rules
%%     //begin of the section of user code
int yywrap()
```

```
/* When the scanner receives an end of file,
   it then runs this function.
*/
{
    return 1; /* Non-zero value terminates the scanner after
 reading end of file.
      */
}

int main(void) // The main function of program for generating
       /* html-page. It contains the call of the
          scanner yylex(), generated by lex tool.
       */
{
    char inpfile[10];
    char outfile[10];


    printf("Enter name of input file:");
    scanf("%s",inpfile);
    printf("\n");

    printf("Enter name of output file:");
    scanf("%s",outfile);
    printf("\n");

    yyin=fopen(inpfile,"r");
    if (yyin==NULL)
    { printf("Error of opening %s\n",inpfile); return 1; }

    yyout=fopen(outfile,"w");
    if (yyout==NULL)
    { printf("Error of opening %s\n",outfile); return 1; }

    /* Call scanner yylex(). After its work html-page will be
       created. It is possible to open it in Netscape Navigator
       to watch it.
    */

    if (yylex()==0)
    { fclose(yyin);
      fclose(yyout);
      printf("Done");
      return 0;
    }
    else
    { fclose(yyin);
      fclose(yyout);
      printf("Error");
      return 1;
    }
}
//end of the section of user code
```

The scanner generated in accordance with this specification creates an HTML page for XML documents, but the scanner doesn't check the syntactic correctness of documents. As HTML pages can be created even if the XML input is not syntactically correct, the syntactic correctness of the HTML output is not guaranteed.

2.2 Using `yacc`

`yacc` specification defines the grammar for the documents to be parsed. An `yacc` specification is similar to a DTD, but this "DTD" contains both logical structure and actions for processing structured documents. DTD is a sort of higher-level grammar definition, while *yacc* specification only uses the Backus-Naur Form of context-free grammars. We build a grammar in a way to use tag structure and data in input files.

`yacc` specification describes and defines the parser to be built. Actions of `yacc` specification will be performed when a logical part of document is found. Grammar rules define possible input documents, so parser generated with `yacc` also checks the input sequence for syntactic correctness against defined grammar.

The Examples 6 and 7 show how `lex` and `yacc` tools can help to design a program, which creates a new structured document for the data base fragment presented in Example 1. A new format consists of an ID of a person and person's E-MAIL, but all persons with status of visitor are enclosed in one logical part at the end of the document. The output format can be represented in this way using a DTD:

Example 4 (A DTD G_2 defining personal ID data.)

```
<!-- XML database fragment DTD -->

<!ELEMENT db (person*,visitors) >
<!ELEMENT visitors (person*)
<!ELEMENT person (id,email) >
<!ELEMENT id (#ID) >
<!ELEMENT email (#PCDATA) >
```

Example 5 (A document conforming G_2)

```
<db>

<person><idnum>"1234"</idnum><email>Pekka.Kilpelainen@cs.uku.fi</email></person>

<person><idnum>"5678"</idnum><email>Martti.Penttonen@cs.uku.fi</email></person>

<person><idnum>"9012"</idnum><email>Eila.Kuikka@cs.uku.fi</email></person>

<person><idnum>"3456"</idnum><email>Paula.Leinonen@cs.uku.fi</email></person>


<visitors>
<person><idnum>"3011"</idnum><email>Yury.Bogoyavlenskiy@cs.uku.fi</email></person>
<person><idnum>"3012"</idnum><email>Uljana.Timoshkina@cs.uku.fi</email></person>
</visitors>
</db>
```

In this case `lex` helps to generate a scanner, which returns to the parser the next input token (its type and its value). Parser moves into the next state, using information about current state and the next token. In this way, the input sequence is parsed.

The following `lex` and `yacc` specifications help to resolve the problem of transforming the structured document of Example 1 into structured document of Example 5.

Example 6 (`lex` specification.)

```
%{ //begin of the section of definitions
#include <string.h>
#include <stdlib.h>
#include <alloca.h>
```

```
#include "exyacc.h"
extern YYSTYPE yylval;

%}

STROKA  [a-zA-Z]+
NUMBER \"[0-9]+\"
NOTHING [ \t\n]+
//end of the section of definitions
%% //begin of the section of rules

    //The parser can ask for any of these tokens.
    //If the scanner finds matching token
    //it returns it to the parser.

{NOTHING}


"<?".+"?>" { return DOP; }
/*Processing instruction.*/

"<" { yylval.s=(char *)calloc(2,sizeof(char));
  strcpy(yylval.s,yytext);
  return LEFT;
}
/*Delimeter elements.*/

">" { yylval.s=(char *)calloc(2,sizeof(char));
  strcpy(yylval.s,yytext);
  return RIGHT;
}
/*Delimeter elements.*/

"=" { yylval.s=(char *)calloc(2,sizeof(char));
  strcpy(yylval.s,yytext);
  return IS;
}
/*Delimeter elements.*/

"/" { yylval.s=(char *)calloc(2,sizeof(char));
  strcpy(yylval.s,yytext);
  return SLASH;
}
/*Delimeter elements.*/

"db" { yylval.s=(char *)calloc(3,sizeof(char));
  strcpy(yylval.s,yytext);
  return DB; //Names of elements
}
/*Return name of DB element*/

"first" { yylval.s=(char *)calloc(6,sizeof(char));
  strcpy(yylval.s,yytext);
  return FIRST;
}
/*Return name of FIRST element*/

"last" { yylval.s=(char *)calloc(5,sizeof(char));
  strcpy(yylval.s,yytext);
```

```
  return LAST;
}
/*Return name of LAST element*/

"person" { yylval.s=(char *)calloc(7,sizeof(char));
  strcpy(yylval.s,yytext);
  return PERSON;
}
/*Return name of PERSON element*/

"idnum" { yylval.s=(char *)calloc(6,sizeof(char));
  strcpy(yylval.s,yytext);
  return IDNUM;
}
/*Return name of IDNUM attribute*/

"status" { yylval.s=(char *)calloc(6,sizeof(char));
  strcpy(yylval.s,yytext);
  return STATUS;
}
/*Return name of STATUS element*/

{NUMBER} { yylval.s=(char *)calloc(20,sizeof(char));
  strcpy(yylval.s,yytext);
  return NUM;
}
/*Return value of meaningful data*/

{STROKA} { yylval.s=(char *)calloc(80,sizeof(char));
  strcpy(yylval.s,yytext);
  return STR;
}
/*Return value of meaningful data*/

//end of the section of rules
%% //begin of the section of user code
int yywrap()
{
    return 1;
}
//end of the section of user code
```

Example 7 (yacc **specification.)**

```
%{ //begin of the section of definitions
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

extern FILE * yyin, * yyout;
extern char * yytext;
int yyparse(void);
void yyerror(char[]);
int yylex(void);

//definition of work variables

char str[250];
//end of the section of definitions
%} //begin of the section of rules
```

```
//type of grammar symbols

%union { char * s;
       }

//definition of grammar symbols

%token <s> DB
%token <s> PERSON
%token <s> FIRST
%token <s> LAST
%token <s> IDNUM
%token <s> STATUS
%token <s> NUM STR DOP
%token <s> RIGHT LEFT SLASH IS
%type <s> root person last first status
%type <s> contdb attr
%type <s> START
%%

/*This section is the analog of DTD. The grammar rules, defined here,
  present logical structure of the source document.
  A part of grammar for defining logical structure is presented
  in traditional form bellow in the comment:
  (it defines a context part of each element)

  START: DOP root
  root: LEFT DB RIGHT contdb LEFT SLASH DB RIGHT
  contdb: person | contdb person
  person: LEFT PERSON attr RIGHT last first LEFT SLASH PERSON RIGHT
                | LEFT PERSON attr RIGHT last first status LEFT SLASH PERSON RIGHT
          attr: IDNUM IS NUM
  last: LEFT LAST RIGHT STR LEFT SLASH LAST RIGHT
  first: LEFT FIRST RIGHT STR LEFT SLASH FIRST RIGHT
  status: LEFT STATUS RIGHT STR LEFT SLASH STATUS RIGHT
*/

START: DOP root { }
/*Input is parsed. Write new document into yyout file.
*/

root: LEFT DB RIGHT { fprintf(yyout,"<db>\n");} contdb LEFT SLASH DB RIGHT {
fprintf(yyout,"<visitors>\n%s</visitors>\n</db>",$5);
/*When entire document is parsed, then put also "visitors group".
*/

contdb: person { strcpy($$,$1); }
| contdb person { strcpy($$,$1); strcat($$,$2); }


person: LEFT PERSON attr RIGHT last first LEFT SLASH PERSON RIGHT {

 fprintf(yyout,"<person><idnum>%s</idnum>
 <email>%s.%s@cs.uku.fi</email></person>\n",$3,$6,$5);

 strcpy($$,"\0");
 /*Find PERSON element without STATUS element. Therefore, it is possible add it
   into the yyout file.
 */
 }
```

```
 | LEFT PERSON attr RIGHT last first status LEFT SLASH PERSON RIGHT {

 sprintf(str,"<person><idnum>%s</idnum>
 <email>%s.%s@cs.uku.fi</email></person>\n",$3,$6,$5);

 strcpy($$,str);
 /*Find PERSON element with STATUS element. Therefore, accumulate it in string
   variable $$. If entire document is parsed, then put all visitors.
 */
 }


attr: IDNUM IS NUM { strcpy($$,$3); }
/*Read attribute. Read value of idnum.
*/

/*Content of PERSON element*/

last: LEFT LAST RIGHT STR LEFT SLASH LAST RIGHT {
    strcpy($$,$4);
  }
/*Find LAST element. Read value of last name.*/


first: LEFT FIRST RIGHT STR LEFT SLASH FIRST RIGHT {
    strcpy($$,$4);
    }
/*Find FIRST element. Read value of first name.*/

status: LEFT STATUS RIGHT STR LEFT SLASH STATUS RIGHT {
    strcpy($$,$4);
    }
/*Find STATUS element. Read value of status.*/

    //end of the section of rules
%%     //begin of the section of user code

void yyerror(msg)    //The error handler function.
char msg[];
{
    printf("yacc %s \n",msg);
}

int main(void)      //The main function contains the call
     //of the parser yyparse(), generated by yacc.
{
    char inpfile[10];
    char outfile[10];


    printf("Enter name of input file:");
    scanf("%s",inpfile);
    printf("\n");

    printf("Enter name of output file:");
    scanf("%s",outfile);
    printf("\n");

    yyin=fopen(inpfile,"r");
    if (yyin==NULL)
```

```
    { printf("Error of opening %s\n",inpfile); return 1; }

    yyout=fopen(outfile,"w");
    if (yyout==NULL)
    { printf("Error of opening %s\n",outfile); return 1; }

    /* Call parser yyparse(). It parses input sequence and produces
       the new structured document.
    */

    if (yyparse()==0)
    { fclose(yyin);
      fclose(yyout);
      printf("Done");
      return 0;
    }
    else
    { fclose(yyin);
      fclose(yyout);
      printf("Error");
      return 1;
    }
}

//end of the section of user code
```

2.3 Web Browser Interface

A Web browser can be used as a natural facility for visualization and interface with an application. An HTML form is used for filling the content data of the document. These data are processed through a CGI interface, but it is also possible to write them into file in the pre-defined format. CGI programs are implemented in programming language C and they also can use subroutines generated with `lex` and `yacc`. These subroutines are suitable for reading data from some source file, with aim to produce a stage of presentation. (For example, it is possible to use the following form (Figure 2) to create a structured document).

All examples in this section are taken from the system[1] of processing work documentation of the Department of Computer Science (DWDoc) of the Petrozavodsk State University.

CGI program processes the HTML-form and writes data into a file in a pre-defined format. The fragment of this file is given bellow:

Example 8 (XML document created with interface presented in Figure 2.)

```
<department> 21
<speciality> 510202
<year> 1
......
......
</year>
<year> 2
<group> 1 </group>
<subgroup> 2 </subgroup>
<student> 15 </student>
<semester> 1
<week> 16 </week>
```

[1]The system is being implemented by Uljana Timoshkina.

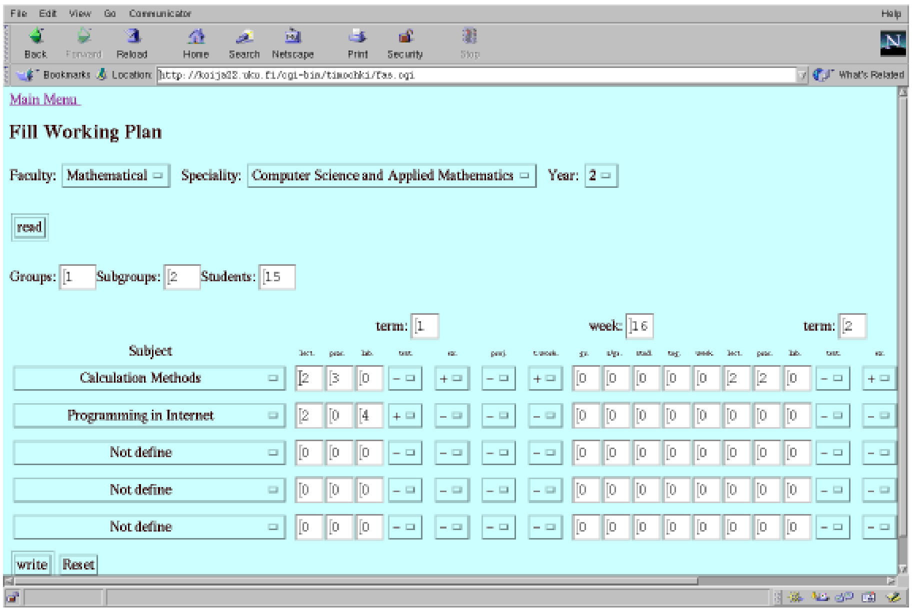

Figure 2: Interface for filling a structured document.

```
<subject>
<name> 7 </name>
<lecture> 2 </lecture>
<practice> 3 </practice>
<exam> + </exam>
<testwork> + </testwork>
</subject>
<subject>
..............
```

HTML pages are used for browsing the output forms of an application. In the example 3 (section 2.1) we show how `lex` may be used to generate HTML-page. In this way all output forms can be represented as HTML-pages and the user can watch them in the convenient form.

3 The DWDoc System

The method described above is used for developing the DWDoc, namely for the implementation of an application for the distribution of staff workload. All manipulations of documents in this system are considered as transformations from one logical structure to another with processing of the data contents. The manipulations are implemented with `lex` and `yacc`, interface parts are implemented using CGI and HTML forms technique.

In the DWDoc `Working Plan` document is transformed into `Calculation of Staff` document and `Distribution` document. User interface of the system is presented in Figure 3.

In the process of the DWDoc implementation:

- The input formats are described by a context-free grammar.
- `yacc` is used for creating a parser in accordance with grammar rules.
- `lex` is used for creating a scanner as a subroutine for a parser.

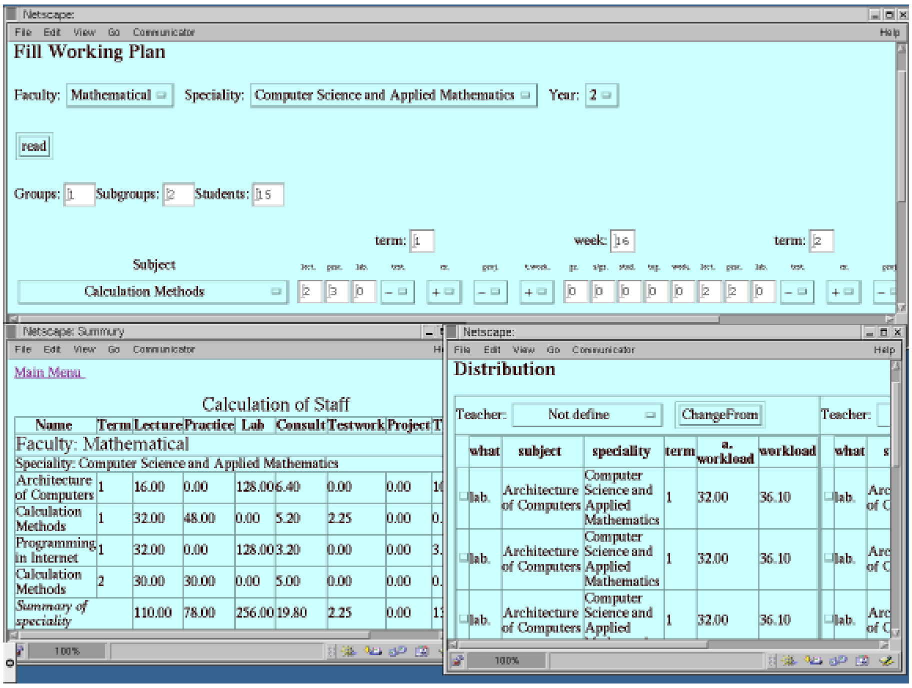

Figure 3: Browsing of some structured documents of the system.

- `Netscape` is used as an interface tool.

The transformation from one logical structure to another requires the knowledge of the structures of both the source and the target documents. Each source document is described by a context-free grammar, which is the main part of the specification for `yacc`. The specification consists of instructions and descriptions for `yacc` to generate a parser with actions. The output document is produced by actions of the `yacc` specification. This output is generated in accordance with a grammar describing the output file. It is important that this approach gives possibility to process and change not only the logical structure but also the data contents of the documents. The data contents manipulations are defined as actions of `lex` and `yacc` specifications. In the DWDoc system the idea of transformations is the following: the grammar in the `yacc` rule section describes the input document structure, the actions specify the structure of the output document.

4 Conclusion

We can point out some positive and negative aspects of the studied approach.

Some advantages are:

- A developer can define the grammars of documents by himself using BNF.
- A developer can describe the transformation of the structure and the manipulations of the data content by actions. The complexity of the actions is limited only by the skills of the developer.
- Only actions must be coded in C manually, while automata for scanning and parsing are produced automatically by `lex` and `yacc`.
- There is a possibility for creating an arbitrary number of translators to be used in a system for processing documentation.

- Using `yacc` tool to generate parser provides a possibility to check structured documents for their correctness: well-formedness of documents and validity of documents in accordance with a written grammar, since grammar rules in `yacc` specification define the document completely.
- Using `lex` tool for generating a scanner provides a possibility to process structured documents without checking them for conforming a pre-defined grammar. In this case performing of some actions is possible when start tag and end tag of logical markup are found.
- The following changes of a document structure can be implemented:
  - Deleting some elements from a document structure.
  - Adding some elements into a document structure.
  - Moving some elements from one level of a structure to another.
- `lex` and `yacc` are standard and well debugged tools. They provide a possibility of generation not only C code, but also C++, Java and Python codes.

Some disadvantages are:

- Parser, generated with `yacc`, uses a complex algorithm for getting a start symbol from input sequence. It uses bottom-up algorithm, while SAX and DOM use some kind of top-down algorithm for parsing input document instances.
- Unfortunately, if output file is used as input file for another translation, it is necessary to repeat the structure definition. It is required again in the rules sections of the `yacc` specification of the second translator, even if the same time the grammar was already defined in the rules of the previous translator.

In summary, we have shown that standard compiler tools — so-called compiler-compiler tools — can be also used for processing structured documents. Of course, the choice of tools depends on the concrete project and problem, but processing structured documents using `lex` and `yacc` is possible and acceptable.

References

[1] A.V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling, Vol. I: Parsing.* Prentice-Hall, Eaglewood Cliffs, N. J., USA, 1972.

[2] Y. A. Bogoyavlenskiy, D. G. Korzun. *LEX and YACC: generators of programs for lexical and syntactical analyses. Computer Science basics and applications*, Vol. 4. PGU, Petrozavodsk, to be published, (in Russian).

[3] N. Bradley. *The XML companion.* Addison Wesley, Harlow, UK, 1998.

[4] T. Bray, J. Paoli, C. M. Sperberg-McQueen and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition).* `http://www.w3.org/TR/2000/REC-xml-20001006`, 2000.

[5] E. Kuikka, P. Leinonen, and M. Penttonen. An approach to document structure transformations. In: Yulin Feng, David Notkin, Marie-Claude (eds): *Proceedings of Conference on Software: Theory and Practice*, pp. 906-913, 16'th IFIP World Computer Congress 2000, Beijing, China.

[6] E. Kuikka, P. Leinonen, M. Penttonen. Overview of tree transducer based document transformation system. In: Ethan V. Munson, Derick Wood (eds): *Preliminary Proceedings of the Fifth International Workshop on Principles of Digital Document Processing (PODDP'00)*, Munich, Germany, 2000.

[7] E. Kuikka, M. Penttonen, and M.-K. Väisänen. Theory and implementation of SYNDOC document processing system. *Proceedings of the Second International Conference on Practical Application of Prolog* (PAP-94). London, UK, 1994, pp.311-327.

[8] John R. Levine, Tony Mason, Doug Brown *lex & yacc* O'Reilly & Associates, Inc., 1992.

[9] D. Raggett, A. Le Hors, I. Jacobs. *HTML 4.01 Specification.* `http://www.w3.org/TR/1999/REC-html401-19991224`, 1999.

[10] V. J. Rayward-Smith. *A first course in Formal Language Theory.* Radio i sviaz, Moscow, 1988, (in Russian).

[11] V. A. Serebryakov. *A course in Construction of compilers.* Moscow State University, 1994, (in Russian).

[12] P. D. Terry. *Compilers and compiler generators.* Rhodes University, 1996.