

DESIGNING AND IMPLEMENTING A HTML LANGUAGE PARSER

REVIEW REPORT

FOR

Theory of Computation and Compiler Design
Embedded Project

FALL SEMESTER –2017-18

G1 SLOT

Group 3:

Chhatre Swapnil Shivaprasad –16BCB0040

Gurkaran Sahni – 16BCB0051

Kshaunish Roy – 16BCB0028

Shrutisha Joshi – 16BCE2304

Rohan Rakesh Kumar Sharma – 16BCE0454



VIT[®]
UNIVERSITY
(Estd. u/s 3 of UGC Act 1956)

SCOPE

VIT UNIVERSITY

VELLORE – 632 014 TAMIL NADU

INDIA

1. INTRODUCTION

A **parser** is a software component that takes input data (frequently text) and builds a data structure – often some kind of parse tree, abstract syntax tree or other hierarchical structure – giving a structural representation of the input, checking for correct syntax in the process. The parsing may be preceded or followed by other steps, or these may be combined into a single step. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters; alternatively, these can be combined in scannerless parsing. Parsers may be programmed by hand or may be automatically or semi-automatically generated by a parser generator. Parsing is complementary to templating, which produces formatted output. These may be applied to different domains, but often appear together, such as the `scanf/printf` pair, or the input (front end parsing) and output (back end code generation) stages of a compiler.

The input to a parser is often text in some computer language, but may also be text in a natural language or less structured textual data, in which case generally only certain parts of the text are extracted, rather than a parse tree being constructed. Parsers range from very simple functions such as `scanf`, to complex programs such as the frontend of a C++ compiler or the HTML parser of a web browser. An important class of simple parsing is done using regular expressions, in which a group of regular expressions defines a regular language and a regular expression engine automatically generating a parser for that language, allowing pattern matching and extraction of text. In other contexts regular expressions are instead used prior to parsing, as the lexing step whose output is then used by the parser.

The use of parsers varies by input. In the case of data languages, a parser is often found as the file reading facility of a program, such as reading in HTML or XML text; these examples are markup languages. In the case of programming languages, a parser is a component of a compiler or interpreter, which parses the source code of a computer programming language to create some form of internal representation; the parser is a key step in the compiler frontend. Programming languages tend to be specified in terms of a deterministic context-free grammar because fast and efficient parsers can be written for them. For compilers, the parsing itself can be done in one pass or multiple passes – see one-pass compiler and multi-pass compiler.

HTML parser is a software component which traverses the whole html document and displays the error such as unbalanced tags (if the html tag has a closing tag), Missing quotes for attribute values, etc. Our objective is to build a HTML parser which will parse the HTML document and prompt the errors in the code. This can be done by first tokenizing which is converting the whole HTML code into tokens. Then the parser converts that list of tokens to form a tree or graph that represents the source text in a manner that is more convenient to access/manipulate by the program. Then analysing the type of error and showing it to the user and suggesting ways for rectification.

2. BACKGROUND

In order to make an HTML parser, we first studied and understood what is HTML.

HTML

Hypertext Markup Language (HTML) is the standard markup language for creating web pages and web applications. With Cascading Style Sheets (CSS) and JavaScript it forms a triad of cornerstone technologies for the World Wide Web. Web browsers receive HTML documents from a web server or from local storage and render them into multimedia web pages. HTML de-scribes the structure of a web page semantically and originally included cues for the appearance of the document.

HTML elements are the building blocks of HTML pages. With HTML constructs, images and other objects, such as interactive forms, may be embedded into the rendered page. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes and other items. HTML elements are delineated by *tags*, written using angle brackets. Tags such as `` and `<input />` introduce content into the page directly. Others such as `<p>...</p>` surround and provide information about document text and may include other tags as sub-elements. Browsers do not display the HTML tags, but use them to interpret the content of the page.

In HTML there are basically two types of tags which are closing tag and the other one is non closing tag.

Closing tags are those tags which need to be closed every time they are opened for use, whereas non closing tags can be used only once and need not be closed.

For example, all the tags such as `<html>`, `<head>`, `<body>`, `` tags are closing tags whereas `
`, `<hr>` tags are not needed to be closed.

In a tag like: `<body bgcolor="red">` we have a set of attributes which are helpful in designing structure of a page in a more precised way. In order to make our parser, we need to analyze the syntax of how the attributes are defined.

The two languages that we are using in order to make the parser is lex and yacc.

Lex

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

YACC:

YACC stands for ***Yet Another Compiler Compiler***. Its job is to analyze the structure of the input stream i.e. grammar of tokens generated by lex. The lex already determines the series of tokens but it is the job of yacc to make decision.

YACC is a bottom up parser. A bottom-up parser builds a derivation by working from the input sentence back toward the start symbol S

In this each grammar is separately defined and separated by '|', also precedence is given to the grammar. It is not directly yacc which parses but the c-source code generated by it does the parsing job.

Yacc can even communicate with lex in case if both have to exchange some values. For this we define **YYTYPE**, that is the default value used to exchange data. Also if we want to exchange the text we can define **YYVAL**.

The general process to parse a code is:

- 1) Run lex
- 2) Run yacc
- 3) Compile both yacc and lex source codes
- 4) Compile other modules
- 5) Link lex and yacc and other sources for operation.

3. WORKING MODEL / PRINCIPLE

We worked on learning the language lex and yacc for constructing our parser.

Lex is used for generating series tokens for further usage by yacc. Its basic job is to identify what are you reading. Now to generate the c source code run lex on the file which we want to parse and the lex files are stored by the extension of .l. The series of tokens are grouped in the grammar for usage by yacc.

We can also work with multiple word, phrases therefore the c source code is not always standalone. Instead of lex, flex is preferred as it has number of benefits like it improve memory and speed, as lex is always memory hungry. The final c source code can be used to test the files we want to parse.

Suppose if the tokens are not recognized, they are echoed back and thus not included in the grammar for yacc usage. We don't get the token names explicitly, rather we get them in the y.tab.h file which is automatically generated by yacc.

The general format of Lex source is:

```
{definitions}  
%%  
{rules}  
%%  
{user subroutines}
```

So, in the above given format we have to write first the definition in which if any prior you have to define. Then it comes to the rules that we specify. Generally, the rules we specify are in regular expression. So the format of specifying the rules are as follows:

We have a Table with two columns: regular expressions and actions which can be written as

```
integer {printf("found keyword INT");}
```

Here we specify a token as integer and then action that has to be performed when you encounter the token in your program. So for getting an idea of how to define regular expression, we already studied that in our course.

YACC:

The basic structure of the yacc grammar definition is to define the sequence of expected tokens. So first the yacc grammar structure looks something like this:

```
%{  
  
/* C includes */  
  
}%  
  
/* Other Declarations */  
  
%%  
  
/* Rules */  
  
%%  
  
/* user subroutines */
```

So in the given structure after defining the prior headers and c includes in the file, we try to define the rules in order to respond to the input program.

YACC Rules

_ A grammar rule has the following form:

A : BODY ;

_ A is a non-terminal name (LHS).

_ BODY consists of names, literals, and actions.

(RHS) _ *literals* are enclosed in quotes, eg: '+'

'nn' ! newline.

'n"' ! single quote.

So the rules can be specified by the action that has to be performed in the following format:

```
XXX : YYY ZZZ  
  
{ printf("a message\n"); }  
  
;
```


Here for giving more than one instruction for the same token we can use the symbol '|'.

Now let us see how we can communicate between a scanner and a parser and how to use yacc with that:

The user must supply an integer-valued function `yylex()` that implements the lexical analyzer (scanner). If there is a value associated with the token, it should be assigned to the external variable `yylval`. The token error is reserved for error handling.

_ *Token numbers* : These may be chosen by the user if desired.

The default is chosen by yacc [in a file `y.tab.h`], the token no. for a literal is its ASCII value. Then other tokens are assigned numbers starting at 257, the endmarker must have a number zero or negative. Then finally it generates `y.tab.h` using 'yacc -d'

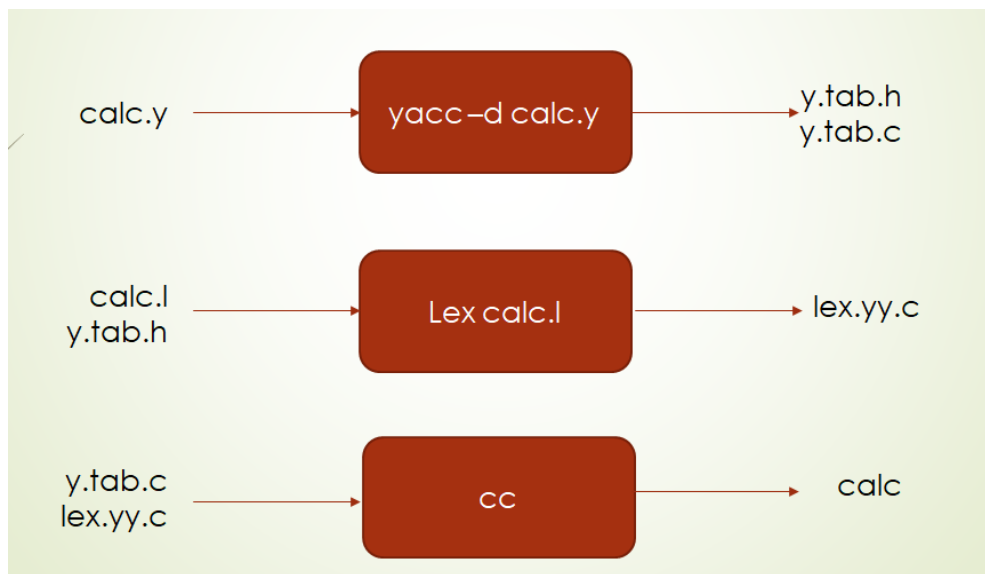
Using Yacc

Suppose the grammar spec is in a file `foo.y`. Then the command 'yacc `foo.y`' yields a file `y.tab.c` containing the parser constructed by yacc. The command 'yacc -d `foo.y`' constructs a file `y.tab.h` that can be `#include'd` into the scanner generated by lex.

The command 'yacc -v `foo.y`' additionally constructs a file `y.output` containing a description of the parser (useful for debugging). The user needs to supply a function `main()` to driver, and a function `yyerror()` that will be called by the parser if there is an error in the input.

4. STAGES OF IMPLEMENTATION:

- When we feed calc.y into YACC , it generates y.tab.h and y.tab.c .
- Then we feed calc.l and y.tab.h into lex. It generates lex.yy.c .
- Then we compile y.tab.c and lex.yy.c , which then generates a parser(calc in this case).



5. WORK DONE:

We studied HTML, Lex and YACC

We have created a basic lex program to detect basic HTML layout.

The code is as follows:

```
%%
"<html>"      {yylval.f = yytext;return html_begin_encountered;}
"<body>"      {yylval.f = yytext;return body_begin_encountered;}
"<p>"         {yylval.f = yytext;return para_begin_encountered;}
"<form>"      {yylval.f = yytext;return form_begin_encountered;}
"<table>"     {yylval.f = yytext;return table_begin_encountered;}
"</table>"    {yylval.f = yytext;return table_end_encountered;}
"</form>"     {yylval.f = yytext;return form_end_encountered;}
"</p>"        {yylval.f = yytext;return para_end_encountered;}
"</body>"     {yylval.f = yytext;return body_end_encountered;}
"</html>"     {yylval.f = yytext;return html_end_encountered;}

[ \t\n]      ;
.            ;
%%
```

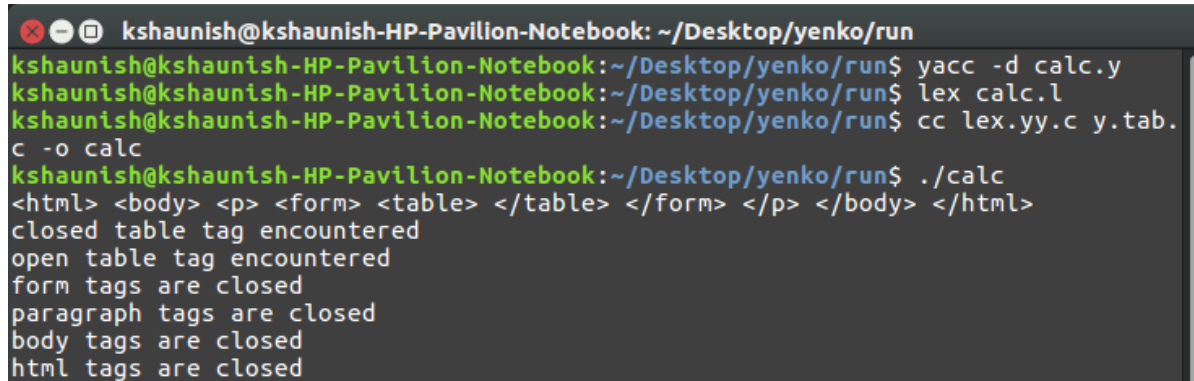
The following is the implementation of HTML language parser using YACC.

```
%%
S : H                                     {printf("%s\n", $1);}
  ;
H : html_begin_encountered B html_end_encountered
  {printf("html tags are closed\n"); $$ = $2;}
  ;
B : body_begin_encountered P body_end_encountered
  {printf("body tags are closed\n"); $$ = $2;}
  ;
P : para_begin_encountered F para_end_encountered
  {printf("paragraph tags are closed\n"); $$ = $2;}
  ;
F : form_begin_encountered T form_end_encountered
  {printf("form tags are closed\n"); $$ = $2;}
  ;
T : table_begin_encountered C           {printf("open table tag encountered\n"); $$ = $2;}
  ;
C : table_end_encountered               {printf("closed table tag encountered\n"); $$ = $1;}
  ;

%%
```

6. RESULT AND ANALYSIS:

The tags we have studied and considered for the code are <html>, <body>, <p>, <form> and <table>. We were successful in implementing these basic html tags and the code that we wrote was able to identify the tag mentioned and whether the given tag was open or closed.



```
kshaunish@kshaunish-HP-Pavilion-Notebook: ~/Desktop/yenko/run
kshaunish@kshaunish-HP-Pavilion-Notebook:~/Desktop/yenko/run$ yacc -d calc.y
kshaunish@kshaunish-HP-Pavilion-Notebook:~/Desktop/yenko/run$ lex calc.l
kshaunish@kshaunish-HP-Pavilion-Notebook:~/Desktop/yenko/run$ cc lex.yy.c y.tab.c -o calc
kshaunish@kshaunish-HP-Pavilion-Notebook:~/Desktop/yenko/run$ ./calc
<html> <body> <p> <form> <table> </table> </form> </p> </body> </html>
closed table tag encountered
open table tag encountered
form tags are closed
paragraph tags are closed
body tags are closed
html tags are closed
```

7. REFERENCES

<http://dinosaur.compilertools.net/> <http://ecomputernotes.com/compiler-design/compiler-construction-tools>
<http://whatis.techtarget.com/definition/yacc-yet-another-compiler-compiler> <https://www.youtube.com/watch?v=54bo1gaHAfk>

Lex & Yacc by John R. Levine Tony Mason Doug Brown O'Reilly & Associates, Inc

Lex and Yacc: A Brisk Tutorial by Saumya K. Debray Department of Computer Science The University of Arizona Tucson, AZ 85721