by Jeffreys Copeland and Haemer

JANE MARINSKY

The purpose of a programming system is to make a computer easy to use. To do this, it furnishes languages and various facilities that are in fact programs invoked and controlled by language features. But these facilities are bought at a price …
– Fred Brooks, *The Mythical Man Month*

There was an Old
Man with a beard,
Who said, "It is
just as I feared!
Two Owls and a Hen,
Four Larks and a Wren,
Have all built their
nests in my beard!"
– Edward Lear,
Book of Nonsense

The Nesting Instinct

Weary of beating ourselves bloody against the rock of "Expect scripts in Perl," we decided we needed to try something easier and less stressful.

Our boss had asked us to look into XML, and writing a little XML-related code seemed like a good place to start. (We'll warn you right off that this isn't an XML column; still, we'll need to tell you a little about XML–a cousin to HTML–and we'll get to that in a second.)

A quick scan through the Web finds a forest of Perl modules to handle XML, such as `XML::Twig` and `XML::Grove`. This was encouraging, since we are comfortable in Perl and Perl is terse and powerful: we can write interesting Perl programs in few enough lines to fit in a 2,000-word column.

But wait. There are about 50 XML modules on the CPAN. Which one shall we use? O'Reilly and Associates' `xml.com` has an article, "Ways to Rome: Processing XML with Perl" that contrasts 11 different approaches to XML using nine different modules. We were not ready for that.

What to do? What to do?

"Perhaps," we thought, "a fresh look at things would help." A little free-association got us an answer.

One view of XML is that it's a language for writing nested text. Any time we hear "nested," we think "parsers" and "stack machines" and "pumping lemmas for context-free grammars." (Well, one Jeff does. The other one thinks about the hatching sparrow's egg he found on the sidewalk earlier this week.) "Maybe this," we thought,"is a job for *yacc*."

Careful readers have noticed that we didn't say free association got us a *good* idea. But what of that?

Besides, we had just installed RedHat 7.1 (on special for $19.95 at our local CompUSA), and writing a *yacc* parser for XML on a system where lots of things were no longer where we expected them to be seemed, oh, challenging. To round out the experience, we decided to switch from *vi* to *emacs*.

Thoughtful study of the Greek and Latin classics teaches that even watching people get torn apart by lions can have entertainment value as long as you have good seats.

So relax, don't worry, have a home brew, read on.

XML is a Good Idea

XML looks like this:

```
<?xml version="1.0"
  standalone="yes"?>
<order>
 <customer>
   <name>Coyote, Ltd.</name>
   <shipping_info>
 <address>1313 Desert
Road</address>
 <city>Flagstaff</city>
 <state>AZ</state>
 <zip>90210</zip>
   </shipping_info>
 </customer>
 <item>
   <product id="1111">Acme
```

```
Rocket Jet Pack</product>
   <quantity type="each">1</quantity>
 </item>
 <item>
   <product id="2222">Roadrunner Chow</product>
   <quantity type="bag">10</quantity>
 </item>
</order>
```

It's data dressed up to look like HTML.

Notice a couple of things. For one, it's structured, but familiar. Orders have a customer followed by items. Customers have names and shipping information. Shipping information has a street address, city, state, and zip. And so on.

The syntax for beginning and ending tags that everyone is now used to from HTML, `<some_tag>` and `</some_tag>`, are borrowed and applied to any tag you want to make up.

Actually, both HTML and XML steal this syntax from their parent, SGML (Standard Generalized Mark-up Language), which gave everyone a way to mark up text that was so generalized that no one would use it.

HTML provided a very simplified subset of SGML and proved that something SGML-like could succeed. But if you've ever wanted to define your own tags in HTML–and, lets face it, we all have–you'll know that HTML can sometimes be too simplified. Enter XML, a simplified version of SGML that still lets you define your own tags.

Second, it's text, and there's a public spec. There's no shortage of structured document formats. Microsoft Word documents are structured in some way. So are Excel spreadsheets.

The next time you try to write a shell script to parse one of these to extract and reformat information needed for another document, drop us a line to tell us how much fun you're having. We need the laugh.

XML files, in contrast, are human-readable. For documents alone, this is a great idea, but it's an idea in an arena where there are already a lot of mature, open, text-based markup languages, such as *troff*, TeX, and HTML.

What transforms XML into something interesting is turning simple text markup on its head by realizing that XML tags can be used to attach meanings to any kind of nested data.

For example?

Well, spreadsheets, which we mentioned earlier, are one. The Gnome spreadsheet, *gnumeric*, uses XML as its native-file format. Run *gnumeric*, type a few things in, save the spreadsheet, then gunzip it (the files are gzipped to save space) and look at it using whatever you use to peruse ASCII files: *cat*, *less*, *vi*, whatever.

Glade, Gnome's user interface builder, spits out interface description files in XML.

For that matter, any data structures you want to pass around can be flattened out with XML, written to a file, and then read in by anything else that understands the same tags. SOAP and XML-RPC, for example, are RPC (remote-procedure-call) mechanisms that pass arguments and return values over the Web as XML-formatted data. WebDAV (Distributed Authoring and Versioning)–Wiki-Wiki Web filtered through too many committees, sort of–passes its data around as XML, too.

And, no matter how intricately structured, no matter how complex the semantics, XML is always something you can read. With *cat*, with *less*, with *vi*. (OK, even with *emacs*.) You can hunt for stuff in it with *grep*. You can print it out on your printer.

It's just text.

Validating XML

What else can we do with XML? We can parse it.

Why is that interesting? Because syntactically valid data, like syntactically valid programs, are a long way towards being correct.

Dimidium facti, qui coepit, habet. –Horace

Well begun is half done.

We find that using `perl -c -w -Mstrict` on our Perl programs goes a long way towards making them work.

When developing data formats as complex as the output of *gnumeric*, having a tool verify that data have the correct structure is also a big help.

Moreover, when data are known to be valid, you don't have to scatter data validation through your programs. No need to write *N* separate sets of routines (with *N* separate sets of bugs) for *N* separate application, to check that each customer has a city, state, and zip code if there is a central data-validator or a single data-validation library that anyone can call.

There are a lot of articles on XML validation, but we particularly like Kip Hampton's "Simple XML Validation in Perl," (`http://www.xml.com/pub/a/2000/11/08/perl`), because it uses Joshua Nathaniel Pritikin's `Test.pm`, which we wrote about a few months back, ("The Art of SoftwareTesting," `http://swexpert.com/C9/SE.C9.AUG.00.pdf`, and "Testy, Aren't We?", `http://swexpert.com/C9/SE.C9.SEP.00.pdf`).

Better still, it uses the module in a way we suspect its author never intended: as part of a stand-alone application.

Rolling Our Own

As we mentioned at the outset, we wanted to see if we understood the concepts, so we decided to build our own XML validator. Since we were impressed by Hampton's paper, we started with his data. These are the customer-order data we showed earlier, available for downloading from O'Reilly's `xml.com` Web site.

And, so as to not ape his approach (or that of any sane person), we thought it might be fun to validate XML the same way we validate Perl: with a parser. To build our parser, we used the universally-available parser-generator, *yacc*.

Before we show you what we ended up with, let's digress for a minute to review some basic Computer Science.

Lack and Yecchs

A compiler's job is to take a program we feed it, tease apart the text to see what it's supposed to do, and then generate machine instructions that correspond to our source.

The "tease apart" section is done, conceptually, in two steps: tokenizing and parsing.

The first, also called "lexical analysis," just breaks the source up into words. In the code fragment

```
if (year==2001) { riley=16 }
```

the tokens are `if`, `(`, `year`, `==`, `2001`, `)`, `{`, `riley`, `=`, `16`, and `}`. Lexical analysis actually can be done by the program

```
perl -pe 's/$RE/\n/'
```

where `RE` is some tedious, incredibly complex, Perl regular expression. You can get from the grammar of the language to the correct Perl regular expression by buying enough beer for some nearby Automata Theorist.

The second step, parsing, verifies that the tokens are in the right order. In the code fragment

```
} { year riley if = == 2001 16 ) (
```

the tokens are the same, but they aren't arranged into syntactically valid C. This step actually can't be done by any tedious Perl regular expression, no matter how complex it is or how much beer you're willing to buy.

The distinction here is that regular expressions are a way to describe, in a single string, a kind of "machine" called a *finite automaton*, and you can prove that finite automata are not able to do paren matching, or brace matching, or anything that requires matching the beginnings of nested structures with their ends. Such jobs are handled by more powerful machines called *parsers*. Parsers can also tokenize, but that's using a hand grenade for a hand-ax.

Early on, all tokenizers and parsers were hand-crafted, but that was tedious. Before long, at Bell Labs, Mike Lesk and E. Schmidt wrote *lex*, a program that writes tokenizers, and Steve Johnson wrote *yacc*, "yet another compiler-compiler," which writes parsers.

Lex takes an input file full of regular expressions and writes C that compiles into a tokenizer. *Yacc* takes an input file full of rules that look sort of like a BNF grammar and writes C that compiles into a parser. (If you don't know what a BNF grammar is–it stands for "Backus-Naur Form," after its inventors–it'll be obvious from the example below.)

These or their GNU replacements, *flex* and *bison*, have been distributed with UNIX systems since the mid-to-late 1970s.

Rubber and Road

By now, you've undoubtedly noticed that validating an XML file requires two steps: breaking it apart into valid tokens, like `</address>`, and ensuring they're correctly nested and in the proper order. This, then, is what brought *lex* and *yacc* to mind.

Writing *yacc* grammars and *lex* source files is straightforward, so without further ado, here's our *lex* input file.

```
/* $Id: lexer.l,v 1.8 2001/06/02 22:10:45 jsh Exp $ */
%{
#include "y.tab.h"
%}

WS   [ \t\n\r]+
OTHER     [^<>]+

%%

{WS}         /* eat white space */
\<\?xml[^>]*\?> return XML;
\<order> return ORDER;
\<\/order>   return _ORDER;
\<customer>      return CUSTOMER;
\<\/customer>     return _CUSTOMER;
\<name>       return NAME;
\<\/name>         return _NAME;
\<shipping_info>   return SHIPPING_INFO;
\<\/shipping_info>     return _SHIPPING_INFO;
\<address>    return ADDRESS;
\<\/address> return _ADDRESS;
\<item>       return ITEM;
\<\/item>         return _ITEM;
\<city>       return CITY;
\<\/city>         return _CITY;
\<state>      return STATE;
\<\/state>   return _STATE;
\<zip>        return ZIP;
\<\/zip>          return _ZIP;
\<product[^>]*> return PRODUCT;
\<\/product> return _PRODUCT;
\<quantity[^>]*> return QUANTITY;
\<\/quantity>     return _QUANTITY;
<<EOF>>        return 0;
{OTHER}      return STRING;

%%
```

We won't drag you through the lexical dirt, but you can see that the file has a rule for each kind of token that needs to be recognized, including a rule to throw away white space:

```
{WS}            /* eat white space */
```

which refers to an earlier rule that defines white space precisely, as one or more white-space characters (blanks, tabs, newlines, or carriage returns) in a row:

```
WS       [ \t\n\r]+
```

Next, we trot out our `yacc` input file:

```
/* $Id: parser.y,v 1.9 2001/06/02 22:24:03 jsh Exp $ */

%token ORDER _ORDER CUSTOMER _CUSTOMER NAME _NAME
%token SHIPPING_INFO _SHIPPING_INFO
%token ADDRESS _ADDRESS
%token CITY _CITY STATE _STATE ZIP _ZIP
%token ITEM _ITEM PRODUCT _PRODUCT
```

```
%token QUANTITY _QUANTITY
%token STRING XML

%% /* Grammar rules and actions follow */

input:    /* empty */
  | input xml order
  ;
xml:      XML ;
order:          ORDER customer items _ORDER ;
customer: CUSTOMER name shipping_info _CUSTOMER ;
name:           NAME string _NAME ;
shipping_info: SHIPPING_INFO address city state
zip _SHIPPING_INFO ;
address:  ADDRESS string _ADDRESS ;
city:           CITY string _CITY ;
state:          STATE string _STATE ;
zip:      ZIP string _ZIP ;
items:          item
    | items item
    ;
item:           ITEM product quantity _ITEM
    | ITEM quantity product _ITEM
    ;
product:  PRODUCT string _PRODUCT ;
quantity: QUANTITY string _QUANTITY ;
string:         STRING ;

%%

/* Lexical analyzer returns a double floating point
number on the stack and the token NUM, or the
ASCII character read if not a number.  Skips all
blanks and tabs, returns 0 for EOF. */

#include <stdio.h>
#include <ctype.h>

main ()
{
 yyparse ();
}


#include <stdio.h>

yyerror (s)  /* Called by yyparse on error */
  char *s;
{
 printf ("%s\n", s);
}
```

And, looking back at Kip's code, which do we like better? His.

Reflections

So what did we learn from this exercise?

Well, first, figuring out how to bend *lex* and *yacc* to our will was a lot of work. Unlike Perl, *lex* and *yacc* are tools that we rarely use. But, as with Perl, wielding them efficiently requires practice.

Perl also has modules, syntax-checking, and a debugger, all of which ease development. If we were writing a lot of parsers, or one really big one, we'd probably use *yacc*, but we'd write tools to make our work easier. Normally, each grammar rule would have an action associated with it, so the parser could generate code. We would build C++ classes so the generation of code from disparate actions could be insulated.

Second, we think that Kip's Perl example is clearer and easier to modify. Some of this is for reasons we already mentioned, but some of it is that the modules he uses are tailored to the job of parsing XML. Like SGML, *lex* and *yacc* are general tools, with more power than this job needs. When we see that much power walking down the road, verbosity and syntactic complexity are often walking on either side.

Third, we are impressed, in retrospect, by a few tools that seem to have clean-burning, easy-to-use power. Regular expressions are a great example. To underscore this, can anyone suggest a clean, simple syntax that would let a Perl programmer specify context-free grammars as easily as we can specify regexes?

Fourth, we offer a tiny piece of psychological insight: to wit, when we finished writing our parser and lexer, we knew we were done because feeding our test case to the validator just gave us back a prompt and nothing else. This is, we discovered, mildly depressing. Wedded for hours to this little project, in the end we'd brought together something old (*yacc* and *lex*), something new (XML) and something borrowed (Kip's example), and wound up blue.

Fortunately, there's plenty of fun to be had around every corner, and we're certain you'll spend the next month having some of it.

But do we now feel more comfortable with XML? Well … yeah, actually, we do.

So, until next time, happy trails.

PS: This column was written on June 2, 2001. On this occasion, the Jeffs wish JSH's alienated first-born, Riley Sherman Haemer, a happy birthday. Though she is lost forever to the Boulder, CO, divorce courts, we both love her and will forever miss her more than words or tears will tell.

Happy trails, little Riley.

Jeffrey Copeland *(*copeland@alumni.caltech.edu*) is currently living in the Pacific Northwest, where he spends his time writing UNIX software in a large development organization and fighting damp rot.*

Jeffrey S. Haemer *(*jsh@usenix.org*) works at Minolta-QMS Inc. in Boulder, CO, building laser printer firmware. Before he worked for QMS, he operated his own consulting firm and did a lot of other things, much like everyone else in the software industry.*

Note: The software from this and past Work columns is available at `http://alumni.caltech.edu/~copeland/work` *or alternately at* `ftp://ftp.cpg.com/pub/Work`.