Build code with lex and yacc, Part 1: Introduction

Meet lex, yacc, flex, and bison

Peter Seebach August 11, 2004

Lex and yacc are tools to automatically build C code suitable for parsing things in simple languages. These tools are most often used for parts of compilers or interpreters, or for reading configuration files. In the first of two articles, Peter Seebach explains what lex and yacc actually do and shows how to use them for simple tasks.

Most people never need to know what lex and yacc do. You occasionally need to have them installed to compile something you downloaded, but for the most part, it just works. Maybe an occasional README file refers to "shift/reduce" conflicts. However, these tools remain a valuable part of the Unix toolbox, and a bit of familiarity with them will go a long way.

In fact, while lex and yacc are almost always mentioned in the same breath, they can be used separately. A number of very entertaining if trivial programs have been written entirely in lex (see Related topics for links to those). Programs using yacc, but not lex, are rarer.

Throughout these articles, the names "lex" and "yacc" will be used to include also flex and bison, the GNU versions of these utilities. The code provided should work in all major versions, such as MKS yacc. It's all one big happily family!

This is a two-part series. The first article will introduce lex and yacc in more general terms, exploring what they do and how they do it. The second shows an actual application built using them.

What are lex and yacc, and why do people refer to them together?

Lex and yacc are a matched pair of tools. Lex breaks down files into sets of "tokens," roughly analogous to words. Yacc takes sets of tokens and assembles them into higher-level constructs, analogous to sentences. Yacc is designed to work with the output of Lex, although you can write your own code to fill that gap. Likewise, lex's output is mostly designed to be fed into some kind of parser.

They're for reading files that have reasonably structured formats. For instance, code in many programming languages can be read using lex and yacc. Many data files have predictable enough

formats to be read using them, as well. Lex and yacc can be used to parse fairly simple and regular grammars. Natural languages are beyond their scope, but most computer programming languages are within their bounds.

Lex and yacc are tools for building programs. Their output is itself code, which needs to be fed into a compiler; typically, additional user code is added to use the code generated by lex and/or yacc. Some simple programs can get by on almost no additional code; others use a parser as a tiny portion of a much larger and more complicated program.

A more detailed look at each of these programs is in order.

Lex: A lexical analyzer generator

A lexical analyzer isn't a handheld gizmo you get on a sci-fi show. It's a program that breaks input into recognized pieces. For instance, a simple lexical analyzer might count the words in its input. Lex takes a specification file and builds a corresponding lexical analyzer, coded in C.

Perhaps the best way to approach this is to look at an example. Here's a simple lex program, taken from the man page for flex.

Listing 1. Counting words using lex

```
int num_lines = 0, num_chars = 0;

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main() {
 yylex();
 printf("# of lines = %d, # of chars = %d\n", num_lines, num_chars);
}
```

This program has three sections, separated by `%%` markers. The first and last sections are plain old C code. The middle section is the interesting one. It consists of a series of rules that lex translates into the lexical analyzer. Each rule, in turn, consists of a regular expression and some code to be run when that regular expression is matched. Any text that isn't matched is simply copied to standard output. So, if your program is trying to parse a language, it's important to make sure that all possible inputs are caught by the lexer; otherwise, you get whatever's left over displayed to the user as though it were a message.

In fact, the code in Listing 1. is a complete program; if you run it through lex, compile it, and run the result, it'll do exactly what it looks like it does.

In this case, it's very easy to see what happens. A newline always matches the first rule. Any other character matches the second. Lex tries each rule in order, matching the longest stream of input it can. If something doesn't match any rules at all, lex will just copy it to standard output; this behavior is often undesirable. A simple solution is to add a last rule that matches anything, and either does nothing (if you're lazy) or emits some kind of diagnostic. Note that lex gives preference to longer matches, even if they're later. So, given these rules:

```
u { printf("You!\n"); }
uu { printf("W!\n"); }
```

and "uuu" for input, Lex will match the second rule first, consuming the first two letters, then the first rule. However, if something can match either of two rules, order in the lex specification determines which one is used. Some versions of lex may warn you when a rule cannot be matched.

What makes lex useful and interesting is that it can handle much more complicated rules. For instance, a rule to recognize a C identifier might look like

```
[a-zA-Z_][0-9a-zA-Z_]* { return IDENTIFIER; }
```

The syntax used is plain old regular expression syntax. There are a couple of extensions. One is that you can give names to common patterns. In the first section of the lex program, before the first `%%`, you can define names for a few of these:

```
DIGIT [0-9]
ALPHA [a-zA-Z]
ALNUM [0-9a-zA-Z]
IDENT [0-9a-zA-Z_]
```

Then, you can refer back to them by putting their names in braces in the rules section:

```
({ALPHA}|_){IDENT}* { return IDENTIFIER; }
```

Each rule has corresponding code that is executed when the regular expression is matched. The code can do any necessary processing and optionally return a value. The value will be used if a parser is using lex's output. For instance, in the case of the simple line-counting program, there's no real need for a parser. If you're using a parser to interpret code in a new language, you will need to return something to the parser to tell it what tokens you've found. You can just use an `enum` or a series of `#define` directives in a shared include file, or you can have yacc generate a list of predefined values for you.

Lex, by default, reads from standard input. You can point it at another file quite easily; it's a bit harder to, for instance, read from a buffer. There is no completely standardized way to do this; the most portable solution is to open a temporary file, write data to it, and hand it to the lexer. Here's a sample bit of code to do this:

Listing 2. Handing a memory buffer to the lexer

```
int
doparse(char *s) {
        char buf[15];
        int fd;
        if (!s) {
                return 0;
        }
        strcpy(buf, "/tmp/lex.XXXXX");
        fd = mkstemp(buf);
```

```
        if (fd < 0) {
                fprintf(stderr, "couldn't create temporary file: %s\n",
                        strerror(errno));
                return 0;
        }
        unlink(buf);
        write(fd, s, strlen(s));
        lseek(fd, 0, SEEK_SET);
        yyin = fdopen(fd, "r");
        yylex();
 fclose(yyin);
}
```

This code carefully unlinks the temporary file, leaving it open but already removed, to clean up automatically. A more careful programmer, or one not writing for an audience with limited space for sample code, might wish to consider the implications of the user's choice of `TMPDIR`.

yacc: yet another compiler compiler

So, you've broken your input into a stream of tokens. Now you need some way to recognize higher-level patterns. This is where yacc comes in: yacc lets you describe what you want to do with tokens. A yacc grammar tends to look sort of like this:

Listing 3. A simple yacc grammar

```
value:
    VARIABLE
  | NUMBER

expression:
    value '+' value
  | value '-' value
```

This means that an expression can take any of several forms; for instance, a variable, a plus sign, and a number could be an expression. The pipe character (`|`) indicates alternatives. The symbols produced by the lexer are called *terminals* or *tokens.* The things assembled from them are called *non-terminals*. So, in this example, `NUMBER` is a terminal; the lexer is producing this result. By contrast, `value` is a non-terminal, which is created only by assembling it from terminals.

Yacc files, like lex files, come in sections separated by `%%` markers. As with a lex file, a yacc file comes in three sections, the last of which is optional, and contains just plain C code to be incorporated in the generated file.

Yacc can recognize patterns of tokens; for instance, as in the example above, it can recognize that an expression can consist of a value, either a plus or minus sign, and another value. It can also take actions; blocks of code enclosed in `{}` pairs will be executed when the parser reaches that point in an expression. For instance, one might write:

```
expression:
value '+' value { printf("Matched a '+' expression.\n"); }
```

The first section of a yacc file defines the objects that the parser will manipulate and generate. In some cases, it could be empty, but most often, it will contain at least a few `%token` directives.

These directives are used to define the tokens the lexer can return. When yacc is run with the `-d` option, it generates a header file defining constants. Thus, our earlier lex example using:

```
({ALPHA}|_){IDENT}* { return IDENTIFIER; }
```

might have a corresponding yacc grammar containing the line:

```
%token IDENTIFIER
```

Yacc would create a header file (called `y.tab.h` by default) containing a line something like:

```
#define IDENTIFIER 257
```

These numbers will be outside the range of possible valid characters; thus, the lexer can return individual characters as themselves, or special tokens using these defined values. This can create a problem when porting code from one system to another: generally, you are best off re-running lex and yacc on a new platform, rather than porting the generated code.

By default, the yacc-generated parser will start by trying to parse an instance of the first rule found in the rules section of the file. You can change this by specifying a different rule with `%start`, but most often it's more reasonable to put the top-level rule at the top of the section.

Tokens, types, and values, oh my!

The next question is how to do anything with the components of an expression. The general way this is done is to define a data type to contain objects that yacc will be manipulating. This data type is a C `union` object, and is declared in the first section of a yacc file using a `%union` declaration. When tokens are defined, they can have a type specified. For a toy programming language, for instance, you might do something like this:

Listing 4. A minimalist %union declaration

```
%union {
 long value;
}
%token <value> NUMBER
%type <value> expression
```

This indicates that, whenever the parser gets a `NUMBER` token returned by the lexer, it can expect that the member named `value` of the global variable `yylval` has been filled in with a meaningful value. Of course, your lexer has to handle this in some way:

Listing 5. Making a lexer use yylval

```
[0-9]+ {
  yylval.value = strtol(yytext, 0, 10);
  return NUMBER;
 }
```

Yacc allows you to refer to the components of an expression by symbolic names. When a non-terminal is being parsed, the components that went into it are named `$1`, `$2`, and so on; the value it will pass back up to a higher-level parser is called `$$`. For instance:

Listing 6. Using variables in yacc

```
expression:
 NUMBER '+' NUMBER { $$ = $1 + $3; }
```

Note that the literal plus sign is `$2` and has no meaningful value, but it still takes up a place. There's no need to specify a "return" or anything else: just assign to the magic name `$$`. The `%type` declaration specifies that an `expression` non-terminal also uses the `value` member of the union.

In some cases, it may be useful to have multiple types of objects in the `%union` declaration; at this point, you have to make sure that the types you declare in `%type` and `%token` declarations are the ones you really use. For instance, if you have a `%union` declaration that includes both integer and pointer values, and you declare a token to use the pointer value, but your lexer fills in the integer value... Bad Things can happen.

Of course, this doesn't solve one last problem: the starting non-terminal expression has nowhere to return to, so you will need to do something with the values it produces. One way to do this is to make sure that you do all the work you need to do as you go; another is to build a single giant object (say, a linked list of items) and assign a pointer to it into a global variable at the end of the starting rule. So, for instance, if you were to build the above expression parser into a general-purpose calculator, just writing the parser would leave you with a program that very carefully parsed expressions, then did nothing at all with them. This is academically very interesting and it has a certain aesthetic appeal, but it's not particularly useful.

This basic introduction should leave you qualified to fool around a little with lex and yacc on your own time; perhaps building a simple calculator that *does* do something with the expressions it parses. One thing you'll find if you play around a bit is that lex and yacc can be confusing to troubleshoot. In the next installment, we'll look at troubleshooting techniques; as well, we will build a larger and more powerful parser for a real-world task.

Related topics

- Part 2 of this series covers a couple of concrete applications of lex and yacc, showing how to build a simple calculator and a program to parse e-mail messages.
- The Lex & Yacc Page has a number of interesting historical references, as well as very good lex and yacc documentation.
- The GNU versions of lex and yacc are flex and bison and, as with all things GNU, have excellent documentation including complete user manuals in a variety of formats.