What are lex and yacc?

Lex and yacc are two programs usually mentioned in the same breath that implement Look-Ahead-Left-Right (LALR -- i.e. not "recursive descent") parsing of non-ambiguous context-free (i.e. not "natural language") grammars. But if you haven't taken a class in techno-babble and/or parsing you probably have no idea what that sentence means, so in English: they provide an automated way to read input files according to any file format you specify. They will automatically die on syntax errors (that's good) and they are very very fast.

Why should you learn a new their pattern syntax when you could just write your own parser? Because they are generalized -- what they do is actually *compile* your pattern into a state machine in C code, which means that it's very fast and very type safe. But I'm not going to lie to you: their syntax is not exactly easy to learn, and even then they do (as one person said about computers in general) "what you *tell* them to do, not what you *want* them to do."

Lex and yacc are open-source freeware, and the official GNU versions are called "flex" and "bison". (They are almost, but not quite, completely interchangeable.)

This webpage is supposed to be a tutorial for complete novices needing to use lex and yacc for some real project. I wrote it as I was learning them, so hopefully it explains things right as you need to know them instead of flooding you with theory and details.

Let's create a lex/yacc parser for a silly format that I'll call "snazzle". Snazzle files have four general sections: a header that just says "sNaZZle 1.0" with version info, a template section that defines types like "type foo" and "type bas", a body section with actual data consisting of 4 numbers and one of the types, and a footer that just says "end". (I don't have a particular file format or program in mind, I'm just coming up with this.) Here's an example of our silly format:

```
sNaZZle 1.3
type foo
type bar
type bas
0 0 10 5        foo
20 10 30 20     foo
7 8 12 13       bas
78 124 100 256  bar
end
```

For making parsers, you want to start at the high level because that's how yacc thinks about them. We start by saying that a "snazzlefile" is comprised of four general parts: "header", "template", "body", and "footer". Then we describe what comprises each of those parts, and what comprises each of the those parts, and so on until we get all the way down what are called *terminal symbols*. Terminal symbols are the smallest units of your grammar -- for our example here, integers like "15" are one of our terminal symbols, and pointedly not the individual characters "1" and "5" that make it up. Terminal symbols, by the way, are the boundary between lex and yacc: lex sees the individual "1" and "5" characters but combines them into "15" before giving it to yacc.

Lex (or flex)

For our silly snazzle files, we'll make 3 terminal symbol types: a NUMBER that's basically an integer, a FLOAT that's a floating-point number that we only need for the version, and a STRING which is pretty much everything else. Furthermore, since this is a tutorial, let's show off lex's power by making a program that just lexs our input without using yacc to parse it. Here's the first whack:

snazzle.lex:

```
%{
#include <iostream>
%}
%%
[ \t] ;
[0-9]+\.[0-9]+ { cout << "Found a floating-point number:" << yytext << endl; }
[0-9]+  { cout << "Found an integer:" << yytext << endl; }
[a-zA-Z0-9]+ { cout << "Found a string: " << yytext << endl; }
%%
main() {
```

```
        // lex through the input:
        yylex();
}
```

First, you should know that lex (and yacc) files have three sections. The first is sort of "control" information, the second is the actual token or grammar rule definitions, and the last is C code to be copied verbatim to the output. (The output of both lex and yacc is C code!) These sections are divided by "%%", which you see on lines 4 and 9. Let's go through this line by line.

- Lines 1 through 3 are more C code to be copied. In the control sections (both lex and yacc), you can indicate C code to be copied to the output by enclosing it with "%{" and "%}". Notice that we don't have anything else in our control section -- the yacc control section gets a lot more usage than the lex one. And we wouldn't need one at all if we didn't use **cout** in the middle section.
- Line 4 is "%%", which means we're done with the control section and moving on to the token section.
- Lines 5-8 are all the same (simple) format: they define a regular expression (a topic in its own right, which I touch on not at all exhaustively at the end of this page) and an action. When lex is reading through an input file and can match one of the *regular expressions*, it executes the *action*. The regular expression is not Perl's idea of a regular expression, so you can't use "\d", but the normal stuff is all available. The *action* is just C++ code that is copied into the eventual lex output; accordingly, you can have a single statement or you can have curly braces with a whole bunch of statements.
- Line 9 is another "%%" delimiter, meaning we're done with the second section and we can go onto the third.
- Lines 10-13 are the third section, which (in both lex and yacc) is exclusively for copied C code. (Notice that, unlike the control section at the top, there is no "%{" or "%}".) We don't normally need to put anything in this section for the lex file, but for this example we're going to put a main() function in here so that we can execute the output without having to make another file and compile them together.

This example can be compiled by running this:

```
% lex snazzle.lex
```

This will produce the file "lex.yy.c", which we can then compile with g++:

```
% g++ lex.yy.c -lfl -o snazzle
```

Notice the "-lfl", which links in the dynamic lex libraries (actually *flex* libraries, hence the "fl"). On some systems you might have to use "-ll" instead. If that doesn't work, ask someone who'll know where on your system the lex/flex libraries are kept.

If all went as planned, you should be able to run it and enter stuff on STDIN to be lexed:

```
% ./snazzle
90
Found an integer:90
23.4
Found a floating-point number:23.4
4 5 6
Found an integer:4
Found an integer:5
Found an integer:6
this is text!
Found a string: this
Found a string: is
Found a string: text
!
```

Which is pretty cool! Notice that the exclamation mark at the very end was just echoed: when lex finds something that doesn't match any of the regexs it echos it to STDOUT. Usually this is a good indication that your token definitions aren't complete enough, but to get rid of it for now you could just add ". ;" to the token section, which will match anything (the "." of regex nomenclature) and do nothing with it (the empty C statement ";").

Now for a little more detail on the syntax for the middle section. In general, it is really as simple as "a regex match" followed by "what to do with it". The "what to do with it" can vary a lot though - for example, most parsers completely ignore whitespace, which you can do by making the "action" just a semicolon:

```
  [ \t] ;  // ignore all whitespace
```

Most parsers also want to keep track of the line number, which you would do by catching all the carriage returns and incrementing a line counter. However, you want the carriage return itself to be ignored as if it were whitespace, so even though you have an action performed here you want to *not* put a `return` statement in it:

```
  \n    { ++linenum; }  // increment line count, but don't return a token
```

And pretty much everything else will need to return a token to yacc, but more on that later.

Reading from STDIN is kind of annoying though, when you'd really like to pick a file to read from. lex reads its input from a global pointer to C-style FILE variable called **yyin**, which is set to STDIN by default. All you have to do is set that pointer to a C-style FILE handle that you've opened yourself, and it'll read from it instead. That changes our example to this (the differences between this and the previous are in bold):

snazzle.lex:

```
%{
#include <iostream>
%}
%%
[ \t] ;
[0-9]+\.[0-9]+ { cout << "Found a floating-point number:" << yytext << endl; }
[0-9]+  { cout << "Found an integer:" << yytext << endl; }
[a-zA-Z0-9]+ { cout << "Found a string: " << yytext << endl; }
. ;
%%
main() {

        // open a file handle to a particular file:
        FILE *myfile = fopen("a.snazzle.file", "r");
        // make sure it's valid:
        if (!myfile) {
                cout << "I can't open a.snazzle.file!" << endl;
                return -1;
        }
        // set lex to read from it instead of defaulting to STDIN:
        yyin = myfile;

        // lex through the input:
        yylex();
}
```

Which you can then run just like you did above, assuming you have "a.snazzle.file":

```
% lex snazzle.lex
% g++ lex.yy.c -lfl -o snazzle
% cat a.snazzle.file
90
23.4
4 5 6
this is text!
% ./snazzle
Found an integer:90
Found a floating-point number:23.4
Found an integer:4
Found an integer:5
Found an integer:6
Found a string: this
Found a string: is
Found a string: text
%
```

But now we've run out of steam without moving on to:

Yacc (or bison)

The first thing we have to do in our silly parser is to start defining the terminal tokens, which for us are ints, floats, and strings. This is done (strangely enough) in the yacc file. We're also going to move main() into the copied-C-code section of the yacc file now:

snazzle.y:

```
%{
#include <iostream>
%}

// define the "terminal symbol" token types I'm going to use, which
// are in CAPS only because it's a convention:
%token INT FLOAT STRING

// yacc fundamentally works by asking lex to get the next token, which it returns as
// an object of type "yystype".  But
// the next token could be of an arbitrary data type, so you can define a C union to
// hold each of the types of tokens that lex could return, and yacc will typedef
// "yystype" as the union instead of its default (int):
%union {
        int ival;
        float fval;
        char *sval;
}

// and then you just associate each of the defined token types with one of
// the union fields and we're happy:
%token <ival> INT
%token <fval> FLOAT
%token <sval> STRING

%%
// this is the actual grammar that yacc will parse, but for right now it's just
// something silly to echo to the screen what yacc gets from lex.  We'll
// make a real one shortly:
snazzle:
        INT snazzle { cout << "yacc found an int: " << $1 << endl; }
        | FLOAT snazzle { cout << "yacc found a float: " << $1 << endl; }
        | STRING snazzle { cout << "yacc found a string: " << $1 << endl; }
        | INT { cout << "yacc found an int: " << $1 << endl; }
        | FLOAT { cout << "yacc found a float: " << $1 << endl; }
        | STRING { cout << "yacc found a string: " << $1 << endl; }
        ;
%%
#include <stdio.h>

// stuff from lex that yacc needs to know about:
extern int yylex();
extern int yyparse();
extern FILE *yyin;

main() {
        // open a file handle to a particular file:
        FILE *myfile = fopen("a.snazzle.file", "r");
        // make sure it is valid:
        if (!myfile) {
                cout << "I can't open a.snazzle.file!" << endl;
                return -1;
        }
        // set lex to read from it instead of defaulting to STDIN:
        yyin = myfile;

        // parse through the input until there is no more:
        do {
                yyparse();
        } while (!feof(yyin));

}

void yyerror(char *s) {
        cout << "EEK, parse error!  Message: " << s << endl;
        // might as well halt now:
        exit(-1);
}
```

Now that the yacc file knows what the terminal types are, we can use them in the lex file to make our life easier:

snazzle.lex:

```
%{
#include <iostream>
#include "y.tab.h"  // to get the token types that we return
%}
%%
[ \t] ;

[0-9]+\.[0-9]+ { yylval.fval = atof(yytext); return FLOAT; }
[0-9]+  { yylval.ival = atoi(yytext); return INT; }
[a-zA-Z0-9]+ {
        // we have to copy because we can't rely on yytext not changing underneath us:
        char *res = new char[strlen(yytext) + 1];
        strcpy(res, yytext);
        yylval.sval = res;
        return STRING;
}

. ;
%%
```

Since the lex file now has to #include "y.tab.h", we have to run yacc before lex (yacc generates y.tab.h if you say "-d" to yacc):

```
% yacc -d snazzle.y
% lex snazzle.lex
% g++ y.tab.c lex.yy.c -lfl -o snazzle
% ./snazzle
yacc found a string: text
yacc found a string: is
yacc found a string: this
yacc found an int: 6
yacc found an int: 5
yacc found an int: 4
yacc found a float: 23.4
yacc found an int: 90
%
```

You'll notice that the input is echoed out backward! This is because for each of the rules you define, yacc does not perform the *action* until it matches the complete *rule*. In the above example, all of the recursive rules were right-recursive (i.e. they looked like "foo: bar foo" instead of "foo: foo bar"). The right-recursive search will print the output backwards since it has to match EVERYTHING before it can start figuring out what's what, AND it has another major drawback: if your input is big enough, right-recursive rules will overflow yacc's internal stack! So a better implementation of the yacc input would be left-recursion:

snazzle.y:

```
%{
#include <iostream>
%}

// define the "terminal symbol" token types I'm going to use, which
// are in CAPS by convention:
%token INT FLOAT STRING

// yacc asks lex for the next token, which is returned as being of type "yystype".  But
// the next token could be of an arbitrary data type, so you can define a C union to
// hold each of the types of tokens that lex could return, and yacc will typedef
// "yystype" as the union instead of its default (int):
%union {
        int ival;
        float fval;
        char *sval;
}

// and then you just associate each of the defined token types with one of
// the union fields and we're happy:
%token <ival> INT
%token <fval> FLOAT
%token <sval> STRING
```

```
%%
// this is the actual grammar that yacc will parse, but for right now it's just
// something silly to echo to the screen what yacc gets from lex.  We'll
// make a real one shortly:
snazzle:
        snazzle INT { cout << "yacc found an int: " << $2 << endl; }
        | snazzle FLOAT { cout << "yacc found a float: " << $2 << endl; }
        | snazzle STRING { cout << "yacc found a string: " << $2 << endl; }
        | INT { cout << "yacc found an int: " << $1 << endl; }
        | FLOAT { cout << "yacc found a float: " << $1 << endl; }
        | STRING { cout << "yacc found a string: " << $1 << endl; }
        ;
%%
#include <stdio.h>

// stuff from lex that yacc needs to know about:
extern int yylex();
extern int yyparse();
extern FILE *yyin;

main() {
        // open a file handle to a particular file:
        FILE *myfile = fopen("a.snazzle.file", "r");
        // make sure it is valid:
        if (!myfile) {
                cout << "I can't open a.snazzle.file!" << endl;
                return -1;
        }
        // set lex to read from it instead of defaulting to STDIN:
        yyin = myfile;
        // parse through the input until there is no more:
        do {
                yyparse();
        } while (!feof(yyin));
}

void yyerror(char *s) {
        cout << "EEK, parse error!  Message: " << s << endl;
        // might as well halt now:
        exit(-1);
}
```

(Note that we also had to change the "$1" in the cout to a "$2" since the thing we wanted to print out is now the second element in the rule.) This gives us the output we would hope for:

```
% yacc -d snazzle.y
% g++ y.tab.c lex.yy.c -lfl -o snazzle
% ./snazzle
yacc found an int: 90
yacc found a float: 23.4
yacc found an int: 4
yacc found an int: 5
yacc found an int: 6
yacc found a string: this
yacc found a string: is
yacc found a string: text
%
```

Now that the groundwork is completed, we can finally define the real file format in the yacc file. First we make what I call single-shot tokens that are used to represent constant strings: SNAZZLE, TYPE, and END, respective representing the strings "sNaZZle", "type", and "end". Then we flesh out the actual rules in the yacc grammar, and end up with this beautiful object d'art:

snazzle.lex:

```
%{
#include <iostream>
#include "y.tab.h"
%}
%%
[ \t] ;
sNaZZle { return SNAZZLE; }
```

```
type { return TYPE; }
end { return END; }
[0-9]+\.[0-9]+ { yylval.fval = atof(yytext); return FLOAT; }
[0-9]+  { yylval.ival = atoi(yytext); return INT; }
[a-zA-Z0-9]+ {
        // we have to copy because we can't rely on yytext not changing underneath us:
        char *res = new char[strlen(yytext) + 1];
        strcpy(res, yytext);
        yylval.sval = res;
        return STRING;
}
. ;
%%
```

snazzle.y:

```
%{
#include <iostream>
%}

// define the token types I can have:
%token INT FLOAT STRING
%token SNAZZLE TYPE
%token END
// since there's a different C datatype for each of the token type, yacc needs a
// union for the lex return value:
%union {
        int ival;
        float fval;
        char *sval;
}
// and then you just associate one of the defined token types with one of
// the union fields and we're happy:
%token  INT
%token  FLOAT
%token  STRING
%%

// the first rule defined is the highest-level rule, which in our
// case is just the concept of a whole "snazzle file":
snazzle:
        header template body_section footer { cout << "done with a snazzle file!" << endl; }
        ;
header:
        SNAZZLE FLOAT { cout << "reading a snazzle file version " << $2 << endl; }
        ;
template:
        typelines
        ;
typelines:
        typelines typeline
        | typeline
        ;
typeline:
        TYPE STRING { cout << "new defined snazzle type: " << $2 << endl; }
        ;
body_section:
        body_lines
        ;
body_lines:
        body_lines body_line
        | body_line
        ;
body_line:
        INT INT INT INT STRING { cout << "new snazzle: " << $1 << $2 << $3 << $4 << $5 << endl; }
        ;
footer:
        END
        ;

%%
#include <stdio.h>

// stuff from lex that yacc needs to know about:
```

```
extern int yylex();
extern int yyparse();
extern FILE *yyin;

main() {
        // open a file handle to a particular file:
        FILE *myfile = fopen("in.snazzle", "r");
        // make sure it's valid:
        if (!myfile) {
                cout << "I can't open a.snazzle.file!" << endl;
                return -1;
        }
        // set lex to read from it instead of defaulting to STDIN:
        yyin = myfile;

        // parse through the input until there is no more:

        do {
                yyparse();
        } while (!feof(yyin));

}

void yyerror(char *s) {
        cout << "EEK, parse error!  Message: " << s << endl;
        // might as well halt now:
        exit(-1);
}
```

This is compiled and run just like the others. (By the way, I heartily recommend putting this crap into a makefile and never thinking about it again):

```
% yacc -d snazzle.y
% lex snazzle.lex
% g++ y.tab.c lex.yy.c -lfl -o snazzle
% cat in.snazzle
sNaZZle 1.3
type foo
type bar
type bas
0 0 10 5 foo
20 10 30 20 foo
7 8 12 13 bas
78 124 100 256 bar
end

% ./snazzle
reading a snazzle file version 1.3
new defined snazzle type: foo
new defined snazzle type: bar
new defined snazzle type: bas
new snazzle: 00105foo
new snazzle: 20103020foo
new snazzle: 781213bas
new snazzle: 78124100256bar
done with a snazzle file!
%
```

So, that concludes this little tutorial, or at least the baseline part of it. I'm now going to pick a few upgrades at random and show you how they might be done.

First little tweak: you'll notice that this completely ignores whitespace, so that you could put the entire snazzle file on a single line and it'll still be okay. You may or may not want this behaviour, but let's assume that's Bad and require there to be carriage returns after the lines just like there is in the sample file "in.snazzle". To do this, we need to do two things: recognize the '\n' token (lex), and add it to the grammar (yacc):

snazzle.lex:

```
%{
#include
#include "y.tab.h"
```

```
%}
%%
[ \t] ;
sNaZZle { return SNAZZLE; }
type { return TYPE; }
end { return END; }
[0-9]+\.[0-9]+ { yylval.fval = atof(yytext); return FLOAT; }
[0-9]+  { yylval.ival = atoi(yytext); return INT; }
[a-zA-Z0-9]+ {
        // we have to copy because we can't rely on yytext not changing underneath us:
        char *res = new char[strlen(yytext) + 1];
        strcpy(res, yytext);
        yylval.sval = res;
        return STRING;
}
\n    return ENDL;
. ;
%%
```

snazzle.y:

```
%{
#include
%}

// define the token types I can have:
%token INT FLOAT STRING
%token SNAZZLE TYPE
%token END ENDL
// since there's a different C datatype for each of the token type, yacc needs a
// union for the lex return value:
%union {
        int ival;
        float fval;
        char *sval;
}
// and then you just associate one of the defined token types with one of
// the union fields and we're happy:
%token  INT
%token  FLOAT
%token  STRING
%%
// the first rule defined is the highest-level rule, which in our
// case is just the concept of a whole "snazzle file":
snazzle:
header template body_section footer { cout << "done with a snazzle file!" << endl; }
        ;
header:
        SNAZZLE FLOAT ENDL { cout << "reading a snazzle file version " << $2 << endl; }
        ;
template:
        typelines
        ;
typelines:
        typelines typeline
        | typeline
        ;
typeline:
        TYPE STRING ENDL { cout << "new defined snazzle type: " << $2 << endl; }
        ;
body_section:
        body_lines
        ;
body_lines:
        body_lines body_line
        | body_line
        ;
body_line:
        INT INT INT INT STRING ENDL { cout << "new snazzle: " << $1 << $2 << $3 << $4 << $5 << endl; }
        ;
footer:
        END ENDL
        ;
%%
```

```
#include

// stuff from lex that yacc needs to know about:
extern int yylex();
extern int yyparse();
extern FILE *yyin;

main() {
        // open a file handle to a particular file:
        //FILE *myfile = fopen("a.snazzle.file", "r");
        FILE *myfile = fopen("in.snazzle", "r");
        // make sure it's valid:
        if (!myfile) {
                cout << "I can't open a.snazzle.file!" << endl;
                return -1;
        }
        // set lex to read from it instead of defaulting to STDIN:
        yyin = myfile;

        // parse through the input until there is no more:

        do {
                yyparse();
        } while (!feof(yyin));

}

void yyerror(char *s) {
        cout << "EEK, parse error!  Message: " << s << endl;
        // might as well halt now:
        exit(-1);
}
```

Also, I'm sick of typing lex and yacc so I made a makefile:

```
% cat Makefile
y.tab.c y.tab.h: snazzle.y
        yacc -d snazzle.y

lex.yy.c: snazzle.lex y.tab.h
        lex snazzle.lex

snazzle: lex.yy.c y.tab.c y.tab.h
        g++ y.tab.c lex.yy.c -lfl -o snazzle
% make snazzle
yacc -d snazzle.y
lex snazzle.lex
g++ y.tab.c lex.yy.c -lfl -o snazzle
% ./snazzle
reading a snazzle file version 1.3
new defined snazzle type: foo
new defined snazzle type: bar
new defined snazzle type: bas
new snazzle: 00105foo
new snazzle: 20103020foo
new snazzle: 781213bas
new snazzle: 78124100256bar
done with a snazzle file!
EEK, parse error!  Message: syntax error
%
```

Eek, we got a parse error! Why now? Well, it turns out that in.snazzle has an extra carriage return at the end:

```
% cat in.snazzle
sNaZZle 1.3
type foo
type bar
type bas
0 0 10 5 foo
20 10 30 20 foo
7 8 12 13 bas
```

```
78 124 100 256 bar
end

%
```

And if we remove it, snazzle would be happy:

```
% cat in.snazzle
sNaZZle 1.3
type foo
type bar
type bas
0 0 10 5 foo
20 10 30 20 foo
7 8 12 13 bas
78 124 100 256 bar
end
% ./snazzle
reading a snazzle file version 1.3
new defined snazzle type: foo
new defined snazzle type: bar
new defined snazzle type: bas
new snazzle: 00105foo
new snazzle: 20103020foo
new snazzle: 781213bas
new snazzle: 78124100256bar
done with a snazzle file!
%
```

Second little tweak: the first little tweak really isn't the best solution, as requiring all input files to have exactly one carriage return after data is unreasonable to say the least. We need to make it more general. Specifically, we should allow any number of carriage returns between lines. In yacc-land, this is best stated as "each line may be followed with one or more carriage returns", like so:

snazzle.y:

```
%{
#include
%}

// define the token types I can have:
%token INT FLOAT STRING
%token SNAZZLE TYPE
%token END ENDL
// since there's a different C datatype for each of the token type, yacc needs a
// union for the lex return value:
%union {
        int ival;
        float fval;
        char *sval;
}
// and then you just associate one of the defined token types with one of
// the union fields and we're happy:
%token  INT
%token  FLOAT
%token  STRING
%%
// the first rule defined is the highest-level rule, which in our
// case is just the concept of a whole "snazzle file":
snazzle:
header template body_section footer { cout << "done with a snazzle file!" << endl; }
        ;
header:
        SNAZZLE FLOAT ENDLS { cout << "reading a snazzle file version " << $2 << endl; }
        ;
template:
        typelines
        ;
typelines:
        typelines typeline
        | typeline
        ;
typeline:
```

```
        TYPE STRING ENDLS { cout << "new defined snazzle type: " << $2 << endl; }
        ;
body_section:
        body_lines
        ;
body_lines:
        body_lines body_line
        | body_line
        ;
body_line:
        INT INT INT INT STRING ENDLS { cout << "new snazzle: " << $1 << $2 << $3 << $4 << $5 << endl; }
        ;
footer:
        END ENDLS
        ;
ENDLS:
        ENDLS ENDL
        | ENDL ;
%%
#include

// stuff from lex that yacc needs to know about:
extern int yylex();
extern int yyparse();
extern FILE *yyin;
```

Note that this didn't require any changes to the lex file -- the underlying tokens didn't change, just how we used them. And of course, this actually works:

```
% make
lex snazzle.lex
g++ y.tab.c lex.yy.c -lfl -o snazzle
% cat in.snazzle
sNaZZle 1.3
type foo
type bar
type bas


0 0 10 5 foo
20 10 30 20 foo
7 8 12 13 bas

78 124 100 256 bar

end


% ./snazzle
reading a snazzle file version 1.3
new defined snazzle type: foo
new defined snazzle type: bar
new defined snazzle type: bas
new snazzle: 00105foo
new snazzle: 20103020foo
new snazzle: 781213bas
new snazzle: 78124100256bar
done with a snazzle file!
%
```

Third little tweak: wouldn't it have been nice if that parse error had given us the line to look at, so that we didn't have to guess-and-check the grammar? Unfortunately lex has no guaranteed way to get the line number (some versions have **yylineno**, but others don't). The best way to keep track of the line number is to have some global variable that you update whenever you see a carriage return:

snazzle.lex:

```
%{
#include
#include "y.tab.h"

int line_num = 1;
```

```
%}
%%
[ \t] ;
sNaZZle { return SNAZZLE; }
type { return TYPE; }
end { return END; }
[0-9]+\.[0-9]+ { yylval.fval = atof(yytext); return FLOAT; }
[0-9]+  { yylval.ival = atoi(yytext); return INT; }
[a-zA-Z0-9]+ {
        // we have to copy because we can't rely on yytext not changing underneath us:
        char *res = new char[strlen(yytext) + 1];
        strcpy(res, yytext);
        yylval.sval = res;
        return STRING;
}
\n    { ++line_num; return ENDL; }

. ;
%%
```

snazzle.y:

```
%{
#include
%}

// define the token types I can have:
%token INT FLOAT STRING
%token SNAZZLE TYPE
%token END ENDL
// since there's a different C datatype for each of the token type, yacc needs a
// union for the lex return value:
%union {
        int ival;
        float fval;
        char *sval;
}
// and then you just associate one of the defined token types with one of
// the union fields and we're happy:
%token  INT
%token  FLOAT
%token  STRING
%%
// the first rule defined is the highest-level rule, which in our
// case is just the concept of a whole "snazzle file":
snazzle:
        header template body_section footer { cout < "done with a snazzle file!" < endl; }
        ;
header:
        SNAZZLE FLOAT ENDLS { cout < "reading a snazzle file version " < $2 < endl; }
        ;
template:
        typelines
        ;
typelines:
        typelines typeline
        | typeline
        ;
typeline:
        TYPE STRING ENDLS { cout < "new defined snazzle type: " < $2 < endl; }
        ;
body_section:
        body_lines
        ;
body_lines:
        body_lines body_line
        | body_line
        ;
body_line:
        INT INT INT INT STRING ENDLS { cout < "new snazzle: " < $1 < $2 < $3 < $4 < $5 < endl; }
        ;
footer:
        END ENDLS
        ;
```

```
ENDLS:
        ENDLS ENDL
        | ENDL ;
%%
#include

// stuff from lex that yacc needs to know about:
extern int yylex();
extern int yyparse();
extern FILE *yyin;
extern int line_num;

main() {
        // open a file handle to a particular file:
        //FILE *myfile = fopen("a.snazzle.file", "r");
        FILE *myfile = fopen("in.snazzle", "r");
        // make sure it's valid:
        if (!myfile) {
                cout << "I can't open a.snazzle.file!" << endl;
                return -1;
        }
        // set lex to read from it instead of defaulting to STDIN:
        yyin = myfile;

        // parse through the input until there is no more:

        do {
                yyparse();
        } while (!feof(yyin));

}

void yyerror(char *s) {
        cout << "EEK, parse error on line " << line_num << "!  Message: " << s << endl;
        // might as well halt now:
        exit(-1);
}
```

Which is pretty cool when you make a "type oops" definition in the middle of the body instead of where it's supposed to be:

```
% make
yacc -d snazzle.y
lex snazzle.lex
g++ y.tab.c lex.yy.c -lfl -o snazzle
% cat in.snazzle
sNaZZle 1.3
type foo
type bar
type bas
0 0 10 5 foo
20 10 30 20 foo
type oops
7 8 12 13 bas
78 124 100 256 bar
end
% ./snazzle
reading a snazzle file version 1.3
new defined snazzle type: foo
new defined snazzle type: bar
new defined snazzle type: bas
new snazzle: 00105foo
EEK, parse error on line 7!  Message: syntax error
%
```

Of course, "syntax error" isn't very helpful, but at least you've got the line number. :)

Tips

directly returning terminal characters

If you have individual characters that are also terminal symbols, you can have lex return them directly instead of having to create a `%token` for them:

```
:               { return ':'; }
;               { return ';'; }
```

You could also list a bunch of them in a single line with this shortcut:

```
[\(\)\{\}:;=,]    { return yytext[0]; }
```

On the yacc side, you can also use them directly:

```
thing: FOO ':' BAR ';'
```

actions before the end of the grammar

The coolest thing I've discovered about yacc so far is that you can put action statements *in the middle of the grammar* instead of just at the end. This means you can get information as soon as it's read, which I find is very useful. For example, a C++ function call might look like this:

```
functioncall:
  returntype name '(' args ')' body
```

You can always add actions at the end of the line, like this:

```
functioncall:
  returntype name '(' args ')' body { cout << "defined function '" << $2 << "'" << endl; }
```

But then you get the function name after 'body' is evaluated, which means the whole thing has been parsed before you find out what it was! However, you can embed the code block in the middle of the grammar and it will be run before 'body' is evaluated:

```
functioncall:
  returntype name { cout << "defining function '" << $2 << << endl; } '(' args ')' body
```

whitespace in lex/yacc files

I've discovered that, unlike most unix tools, lex and yacc are surprisingly lenient about whitespacing. This is really nice, because it's hard enough to understand their syntax without the ability to reindent things. :)

```
thing: FOO ':' BAR { cout << "found a foo-bar!" << endl; }
  | BAS ':' BAR { cout << "found a bas-bar!" << endl; }

thing:
  FOO ':' BAR { cout << "found a foo-bar!" << endl; }
  |
  BAS ':' BAR { cout << "found a bas-bar!" << endl; }

thing:
  FOO ':' BAR
    { cout << "found a foo-bar!" << endl; }
  | BAS ':' BAR
    { cout << "found a bas-bar!" << endl; }
```

avoiding -lfl and link problems with yywrap

If you know that you only want to parse one file at a time (usually a good assumption), you don't need to provide your own yywrap function that would just "return 1". Inside your lex file, you can say "%option noyywrap" and the generated output will define yywrap as a local macro returning 1. This has the added bonus of removing the requirement to link with the flex libraries, since they're only needed to get the default yywrap defined as a function returning 1.

renaming and moving lex's output

Usually you want to rename the generated identifiers so that you can have more than one lex-based parser in a program. This is normally accomplished by passing a "-P" or "-p" option to lex and yacc, but you can also do it in lex with the **%option prefix="foo"** construct. Similarly, if you don't like the default output filename, you can specify it directly in your lex file with the **%option outfile="lex.foo.cc"** construct.

Advanced

start states

In a lex/yacc grammer, it is almost impossible to allow multiline commenting such as C's `/* .. */`. What you really want is for the parser go into a sort of "ignore everything" state when it sees "/*", and go back to normal when it sees "*/". Clearly you wouldn't want yacc to do this, or else you'd have to put a sort of optional 'comment' target all over every contruct in your syntax. So you can see how it fell to lex to implement some way to do this.

Lex allows you to specify *start states*, which are just regex pattern rules like any other **but they're only matched if you're in a particular state**. The syntax for a pattern that should only be matched if we're in state 'FOO' looks like this:

```
<FOO>bar     ;  // we're in state FOO and we saw "bar"
```

Note that *you* come up with the state names -- they're not predefined or anything. Though when you create a state, you do have to declare it in your lex file's control section (that's the first section, before the first "%%"):

```
%x FOO
```

So how do you get into the state? There's a special lex construct (I think it's ultimately a C macro) that goes into any regular code block to get there:

```
bar          { BEGIN(FOO); }  // we saw a "bar", so go into the "FOO" state
```

And how about getting back out of that state and going back to where you were initially? Instead of somthing obvious (say, "END()"?), they decided to make a default state called "INITIAL." Any lex match pattern that doesn't have a state in front of it is assumed to be in the INITIAL state. To get back to it, you just jump to it:

```
bas          { BEGIN(INITIAL); }  // jump back to normal pattern matching
```

Note that the BEGIN thing is for all intents and purposes normal C code, despite the fact that it's not. :) What that means is that you can treat it like code and put it anywhere you want -- you can even make it conditional:

```
bar          { do_something(); BEGIN(FOO); do_something_else(); }
..
bar          { if (some_value == 42) BEGIN(FOO); }
..
```

Back to the original problem -- multiline commenting. Here's a way to do C-style block commenting using start states:

```
\/\*         {   // start of a comment: go to a 'COMMENTS' state.
  iscommented = true;  // random local variable that this example doesn't use for anything
  BEGIN(COMMENTS);
  }
<COMMENTS>\*\/ {   // end of a comment: go back to normal parsing.
  iscommented = false;
  BEGIN(INITIAL);
  }
<COMMENTS>\n      { ++linenum; }   // still have to increment line numbers inside of comments!
<COMMENTS>.       ;   // ignore every other character while we're in this state
```

Note that you can also have a lex match pertain to *multiple* states by listing them all:

```
<MYSTATE,ANOTHERSTATE>foo { ... }
```

This occasionally comes in handy.

reading gzipped input

This is either very tricky or very easy, depending on whether you have to do it or whether you've already done it. :)

The core idea is that you need to redirect lex's input, which is supposed to be yyin. However, being a good linux monkey, I use the gnu version of lex (flex), which doesn't use yyin in the first place, so that means the remainder of this is only useful for fellow flex users.

For speed reasons, flex bypasses yyin's character-based getc() in favor of much faster block-based fread(). So while overriding yyin won't get you anywhere, you can still override lex's call to fread.

In the first section of your lex file, you'll want to declare that you have a better input function, and then tell lex it has to use yours instead:

```
	extern int my_abstractread(char *buff, int buffsize);
	#define YY_INPUT(buff, res, buffsize) (res = my_abstractread(buff, buffsize))
```

Somewhere or another (I put it in my top-level files because it doesn't have to go in with the lex/yacc source) you need to define your input function. Mine looks like this:

```
#include <stdio.h>  // for FILE
#include <zlib.h>  // direct access to the gzip API

// (ew!) global variables for the open input file.  Only one of these at
// a time will be in use: my_gzfile for gzipped input, or my_file for
// non-gzipped input:
gzFile my_gzfile = 0;
FILE *my_file = 0;

// block input reader that handles regular or gzipped input:
int my_abstractread(char *buff, int buffsize) {

	// This is called on a request by lex to get the next 'buffsize' bytes of data
	// from the input, and return the number of bytes read.

	// either the file was gzipped:
	if (my_gzfile) {

		int res = gzread(my_gzfile, buff, buffsize);
		if (res == -1) {
			// file error!
			abort();
		}

		return res;
	}

	// or else it's not:
	else if (my_file) {

		int res = fread(buff, 1, buffsize, my_file);
		if (ferror(my_file)) {
			// file error!
			abort();
		}

		return res;
	}

	// error! no open file to read?
	abort();

	return 0;
}
```

Then, when you find out what file you're being asked to parse, you need to figure out whether it's gzipped or not. I do this by seeing if the last three characters are ".gz", which is usually a good assumption:

```
        if (filename.rfind(".gz") == filename.size() - 3) {

                // set the "gzipped" global file variable:
                my_gzfile = gzopen(filename.c_str(), "r");
                if (!my_gzfile) {
                        // file not found?
                        abort();
                }
        }
        else {

                // set the "un-gzipped" global file variable:
                my_file = fopen(filename.c_str(), "r");
                if (!my_file) {
                        // file not found?
                        abort();
                }
        }
```

And finally, you have to change your yyparse()-calling loop to use these new file handles instead of yyin:

```
        do {
                myparse();
        } while ((my_file && !feof(my_file)) || (my_gzfile && !gzeof(my_gzfile)));
```

For reference, the API to libz is available here (last I looked). I have been consistently surprised with how easy it is to use libz directly to avoid silly hacks like "popen"ing "gunzip -c" and redirecting its output. Kudos to the libz team!

Also, it's been brought to my attention that half of my code is unneeded due to one nifty little feature that libz provides: it works equally well on **unzipped** files. Hence, we don't need the my_file pointer at all because we could just use libz to filter *all* of our input for us. That makes the lex-external code much smaller:

```
gzFile my_gzfile = 0;

int my_abstractread(char *buff, int buffsize) {

        // called on a request by lex to get the next 'buffsize' bytes of data
        // from the input, and return the number of bytes read.

        int res = gzread(my_gzfile, buff, buffsize);
        if (res == -1) {
                // file error!
                abort();
        }

        return res;
}

...

        // libz is very nice in that it handles unzipped files as well, which
        // means that no matter what the input file is we can just pass it
        // through gzopen and not worry about it:
        my_gzfile = gzopen(filename.c_str(), "r");
        if (!my_gzfile)  {
                // file cannot be opened
                abort();

        ...

        do {
                myparse();
        } while (!gzeof(my_gzfile));
```

Regular expression overview

You've used regexs before without knowing it, for example when you use wildcards on file names:

```
% ls *asdf*
asdf
asdf2
asdf3
new.asdf
new.asdf.mangled
%
```

The shell's idea of regular expressions isn't quite accurate with the "real" definition of regular expressions, but at least the idea is the same. Here, we're telling ls to list all the files that have "asdf" somewhere in the name. We could ask for just the files *starting* with asdf by saying "asdf*", or all the files ending with asdf with "*asdf". The asterisk basically means "anything can go here".

Regular expressions are really just the scientific deconstruction of such pattern matching. As such, you can imagine they're not a whole lot of fun. :) There's actually an entire O'Reilly book dedicated to them, if you really want to see what they're all about. (I'd love to make a wisecrack about being great for insomnia, but right now it's ranked #5,076 on Amazon's sales rank. Guess it can't be too unbearable!)

With that said, my little overview here is clearly not going to be exhaustive, but should give you the general idea. Lex's regular expressions consist of "characters" and "meta-characters". "Characters" are interpreted literally as real text, whereas "meta-characters" change how the search works. For example, listing "foo" as a regular expression will match exactly one string, which is "foo". But if you add the metacharacter "+" (which means "one or more of the previous character") after the "f" to get "f+oo", it will match "foo", "ffoo", and "ffffffffffffffffoo". A table of meta-characters follows:

metacharacter	description
+	previous expression can match one or more times
*	previous expression can match zero or more times
?	previous expression can match zero or one time
.	can match any character except the carriage return '\n'

I say "expression" above instead of just "character" because you can also make groups of things. By saying "[abcde]" it will match any *one* of the characters in the brackets, so you'll match "a" and "b" but not "ac". If you add a plus after the closing bracket, though, then you *will* match "ac" as well.

Brackets also allow negation: "[^abc]" will match anything that's not in the brackets, so "d" and "foo" would pass but not "b".

And, most useful of all, brackets allow for ranges: "[a-e]" is the same as "[abcde]". So you will frequently see stuff like "[0-9]+" to match integer values, or "[a-zA-Z]+" to match words, etc.

So how do you match an actual bracket, or an actual period, if the regular expression searcher is immediately going to think you're defining a meta-character? By adding a backslash (\) in front of it! So "a\.b" will match the string "a.b" but not "acb", whereas "a.b" as a regular expression will match both. And to be complete, backslash itself can be backslashed, so that "a\\." will match the string "a\b". (What fun!)

This stuff just takes some playing with to get used to. It's not really that bad. :)

Chris verBurg
~~*April 2, 2003*~~
~~*July 23, 2004*~~
February 9, 2005