w w w . a q u a m e n t u s . c o m

What are Flex and Bison?

Flex and Bison are aging unix utilities that help you write very fast parsers for almost arbitrary file formats. Formally, they implement Look-Ahead-Left-Right (as opposed to "recursive descent") parsing of non-ambiguous context-free (as opposed to "natural language") grammars.

Why should you learn the Flex/Bison pattern syntax when you could just write your own parser? Well, several reasons. First, Flex and Bison will generate a parser that is virually guaranteed to be faster than anything you could write manually in a reasonable amount of time. Second, updating and fixing Flex and Bison source files is a lot easier than updating and fixing custom parser code. Third, Flex and Bison have mechanisms for error handling and recovery, which is something you definitely don't want to try to bolt onto a custom parser. Finally, Flex and Bison have been around for a long time, so they far freer from bugs than newer code.

This webpage is supposed to be a tutorial for complete novices needing to use Flex and Bison for some real project. I wrote it as I was learning them, so hopefully it explains things right as you need to know them. The Flex and Bison tutorials I found Googling around tended to be really heavy on theory and details.

lex vs. flex, yacc vs. bison

In addition to hearing about "flex and bison", you will also hear about "lex and yacc". "lex

and yacc" are the original tools; "flex and bison" are their almost completely compatible newer versions. Only very old code uses lex and yacc; most of the world has moved on to Flex and Bison.

All four of the above are C-based tools; they're written in C, but more important their output is C code. However, my project was in C++ -- so this is also a tutorial on how to use C++ with Flex and Bison!

what we'll do!

Let's create a Flex/Bison 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 typedef 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 Bison thinks about them. We start by saying that a "snazzlefile" is comprised of four general parts; we'll call them "header", "typedefs", "body", and "footer". Then we describe the sub-parts that comprise each of those four general parts, and then what comprises each of the those sub-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. (But *not* the individual characters "1" and "5" that make it up.) Terminal symbols are the boundary between Flex and Bison: Flex sees the individual "1" and "5" characters but combines them into "15" before giving it to Bison.

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 (which 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 Flex's power by making a program that just lexes our input without using Bison to parse it. Here's the first whack:

snazzle.l

```
1         %{
2         #include <iostream>
3         using namespace std;
4         #define YY_DECL extern "C" int yylex()
5         %}
6         %%
7         [ \t\n]         ;
8         [0-9]+\.[0-9]+  { cout << "Found a floating-point number:" << yytext
9         [0-9]+          { cout << "Found an integer:" << yytext << endl; }
10        [a-zA-Z0-9]+    { cout << "Found a string: " << yytext << endl; }
11        %%
12        main() {
13                // lex through the input:
14                yylex();
15        }
```

Flex and Bison files have three sections:

1. the first is sort of "control" information,
2. the second is the actual token/grammar definitions,
3. the last is C code to be copied verbatim to the output.

These sections are divided by %%, which you see on lines 6 and 11. Let's go through this line by line.

- Lines 1 (and 5) are delimeters that tell Flex that lines 2 through 4 are C code to be copied directly to the generated lexer.
- Lines 2 and 3 are to get access to `cout`.
- Line 4 is to force the g++ compiler to not apply "name-mangling" to the `yylex` identifier. If we don't do this, g++ generates the function's name as something like "_Z5yylexv", where no one will ever be able to find it or link to it.
- Line 6 is %%, which means we're done with the control section and moving on to the token section. Notice that we don't have much in our control section -- the Bison control section gets a lot more usage than the Flex one. And here, we wouldn't need one at all if we weren't using this from C++.
- Lines 7-10 are all the same (simple) format: 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 Flex 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 flex output; accordingly, you can have a single statement or you can have curly braces with a whole bunch of statements. Some specifics on the file format I discovered: the action has to be left-justified (if there's whitespace beginning a line where a pattern is expected, the line is considered a comment!); the separation between the pattern and the action is just whitespace (even just a single space will do); the action is not limited to a single line if you use curly braces.
- Line 11 is another %% delimiter, meaning we're done with the second section and we can go onto the third.
- Lines 12-15 are the third section, which 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 Flex 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 link them together.

This example can be compiled by running this:

```
% flex snazzle.l
```

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 flex libraries. (If you don't have Flex installed, and you can't get it installed, you might try "-ll" instead -- that requests lex's .so instead of flex's .so. If that doesn't work, ask someone who'll know where on your system the Flex libraries are kept.) (Or, leave the -l off completely, and do a quick Google search to figure out how to implement stub versions of the missing functions it complains about.)

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
 !
```

Pretty cool! Notice that the exclamation mark at the very end was just echoed: when Flex 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
```

Pretty much everything else will need to return a token to Bison, but more on that later.

reading from a file

OK, first big upgrade: reading from a file. Right now we're reading from STDIN, which is kind of annoying, because most of the time you'd really like to pick a file to read from. Flex reads its input from a global pointer to a C `FILE` variable called `yyin`, which is set to STDIN by default. All you have to do is set that pointer to your own file handle, and it'll read from it instead. That changes our example as follows (the differences are in bold):

snazzle.l

```
%{
#include <iostream>
using namespace std;
#define YY_DECL extern "C" int yylex()
%}
%%
[ \t\n]          ;
[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();
}
```

Here's how to run our new code, though there's nothing earth-shattering:

```
% flex snazzle.l
% 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
%
```

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. Even though these are the types of tokens (which is what Flex returns), defining token types is done in the Bison file. We're also going to move `main()` into the copied-C-code section of the Bison file now:

snazzle.y

```
%{
#include <cstdio>
#include <iostream>
using namespace std;

// stuff from flex that bison needs to know about:
extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;

void yyerror(const char *s);
%}

// Bison fundamentally works by asking flex to get the next token, which it
// returns as an object of type "yystype".  But tokens could be of any
// arbitrary data type!  So we deal with that in Bison by defining a C union
// holding each of the types of tokens that Flex could return, and have Bison
// use that union instead of "int" for the definition of "yystype":
%union {
        int ival;
        float fval;
        char *sval;
}

// define the "terminal symbol" token types I'm going to use (in CAPS
// by convention), and associate each with a field of the union:
%token <ival> INT
%token <fval> FLOAT
%token <sval> STRING
```

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

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

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

snazzle.l

```
%{
#include <cstdio>
#include <iostream>
using namespace std;
#define YY_DECL extern "C" int yylex()

#include "snazzle.tab.h"  // to get the token types that we return

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

```
[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 underne
        yylval.sval = strdup(yytext);
        return STRING;
}
.                ;
%%
```

Since the Flex file now has to #include a file generated by Bison (snazzle.tab.h), we have to run Bison first:

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

You'll notice that the input is echoed out backward! This is because for each of the rules you define, Bison 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 Bison's internal stack! So a better implementation of the Bison grammar would be left-recursion:

snazzle.y

```
%{
#include <cstdio>
#include <iostream>
using namespace std;

#include "y.tab.h"  // to get the token types that we return

// stuff from flex that bison needs to know about:
extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;

void yyerror(const char *s);
%}

// Bison fundamentally works by asking flex to get the next token, which it
```

```
// returns as an object of type "yystype".  But tokens could be of any
// arbitrary data type!  So we deal with that in Bison by defining a C union
// holding each of the types of tokens that Flex could return, and have Bison
// use that union instead of "int" for the definition of "yystype":
%union {
        int ival;
        float fval;
        char *sval;
}

// define the "terminal symbol" token types I'm going to use (in CAPS
// by convention), and associate each with a field of the union:
%token <ival> INT
%token <fval> FLOAT
%token <sval> STRING

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

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 flex 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(const 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:

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

Now that the groundwork is completed, we can finally define the real file format in the Bison file. First we make the easy tokens: the ones that are used to represent constant strings like SNAZZLE, TYPE, and END, respectively representing the strings "sNaZZle", "type", and "end". Then we flesh out the actual rules in the Bison grammar, and end up with this beautiful object d'art:

snazzle.l

```
%{
#include <iostream>
#include "snazzle.tab.h"
using namespace std;
#define YY_DECL extern "C" int yylex()
%}
%%
[ \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 underne
        yylval.sval = strdup(yytext);
        return STRING;
}
.               ;
%%
```

snazzle.y

```
%{
#include <cstdio>
#include <iostream>
using namespace std;

#include "y.tab.h"  // to get the token types that we return

// stuff from flex that bison needs to know about:
extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;
```

```
void yyerror(const char *s);
%}

// Bison fundamentally works by asking flex to get the next token, which it
// returns as an object of type "yystype".  But tokens could be of any
// arbitrary data type!  So we deal with that in Bison by defining a C union
// holding each of the types of tokens that Flex could return, and have Bison
// use that union instead of "int" for the definition of "yystype":
%union {
        int ival;
        float fval;
        char *sval;
}

// define the constant-string tokens:
%token SNAZZLE TYPE
%token END

// define the "terminal symbol" token types I'm going to use (in CAPS
// by convention), and associate each with a field of the union:
%token <ival> INT
%token <fval> FLOAT
%token <sval> 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!
        ;
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
        ;
footer:
```

```
        END
        ;

%%

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 flex 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(const 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:

```
% bison -d snazzle.y
% flex snazzle.l
% g++ snazzle.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.

a makefile

When you finally get sick of manually rerunning Flex and Bison, and also forgetting in which order to do it, I heartily recommend setting up a makefile. Here's a sample one I made:

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

lex.yy.c: snazzle.l snazzle.tab.h
        flex snazzle.l

snazzle: lex.yy.c snazzle.tab.c snazzle.tab.h
        g++ snazzle.tab.c lex.yy.c -lfl -o snazzle
% make snazzle
bison -d snazzle.y
flex snazzle.l
g++ snazzle.tab.c lex.yy.c -lfl -o snazzle
%
```

forcing carriage returns

Next 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 (flex), and add it to the grammar (bison):

```
snazzle.l
%{
#include
#include "snazzle.tab.h"
using namespace std;
#define YY_DECL extern "C" int yylex()
%}
%%
[ \t\n]          ;
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 underne
```

```
        yylval.sval = strdup(yytext);
        return STRING;
}
\n              return ENDL;
.               ;
%%
```

snazzle.y

```
%{
#include <cstdio>
#include <iostream>
using namespace std;

#include "y.tab.h"  // to get the token types that we return

// stuff from flex that bison needs to know about:
extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;

void yyerror(const char *s);
%}

// Bison fundamentally works by asking flex to get the next token, which it
// returns as an object of type "yystype".  But tokens could be of any
// arbitrary data type!  So we deal with that in Bison by defining a C union
// holding each of the types of tokens that Flex could return, and have Bison
// use that union instead of "int" for the definition of "yystype":
%union {
        int ival;
        float fval;
        char *sval;
}

// define the constant-string tokens:
%token SNAZZLE TYPE
%token END ENDL

// define the "terminal symbol" token types I'm going to use (in CAPS
// by convention), and associate each with a field of the union:
%token <ival> INT
%token <fval> FLOAT
%token <sval> 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!" << enc
        ;
header:
        SNAZZLE FLOAT ENDL { cout << "reading a snazzle file version " << $2 <<
        ;
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
        ;
footer:
        END 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;

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

}

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

But when we go to run it, we get a parse error:

```
% make
% ./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
%
```

Why?? 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!
%
```

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

snazzle.y

```
%{
#include <cstdio>
#include <iostream>
using namespace std;

#include "y.tab.h"  // to get the token types that we return

// stuff from flex that bison needs to know about:
extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;

void yyerror(const char *s);
%}

// Bison fundamentally works by asking flex to get the next token, which it
// returns as an object of type "yystype".  But tokens could be of any
// arbitrary data type!  So we deal with that in Bison by defining a C union
// holding each of the types of tokens that Flex could return, and have Bison
// use that union instead of "int" for the definition of "yystype":
%union {
        int ival;
        float fval;
        char *sval;
}

// define the constant-string tokens:
%token SNAZZLE TYPE
%token END ENDL

// define the "terminal symbol" token types I'm going to use (in CAPS
// by convention), and associate each with a field of the union:
%token <ival> INT
%token <fval> FLOAT
%token <sval> 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!" << enc
        ;
header:
        SNAZZLE FLOAT ENDLS { cout << "reading a snazzle file version " << $2 <<
        ;
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
        ;
footer:
        END ENDLS
        ;
ENDLS:
        ENDLS ENDL
        | ENDL ;
%%
```

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

```
% make
flex snazzle.l
g++ snazzle.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!
%
```

line numbers

Next 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 Flex has no guaranteed way to get the line number (well, there's `yylineno`, but it's almost completely manual, and in some situations makes your parser very slow, so you might as well just do it yourself). The best way to keep track of the line number is to have a global variable that you update whenever you see a carriage return:

snazzle.l

```
%{
#include
#include "y.tab.h"
using namespace std;
#define YY_DECL extern "C" int yylex()
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]+   {
        yylval.sval = strdup(yytext);
        return STRING;
}
\n             { ++line_num; return ENDL; }
.              ;
%%
```

snazzle.y

```
%{
#include <cstdio>
#include <iostream>
using namespace std;

#include "y.tab.h"  // to get the token types that we return

// stuff from flex that bison needs to know about:
extern "C" int yylex();
extern "C" int yyparse();
extern "C" FILE *yyin;
extern int line_num;

void yyerror(const char *s);
%}

// Bison fundamentally works by asking flex to get the next token, which it
// returns as an object of type "yystype".  But tokens could be of any
// arbitrary data type!  So we deal with that in Bison by defining a C union
// holding each of the types of tokens that Flex could return, and have Bison
// use that union instead of "int" for the definition of "yystype":
%union {
        int ival;
```

```
        float fval;
        char *sval;
}

// define the constant-string tokens:
%token SNAZZLE TYPE
%token END ENDL

// define the "terminal symbol" token types I'm going to use (in CAPS
// by convention), and associate each with a field of the union:
%token <ival> INT
%token <fval> FLOAT
%token <sval> 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!'
        ;
header:
        SNAZZLE FLOAT ENDLS { cout < "reading a snazzle file version " < $2 < er
        ;
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 < $
        ;
footer:
        END ENDLS
        ;
ENDLS:
        ENDLS ENDL
        | ENDL ;
%%

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(const char *s) {
        cout << "EEK, parse error on line " << line_num << "!  Message: " << s <
        // 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
bison -d snazzle.y
flex snazzle.l
g++ snazzle.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 Flex 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 Bison side, you can also use them directly:

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

actions before the end of the grammar

The coolest thing I've discovered about Bison 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 <
```

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; } '('
```

whitespace in flex/bison files

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

```
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 flex 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 identifiers

Flex and Bison generate code that uses the global namespace, which means that if you ever try to have more than one in a single program you're going to have identifier collisions. To get around that, you can tell both Flex and Bison to prefix their identifiers with a custom string you specify.

In Flex, you can use either the command line option **-P foo** or the option syntax **%option prefix="foo"** in the first section of the file.

In Bison, you can use either the command line option **-p foo** or the option syntax **%name-prefix "foo"**.

moving output

Similar to why you rename identifiers, you usually don't want to use the default output file names because multiple parsers will all step on each other.

In Flex, you can specify either **-o lex.foo.cc** on the command line (it has to be before your input file!) or **%option outfile="lex.foo.cc"** in the first section of the .l file.

In Bison, you can specify either **-o "foo.tab.cc"** on the command line or **%output "foo.tab.cc"** in the first section of the .y file. However, Bison usually names its output after its input anyway; an input file named "foo.y" will already generate "foo.tab.c".

ADVANCED

start states

In a Flex/Bison 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 Bison to do this, or else you'd have to put optional 'comment' targets all over every contruct in your syntax; you can see how it fell to Flex to implement some way to do this.

Flex 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 Flex 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 Flex 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 Flex 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.
        BEGIN(COMMENTS);
```

```
        }
<COMMENTS>\*\/    {   // end of a comment: go back to normal parsing.
        BEGIN(INITIAL);
        }
<COMMENTS>\n      { ++linenum; }   // still have to increment line numbers insic
<COMMENTS>.       ;   // ignore every other character while we're in this state
```

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

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

This occasionally comes in handy, such as keeping you from having to duplicate the line-counting pattern/code above.

reading gzipped input

This is, surprisingly, rather easy. The basic idea is to filter Flex's input through libz. libz is very nice because it will pass through, unchanged, any input which isn't zipped -- so we don't need to check if the input is zipped and handle it differently!

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!

In the first section of your Flex file, you'll want to declare that you have a better input function, and then tell Flex 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 Flex/Bison source) you need to define your input function. Mine looks like this:

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

// (ew!) global variable for the gzip stream.
gzFile my_gzfile = 0;

int my_abstractread(char *buff, int buffsize) {

        // called on a request by Flex 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;
```

```
}
```

Then, instead of using fopen, you use libz:

```
// 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();
}
```

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

```
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. Flex'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 enclosing whatever you want in parentheses.

Brackets are very cool -- 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".

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
2011-12-08