Jumpstart Flex and Bison

Bo Waggoner

Updated: 2014-10-18

Abstract

Flex and Bison are tools for writing a compiler (they are free/replacement versions of the famous Lex and Yacc). We try to get some minimal working examples going quickly.

1 Installation

You can download and install Flex and Bison from the web:
Flex homepage: `http://flex.sourceforge.net/`.
Bison homepage: `http://www.gnu.org/software/bison/`.
Or on Linux, install using

```
$ apt-get install flex
$ apt-get install bison
```

(substituting the appropriate package manager for apt-get).
The websites have links to the manuals.

2 Setup

You write a file, say "example1.l". Flex converts this to a C or C++ program (your choice) that will take in a stream of characters and try to match patterns. Each time it succesfully matches a pattern, it takes some appropriate action, which usually includes outputting a "token" indicating what was matched.
Then you write a file, say "example1.y", which includes a grammar. Yacc converts this to a C or C++ program that takes in a stream of "tokens", parses them into a parse tree according to the grammar, and takes some appropriate actions.
In practice, **you compile the outputs of flex and bison together into a single C or C++ program** that takes a stream of input characters. Each time flex matches on this input, it does its action and sends a token to yacc, which attempts to update its parse tree and take its actions.

3 Example 0

We will write an interpreter for programs that look like this:

```
hello-world;
hello-world ; hello-world ;
quit;
```

There are two commands "hello-world" and "quit". Every command must be followed a semicolon. A valid program consists of or more instances of a hello-world command, followed by one instance of a quit command.
Gameplan: write a *flex* file *example0.l* and a *bison* file *example0.y*. Run the respective programs to get some C++ output files. Compile this with C++ to get our final parser program.

```
  // FLEX file example0.l
  // NOTE: comments should be indented (not at left margin!)
%{
#include <stdio.h>
#include "example0.tab.h"  // the output of bison on example0.y
void yyerror(char*);  // we need to forward declare these functions,
int yyparse(void);    // don't worry about them
%}

%%

  // on reading this   do this
";"                  return SEMICOLON;
"hello-world"        {printf("got HELLO token\n"); return HELLO;}
"quit"               {printf("got QUIT token\n"); return QUIT;}
[ \t\n]+             ;  // do nothing on whitespace

%%

void yyerror(char* str) {printf("ERROR: Could not parse!\n");}
int yywrap(void) { }
int main(void) {
  // we don't want to do anything extra, just start the parser
  yyparse();  // yyparse is defined for us by flex
}
```

A flex program consistes of three sections, separated by %%. The first contains a C++ section, rounded by %{ %}, that is included verbatim in the output C++ file that flex produces.
The second section is the one where we define what to read and what to do. On the left is a regular expression; on the right is the action to take upon reading something that matches that regular expression. In this case, the action we take is to "return" a token to bison, who will attempt to construct a parse tree from them and take actions. To actually return the string matched by the regular expression, see example1. (Note that SEMICOLON, HELLO, and QUIT will all be constants defined in example0.tab.h, which will be produced by bison, as we'll see next.)
The third section just consists of verbatim C++ again (no need to even include the %{ %} this time). This includes the main method of our program. You could put this section in a separate C++ file altogether and compile it with those produced by flex and bison.

```
  // BISON file example0.y
%{
#include <stdio.h>
#include <stdlib.h>
extern int yylex();
extern void yyerror(char*);
%}

  // the different types of tokens we want to return
%union {
  int     int_token;
}

  // list the different tokens of each type
%token <int_token>  QUIT HELLO SEMICOLON

  // indicate which of the below nodes is the root of the parse tree
%start root_node

%%

root_node:  hello_node root_node | quit_node;

hello_node: HELLO SEMICOLON
  {printf("parsed a hello node!\nHello, user!\n");};

quit_node:  QUIT SEMICOLON
  {printf("parsed a quit node!\nGoodbye, user!\n"); exit(0);};
```

A bison program likewise has three sections, separated by %%. However, in this example, we did not need the third section so we omitted it and its %% separator. The first section contains two types of things:

1. Some C++ to be included verbatim at the top of the generated file, again wrapped with %{ %}. For instance, if you use std::map in the code below, you'd better #include <map> here.
2. Some directives to bison of the form %directive. Here we use the most basic ones:
   (a) %union: See, flex has to send "tokens" to bison. In addition to those, it can sometimes send values, and those values must be of particular types. Those types are defined here. However, in example0 we just use int as a default and don't worry about it. In example1 we will actually make use of this feature.
   (b) %token: This lists the possible tokens flex can return. They're associated with the type of data flex returns in the yystype object. Here, I've specified that when the token is QUIT, HELLO, or SEMICOLON, the kind of data is a "int_token" (though, again, the choice of int doesn't matter in this example).
   (c) %start: This just tells us which of the nodes of the parse tree is the "root" (place where the program starts).

The second section contains the actual grammar. The syntax is name_of_node: rule 1 {what to do on rule 1} | rule 2 {what to do on rule 2} | ... ;
Here, the grammar says a program consists of a *root_node*. A *root_node* is either a *hello_node* followed by a *root_node*, or a *quit_node*. We have specified actions to take (print out a message) upon successfully parsing a *hello_node* or a *quit_node*.

To compile (assuming Linux here, may require modification to your OS):

```
$ bison -d example0.y
$ flex example0.l
$ gcc -o parser0 lex.yy.c example0.tab.c
```

The option "-d" for bison tells it to play nice with flex rather than creating its own standalone thing. The end result of these three lines, assuming no errors, is to create a program "parser0". When run, it waits for input from the command line and attempts to parse it.
When run, the program just accepts input from the command line.

```
$ ./parser0
hello-world
got HELLO token
```

So far, so good. The lexer has observed the hello-world token. However, the parser needs a semicolon after it in order to parse the hello-world statement. There can be arbitrary whitespace in between though.

```
;
parsed a HELLO statement!
Hello, user!
```

Perfect.

```
quit;
got QUIT token
parsed a quit statement!
Goodbye, user!
```

4 Example 1

This time we'll make something that can interpret a program from file. (Not *quite* compiling yet!) The ridiculously simple programming language can only store values into variables and do arithmetic, for instance:

```
x = 33; y = 2+2.1;
print x/(y+5);
```

Its programs consist of one or more statements. Each statement is either an assignment of an expression to a variable, or a command to print an expression. An expression is just an arithmetic expression involving numbers and variables.

```
  // FLEX file example1.l
%{
#include <cstdio>
#include <cstdlib>
#include <string>
using namespace std;
#include "example1.tab.h"  // output of bison on example1.y
void yyerror(char*);
int yyparse(void);
%}

%%

[ \t\n]+                ;  // do nothing on whitespace
"print"                 return PRINT;
[a-zA-Z][a-zA-Z0-9]*    {yylval.str_val = new string(yytext); return VARIABLE;}
[0-9][0-9]*(.[0-9]+)?   {yylval.double_val = atof(yytext); return NUMBER;}
"="                     return EQUALS;
"+"                     return PLUS;
"-"                     return MINUS;
"*"                     return ASTERISK;
"/"                     return FSLASH;
"("                     return LPAREN;
")"                     return RPAREN;
";"                     return SEMICOLON;

%%

void yyerror(char* str) {printf("Parse Error: \n%s\n",str);}
int yywrap(void) { }
int main(int num_args, char** args) {
  if(num_args != 2) {printf("usage: ./parser1 filename\n"); exit(0);}
  FILE* file = fopen(args[1],"r");
  if(file == NULL) {printf("couldn't open %s\n", args[1]); exit(0);}
  yyin = file;  // now flex reads from file
  yyparse();
  fclose(file);
}
```

There are a few differences here. The main one is that, if we encounter something that matches the VARIABLE or NUMBER token, we read what string it is that was matched, yytext, and store some corresponding value in yylval. Note also that we read the input from file instead of stdin now.

```
  // BISON file example1.y
%{
#include <cstdio>
#include <cstdlib>
#include <string>
#include <map>
using namespace std;
map<string,double> vars;   // map from variable name to value
extern int yylex();
extern void yyerror(char*);
void Div0Error(void);
void UnknownVarError(string s);
%}

%union {
  int     int_val;
  double  double_val;
  string* str_val;
}

%token <int_val>    PLUS MINUS ASTERISK FSLASH EQUALS PRINT LPAREN RPAREN SEMICOLON
%token <str_val>    VARIABLE
%token <double_val> NUMBER

%type <double_val> expression;
%type <double_val> inner1;
%type <double_val> inner2;

%start parsetree

%%

parsetree:    lines;
lines:        lines line | line;
line:         PRINT expression SEMICOLON           {printf("%lf\n",$2);}
            | VARIABLE EQUALS expression SEMICOLON {vars[*$1] = $3; delete $1;};
expression:   expression PLUS inner1               {$$ = $1 + $3;}
            | expression MINUS inner1              {$$ = $1 - $3;}
            | inner1                               {$$ = $1;};
inner1:       inner1 ASTERISK inner2               {$$ = $1 * $3;}
            | inner1 FSLASH inner2
              {if($3 == 0) Div0Error(); else $$ = $1 / $3;}
            | inner2                               {$$ = $1;};
inner2:       VARIABLE
              {if(!vars.count(*$1)) UnknownVarError(*$1); else $$ = vars[*$1]; delete $1;}
            | NUMBER                               {$$ = $1;}
            | LPAREN expression RPAREN             {$$ = $2;};

%%

void Div0Error(void) {printf("Error: division by zero\n"); exit(0);}
void UnknownVarError(string s) {printf("Error: %s does not exist!\n", s.c_str()); exit(0);}
```

Here there are some important differences:

1. **Different tokens (like VARIABLE, NUMBER) have different types**, and these types are defined in the union above. The job of flex is to set the corresponding type of yylval when it

encounters the token. For instance, if it encounters a NUMBER, it sets yylval.double_val to some double.

2. **Bison has access to these values stored in yylval using the $1, $2, ... variables.** If a rule of the grammar has three variables on the right-hand side (*e.g.* expression: expression PLUS expression), then the $1 refers to the first variable and so on. Notice that in this example $2 refers to the PLUS token, so it's useless (will always return some constant).
3. **We can set the type and value of a node of the parse tree.** In this case, we've set "expression", "inner", and "inner2" to all have type double_val. In general the type we pick must be in the union defined above. And to set the value of the current node, we use $$ = ____.

Again, to compile (this time using C++):

```
$ bison -d example1.y
$ flex example1.l
$ g++ -o parser1 lex.yy.c example0.tab.c
```

Here's an example program, "example1.myprog":

```
a = 177;
b = 99.3;
goulash = a*(a-b)/(a+b);
print goulash;
print goulash + (a/goulash)*(b/goulash);
print (goulash + a/goulash)*(b/goulash);
```

And the output:

```
$ ./parser1 example1.myprog
49.775244
56.869318
106.394074
```

OK, it works. (Check by running `python example1.myprog`.)

5 Towards A Compiler

Suppose we wanted a compiler instead of an interpreter. How would we get there? We'd need a data structure storing our parse tree; it could be a Node class with subclasses for each particular type of node on the tree, or something similar. Then we could begin with a root node, and for each successful parse (say of an arithmetic expression) we'd create a new node, whose constructor would take as arguments the particular inputs (say the values in that expression). At the end, if parsing succeeded, we'd have a data structure storing the parse tree, which we could walk through to generate code or optimize or so on. That's just one approach. Luckily, you can use as many external header files and C++ files with flex and bison as you want. In fact, the main method doesn't even need to be in a flex or bison file. So you can keep those to the lexer and parser and build your compiler separately, as long as you compile them all at once (*e.g.* g++ -o compiler lex.yy.c grammar.tab.c my_compiler.cpp).