# Flex and Bison

**Flex**

* yylex() : the scanner/lexer routine
* yytext: points to input text that matched
* yylval: variable that stores the returned token (usually a union)
* yyin: input file,
  + eg: yyin = fopen(argv[1], "r")
  + if not set, reads from stdin
* yywrap(): called at end of yyin file
  + returns 1 if no more input files
  + 0 if another input file to parse
* %option noyywrap: don’t call yywrap() (modern way)
  + yyrestart(f): read from stdio file f. if not called, yylex reads from stdin
    - resets currently read buffer
  + eg:

for(i = 1; i < argc; i++) {

FILE \*f = fopen(argv[i], "r");

if(!f) {

perror(argv[i]);

return (1);

}

yyrestart(f);

yylex();

fclose(f);

* yyterminate(): macro which returns YY\_NULL (0 by default) from the scanner
  + bison interprets 0 as end of input
* yy\_create\_buffer()
* yy\_delete\_buffer()
* yy\_switch\_to\_buffer()
* %option yylineno: tells flex to define an integer variable called yylineno and to

maintain the current line number in it

* %option case-insensitive: tells flex to build a scanner that uppercase and lowercase the same
* ECHO: print the matched token
* Named patterns eg:  
  /\* Universal Character Name \*/

UCN (\\u[0-9a-fA-F]{4}|\\U[0-9a-)

/\* float exponent \*/

EXP ([Ee][-+]?[0-9]+)

/\* integer length \*/

ILEN ([Uu](L|l|LL|ll)?|(L|l|LL|ll)[

* Each pattern is enclosed in parentheses to avoid an old lex/flex incompatibility

**Input**

* 3 main ways:
  1. set yyin to input files
  2. create and use YY\_BUFFER\_STATE
  3. Redefine YY\_INPUT
* YY\_BUFFER\_STATE: data structure to handle input
  + usually contains a FILE\* (file to be read)
  + can also be used to read from a string in memory
* Approximate default input behaviour:

YY\_BUFFER\_STATE bp;

extern FILE\* yyin;

... whatever the program does before the first call to the scanner

if(!yyin) yyin = stdin; //default input is stdin

bp = yy\_create\_buffer(yyin,YY\_BUF\_

//YY\_BUF\_SIZE defined by flex, typically 16K

yy\_switch\_to\_buffer(bp);// tell it to use the buffer we just made

yylex(); // or yyparse() or whatever calls the scanner

* When new file is assigned to yyin, YY\_NEW\_FILE is equivalent to yyrestart(yyin)
* Other ways of reading input:
  + yy\_scan\_string("string") to scan a null-terminated string
  + yy\_scan\_buffer(char\*base, size) to scan a buffer of known size
  + Can redefine the YY\_INPUT macro (newer way is YY\_BUFFER\_STATE):

#define YY\_INPUT(buf,result,max\_size) ..

result-> amount read or 0 at EOF

* <<EOF>>: pseudopattern that matches on EOF
* input(): read next char from input stream
* unput(c): push char c back to input stream

**Output**

* Default rule that copies otherwise unmatched output to yyout:J  
  .      ECHO;  
  #define ECHO fwrite( yytext, yyleng, 1, yyout )
* %option nodefault // don't add the default rule and report error (recommended)

**Start States**

* %x <state name> : exclusive start state (only match patterns marked for this state)
* %s <state name> : inclusive start state (match patterns marked for this state and those not marked for any)
* INITIAL state: always defined by flex
* Eg:  
  ...  
  %x IFILE // defining exclusive state  
  ...  
  ^"#"[ \t]\*include[ \t]\*\[\"<] { BEGIN IFILE; }  // enter IFILE state when input has #include  
  ...

<IFILE>[^ \t\n\">]+     {

{ int c;

while((c = input()) && c != '\n') ;

}

yylineno++;

if(!newfile(yytext))

yyterminate(); /\* no such file \*/

BEGIN INITIAL;

      }

**Compilation**

* Usage Steps:  
  flex fb1-1.l

$ cc lex.yy.c -lfl (-lfl : link to flex library)

* lex.yy.c: C program generated
* Sections:
  + Definitions/Options
  + list of patterns and actions
    - Longer match preferred
    - Pattern that appears first takes preference
    - literal string tokens written as quoted strings

* C code copied to generated code
  + if left blank, a tiny main created with call to yylex()

**Tokens**

* Usually we return the token as the action
  + scanner acts as a coroutine; that is, each time it returns, it remembers where it was, and on the next call it picks up where it left off
  + If action code returns, scanning resumes on the next call to yylex(); if it doesn’t return, scanning resumes immediately.
* Token has two parts:
  + token
  + token's value (arbitrary numbers, bison starts tokens at 258, 0 means end of file)
  + bison creates .h with definitions of token numbers

* If calling the lexer from the parser (bison), don't need to define token values. Bison will do that in the \*.tab.h
  + just include the \*.tab.h in the first section of y file
  + Also, don’t need main function with yylex() call, as bison will call it
* [^ \t\n\r\f\v]+ : match any characters other than whitespace (^ = negation)
* An action consisting solely of a vertical bar tells flex that the action for this rule is the same as the action for the next rule

**Bison**

**Tokens**

* Union to define the types:

%union {

struct ast \*a;

double d;

}  
yylval (flex) will now be a union

* need to tell bison what symbols have what types of values by putting the appropriate name from the union in angle brackets (< >):  
  %token <d> NUMBER  
  ...   
  %type <a> exp factor term
* don’t have to declare a type for a token or declare a nonterminal at all if you don’t use the symbol’s value
* error if you attempt to use the value of a symbol that doesn’t have an assigned type

**Grammar**

* any rule without explicit action code gets the default action $$ = $1; (error if LHS and RHS types don’t match)

* Context Free Grammar(CFG), Backus-Naur Form(BNF):

<exp> ::= <factor>

| <exp> + <factor>

<factor> ::= NUMBER

| <factor> \* NUMBER

::= can be read “is a” or “becomes,” and | is “or,” another way to create a branch of the same kind.

* all tokens are considered to be symbols, but there are also symbols that are not tokens
* Bison example (Simplified BNF):

/\* simplest version of calculator \*/

%{

#include <stdio.h>

%}

/\* declare tokens \*/

%token NUMBER

%token ADD SUB MUL DIV ABS

%token EOL

%%

calclist: /\* nothing \*/ matches at beginning of input

| calclist exp EOL { printf("= %d\n", $1); } EOL is end of an expression

;

exp: factor default $$ = $1

| exp ADD factor { $$ = $1 + $3; }

| exp SUB factor { $$ = $1 - $3; }

;

factor: term default $$ = $1

| factor MUL term { $$ = $1 \* $3; }

| factor DIV term { $$ = $1 / $3; }

;

term: NUMBER default $$ = $1

| ABS term { $$ = $2 >= 0? $2 : - $2; }

* Any symbols not declared as tokens have to appear on the left side of at least one rule in the program. (If a symbol neither is a token nor appears on the left side of a rule, it’s like an unreferenced variable in a C program. It doesn’t hurt anything, but it probably means the programmer made a mistake.)
* semicolon marks end of a rule

* symbol on the left side of the first rule is the *start symbol*
* $$ : the value of the target symbol (the one to the left of the colon)
* $1, $2,…: values on the right side of the colon
* values of tokens are whatever was in yylval when the scanner returned the token
* common two-rule recursive idiom: the first rule is empty and matches nothing; the second adds an item to the list
* rules with a single symbol on the right side are syntactic glue to put the grammar together
* Can have a single quoted character as a token, without giving it a name, eg:

exp: factor

| exp '+' factor { $$ = newast('+', $1,$3); }

| exp '-' factor { $$ = newast('-', $1,$3);}

;

* Specifying Operator Precedence and Associativity
  + 2 ways:
    1. Implicitly: use separate non-terminal symbols for each precedence level
    2. Explicitly in the definition section. Eg:  
       %left '+' '-'  //left associative, lowest precedence

%left '\*' '/'  //left associative, next higher precedence

%nonassoc '|' UMINUS //no associativity, highest precedence  
  
usually the faster option as it is smaller, with fewer rules to reduce

* Bison assigns each rule the precedence of the rightmost token on the right hand side
* On a shift/reduce conflict, consults the table of precedence  
  If all the rules involved in the conflict have a precedence assigned, use it to resolve the conflict
* Use precedence only for (Advisable to fix grammar in all other cases):
  1. expression grammars
  2. dangling else in if/then/else constructs

* Use one token for several syntactically similar operators helps keep down the size of the grammar, eg:  
  %token IF THEN ELSE WHILE DO LET

**Compilation**

* Makefile example when using bison and flex together:

myprog myprog.l fb1-5.y

bison -d myprog.y // -d = definitions. this creates myprog.tab.c and myprog.tab.h

flex myprog.l // creates lex.yy.c

cc -o $@ myprog.tab.c lex.yy.c -lfl

* Bison automatically names its generated C file to match the .y file, but flex always calls its C file lex.yy.c unless you tell it otherwise  
  eg:

fb3-1: fb3-1.l fb3-1.y fb3-1.h

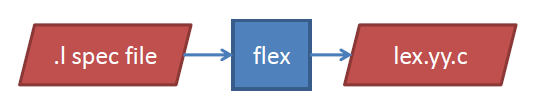
bison -d fb3-1.y

flex -ofb3-1.lex.c fb3-1.l

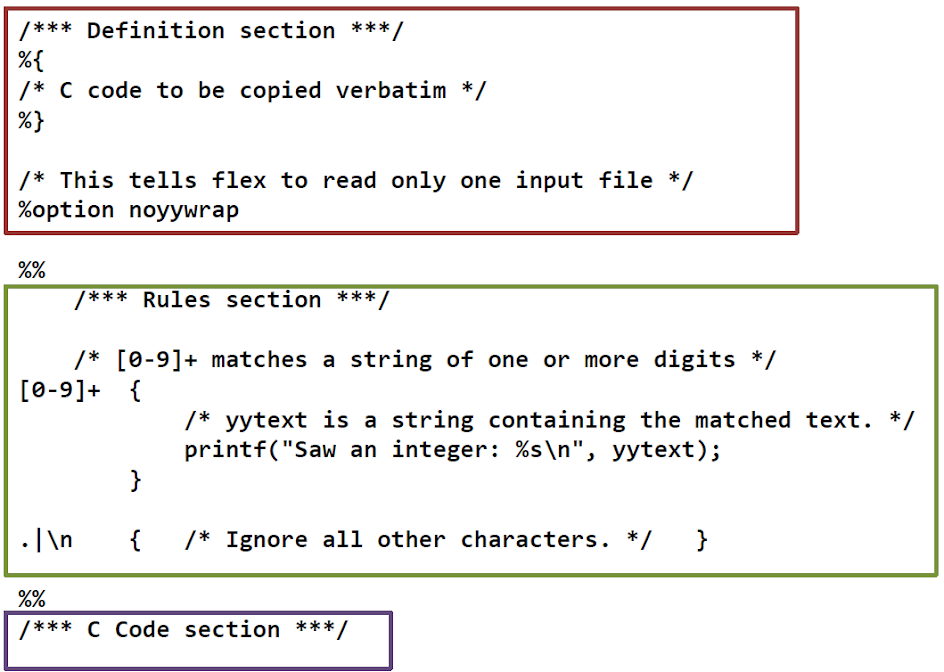
cc -o $@ fb3-1.tab.c fb3-1.lex.c fb3-1funcs.c



* Flex/Bison have mechanisms for error handling and recovery
* Lex/Flex
  + Tool to generate lexical analyzers
  + .l spec file
  + Spec file used to generate tokenizer/scanner



* Tokenizer (lex.yy.c): reads  an input file, generates series of tokens for the *parser*
* Spec file example/layout:



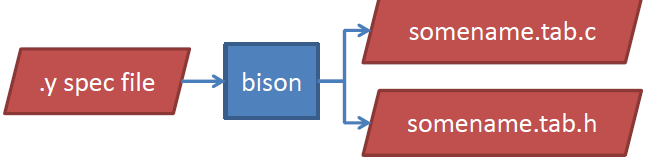
* Flex Rule
  + Matches text input via regular expressions
  + Returns token type
  + Format:  
    REGEX {  
             /\*Code\*/  
             return TOKEN\_TYPE;  
          }
  + Matches token with *longest match*
  + Selects *first applicable rule*
    - Eg:  
      input: "post"  
      Rule1: "post"  {  printf("Hello"); }  
      Rule1: [a-zA-Z]+  {  printf("world");  }  
        
      prints "Hello" not "world"
* **Example** (yytext contains the matched string)  
    
  [0-9]+ {   
              /\*Code\*/   
              yylval.dval = atof(yytext); // store the number  
              return NUMBER; // return type declared in .y file  
         }   
    
  [A-Za-z]+ {   
              /\*Code\*/   
              struct symtab \*sp = symlook(yytext); // store the text  
              yylval.symp = sp;  
              return WORD; //return token type, declared in .y file  
            }  
    
   .        { return yytext[0]; } //return the character
* 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.
* when Flex finds something that doesn't match any of the regexs it echos it to STDOUT
  + to get rid of it for now you could just add . ; to the token section -- that will match any one character (the ".") and do nothing with it (the empty C statement ";").
* Handling carriage returns:

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

* Flex reads its input from a global pointer to a C FILE variable called yyin, which is set to STDIN by default
  + Eg:

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

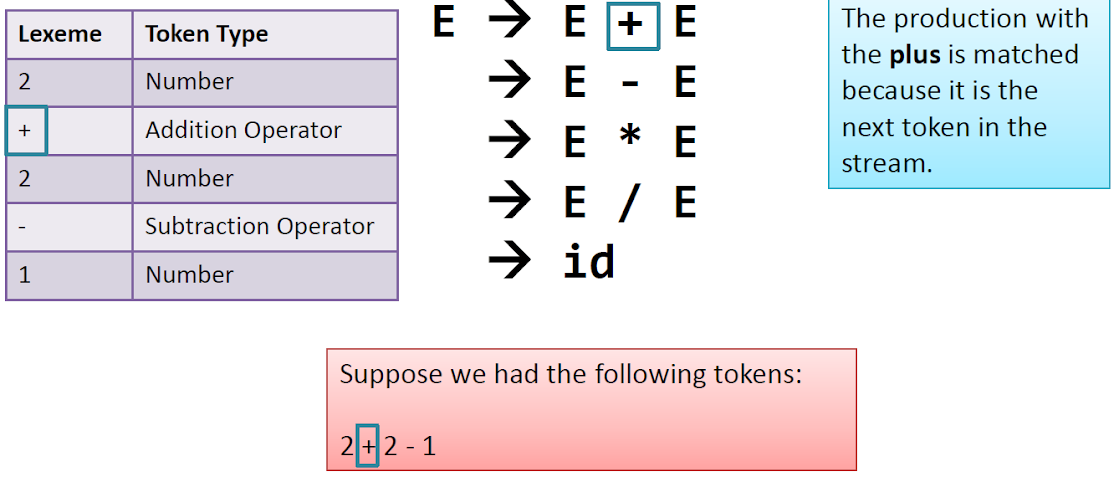
* Yacc/Bison
  + Tool to generate parsers
  + Uses non-ambiguous context-free (as opposed to "natural language") grammars
  + .y spec file
  + Spec file used to generate *parser*



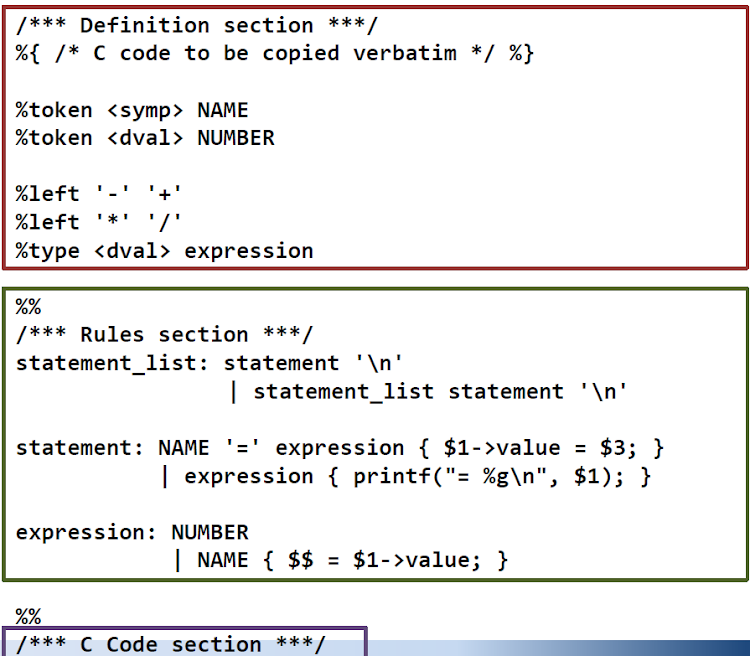
* Grammar:

A set of formation rules for strings in a formal language. The rules describe how to form strings from the language's alphabet (tokens) that are valid according to the language's syntax.

* Example



* Spec file (.y) example/layout



* Even though Flex figures out and returns tokens by type, defining those types is done in the Bison file

* Definition Section:

%{

    /\* C code to be copied verbatim \*/

%}

%token <symp> NAME    //declaration of tokens: %token <TYPE> NAME

%token <dval> NUMBER

%left '-' '+'   //operator precedence and associativity

%left '\*' '/'   // '\*' and '/' have higher precedence than '-' and '+'  
                // associativity options: %left, %right, %nonassoc

%type <dval> expression  //defined non-terminal name (LHS of productions)

* // Declare stuff from Flex that Bison needs to know about:  
    extern int yylex();  
    extern int yyparse();  
    extern 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".
  + by default yystype is a typedef of "int"
  + for non-trivial projects, tokens could be of any arbitrary data type.
  + override yystype's default typedef to be a C union instead.
    - Unions can hold all of the types of tokens that Flex could return
    - Bison implements this mechanism with the %union directive:  
      %union {  
        int ival;  
        float fval;  
        char \*sval;  
      }
* Define the "terminal symbol" token types using (in CAPS by convention), and associate each with a field of the %union:  
  %token <ival> INT  
  %token <fval> FLOAT  
  %token <sval> STRING

* Once the terminal symbols are declared in the .y file, we run *bison first* with the option **-d** to create a .h file
* This .h file can be included in the .l flex file:

snazzle.l  
%{  
  #include <cstdio>

#include "snazzle.tab.h"  // to get the token types from Bison

%}  
%%  
[ \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 strdup yytext because Flex will change it for the next token.  
  /\* Note that this memory must be freed somewhere, so that's why we call  
   free() in the Bison section (see grammar section below).  (Aside: we use free() instead of delete because strdup is a C function that uses malloc, not a C++ function that uses new.)\*/  
  yylval.sval = strdup(yytext);  
  return STRING;  
}  
%%



* Rules Section (Grammar)

statement\_list: statement '\n' //statement

              | statement\_list statement '\n'  //statement list followed by statement

statement: NAME '=' expression { $1->value = $3; }  //assignment statement

           | expression { printf("= %g\n", $1); }   //simple expression

expression: NUMBER

            | NAME { $$ = $1->value; }

Executable statements, run when the rule is matched

$num correspond to tokens in the production  
eg:  NAME '=' expression { $1->value = $3; }, 'NAME' is $1, '=' is $2, 'expression' is $3

* Example

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; free($1);  
    }  
  | INT            {  
      cout << "bison found an int: " << $1 << endl;  
    }  
  | FLOAT          {  
      cout << "bison found a float: " << $1 << endl;  
    }  
  | STRING         {  
      cout << "bison found a string: " << $1 << endl; free($1);  
    }  
  ;  
%%

* The above grammar is right recursive => it'll print the output in reverse (needs to match the full rule)
  + It will also overflow Bison's internal stack for large inputs
* Better option: Left recursive grammar:  
  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; free($2);  
      }**  
    | INT {  
        cout << "bison found an int: " << $1 << endl;  
      }  
    | FLOAT {  
        cout << "bison found a float: " << $1 << endl;  
      }  
    | STRING {  
        cout << "bison found a string: " << $1 << endl; free($1);  
      }  
    ;  
  %%
* C Code Section
  + Example

int main(int, char\*\*) {  
  // 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:  
 yyparse();  
}

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

* Terminal Symbols
  + Smallest unit of the grammar
  + Boundary b/w flex and bison
* Source of snazzle example: [http://aquamentus.com/flex\_](http://aquamentus.com/flex_bison.html)
* A Complete Realistic Example
  + snazzle.l

%{  
  #include "snazzle.tab.h"  
%}  
%%  
[ \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]+   {  
  yylval.sval = strdup(yytext);  
  return STRING;  
}  
.              ;  
%%

* snazzle.y

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

// stuff from flex that bison needs to know about:  
  extern int yylex();  
  extern int yyparse();  
  extern FILE \*yyin;  
  
  void yyerror(const char \*s);  
%}

%union {  
  int ival;  
  float fval;  
  char \*sval;  
}

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

// define the "terminal symbol"  
%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!" << 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;  
      free($2);  
    }  
 ;  
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;  
      free($5);  
    }  
  ;  
footer:  
  END  
  ;**

%%

int main(int, char\*\*) {  
  // 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:  
  yyparse();  
}

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

* Compile and Run

% 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!  
%

* The Makefile

% 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  
%

* Some Tweaks  
  + *To recognize carriage returns:*
    - Recognize the '\n' token in flex (in the rules section):  
      **\n       return ENDL;**
    - Add the token to the .y file (bison definition section):  
      %token END **ENDL**
    - Add relevant rules to the bison grammar, so the .y file becomes:

snazzle.y  
%{  
  #include <cstdio>  
  #include <iostream>  
  using namespace std;

// stuff from flex that bison needs to know about:  
  extern int yylex();  
  extern int yyparse();  
  extern FILE \*yyin;  
  
  void yyerror(const char \*s);  
%}

%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!" << 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;  
      free($2);  
    }**  
  ;  
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;  
      free($5);  
    }**  
  ;  
footer:  
  **END ENDLS**  
  ;  
**ENDLS:  
  ENDLS ENDL  
  | ENDL ;**  
%%

* *Adding line numbers*
  + Add a variable to keep track of line numbers (.l file)
  + Add a rule (.l file) to increment it whenever the \n is seen
  + define the line number variable in the .y file definition section
  + In the error handling in the .y file, can print the line number
* *Additional tips:* [*http://aquamentus.com/flex\_*](http://aquamentus.com/flex_bison.html#9)
  + directly returning terminal characters
  + actions before the end of the grammar (bison)
  + whitespaces in .y and .l files
  + prefixing identifiers to avoid name collisions with other programs using flex/bison
  + renaming output files to avoid collisions with other programs
* *Start states in flex (allows user to match a rule only if the program is in a specific state):*
  + <FOO>bar     ;  // we're in state FOO and we saw "bar"
  + State names are user defined
  + States must be declared in flex's control section:  
    %x FOO
  + To get into a state (in any code block):  
    bar      { BEGIN(FOO); }  // we saw a "bar", so go into the "FOO" state
  + To jump back to the 'initial' state (pre-defined default)  
    bas      { BEGIN(INITIAL); }  // jump back to normal pattern matching
  + BEGIN can be made conditional:

bar    { do\_something(); BEGIN(FOO); do\_something\_else(); }  
..  
bar    { if (some\_value == 42) BEGIN(FOO); }

* Using this concept to recognize multiline comments (/\* … \*/):  
    
  \/\*                    { // 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 inside of comments!  
  <COMMENTS>.       ;   // ignore every other character while we're in this state
* Can have a flex match pertaining to multiple states:

<MYSTATE,ANOTHERSTATE>foo { ... }

* [http://aquamentus.com/flex\_](http://aquamentus.com/flex_bison.html#18)

* *Reading gzipped input :* [*http://aquamentus.com/flex\_*](http://aquamentus.com/flex_bison.html#19)