flex & bison ch01 tutorial

Source: *flex & bison*. John Levine. © 2009 O'Reilly Media, Inc.

The book is available online via UTD ebook safari

flex & bison

flex & bison. Ch01. John Levine. © 2009 O'Reilly Media, Inc.

- Flex and Bison are tools for building programs that handle structured input.
- They were originally tools for building compilers, but they have proven to be useful in many other areas.
- Lexical Analysis and Parsing
- The earliest compilers back in the 1950s used utterly ad hoc techniques to analyze the syntax of the source code of programs they were compiling.
- During the 1960s, the field got a lot of academic attention, and by the early 1970s, syntax analysis was a well-understood field.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Bison is descended from yacc, a parser generator written between 1975 and 1978 by Stephen C. Johnson at Bell Labs.
- As its name, short for “yet another compiler compiler” suggests, many people were writing parser generators at the time.
- Johnson’s tool combined a firm theoretical foundation from parsing work by D. E. Knuth, which made its parsers extremely reliable, and a convenient input syntax.
- These made it extremely popular among users of Unix systems, although the restrictive license under which Unix was distributed at the time limited its use outside of academia and the Bell System.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- In about 1985, Bob Corbett, a graduate student at the University of California, Berkeley, reimplemented yacc using somewhat improved internal algorithms, which evolved into Berkeley yacc.
- Since his version was faster than Bell’s yacc and was distributed under the flexible Berkeley license, it quickly became the most popular version of yacc.
- Richard Stallman of the Free Software Foundation (FSF) adapted Corbett’s work for use in the GNU project, where it has grown to include a vast number of new features as it has evolved into the current version of bison.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- In 1975, Mike Lesk and summer intern Eric Schmidt wrote lex, a lexical analyzer generator, with most of the programming being done by Schmidt.
- They saw it both as a standalone tool and as a companion to Johnson's yacc.
- Lex also became quite popular, despite being relatively slow and buggy.
- Schmidt nonetheless went on to have a fairly successful career in the computer industry where he is now the CEO of Google.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Bison is now maintained as a project of the FSF and is distributed under the GNU Public License.
- In about 1987, Vern Paxson of the Lawrence Berkeley Lab took a version of lex written (in ratfor - an extended Fortran popular at the time) and then translated it into C, calling it flex, for "Fast Lexical Analyzer Generator."
- Since it was faster and more reliable than AT&T lex and, like Berkeley yacc, available under the Berkeley license, it has completely supplanted the original lex.
- Flex is now a SourceForge project, still under the Berkeley license.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Lexical Analysis and Parsing
- One of the key insights was to break the job into two parts:

  lexical analysis (also called lexing or scanning) and

  syntax analysis (or parsing).

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Lexical Analysis and Parsing
- Roughly speaking, scanning divides the input into meaningful chunks, called tokens, and parsing figures out how the tokens relate to each other. For example, consider this snippet of C code:

  ```
  alpha = beta + gamma ;
  ```
- A scanner divides this into the tokens alpha, equal sign, beta, plus sign, gamma, and semicolon.
- Then the parser determines that beta + gamma is an expression, and that the expression is assigned to alpha.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- **Regular Expressions and Scanning**
- Scanners generally work by looking for patterns of characters in the input.
- For example, in a C program, an integer constant is a string of one or more digits, a variable name is a letter followed by zero or more letters or digits, and the various operators are single characters or pairs of characters.
- A straightforward way to describe these patterns is *regular expressions*, often shortened to *regex* or *regexp*.
- These are the same kind of patterns that the editors ed and vi and the search program egrep use to describe text to search for.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- **Regular Expressions and Scanning**
- A flex program basically consists of a list of regexps with instructions about what to do when the input matches any of them, known as *actions*.
- A flex-generated scanner reads through its input, matching the input against all of the regexps and doing the appropriate action on each match.
- Flex translates all of the regexps into an efficient internal form that lets it match the input against all the patterns simultaneously, so it’s just as fast for 100 patterns as for one.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Our First Flex Program
- Unix systems come with a word count program, which reads through a file and reports the number of lines, words, and characters in the file.
- Flex lets us write wc in a few dozen lines, shown in Example 1-1.
- Example 1-1. Word count fb1-1.l

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- Much of this program should look familiar to C programmers, since most of it is C.
- A flex program consists of three sections, separated by %% lines.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- The first section contains declarations and option settings.
- The second section is a list of patterns and actions.
- The third section is C code that is copied to the generated scanner, usually small routines related to the code in the actions.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- In the declaration section
- code inside of %{ and %} is copied through verbatim near the beginning of the generated C source file.
- In this case it just sets up variables for lines, words, and characters.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- In the second section, each pattern is at the beginning of a line, followed by the C code to execute when the pattern matches.
- The C code can be one statement or possibly a multiline block in braces, { }.
- Each pattern must start at the beginning of the line, since flex considers any line that starts with whitespace to be code to be copied into the generated C program.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- In this program, there are only three patterns.
- The first one, [a-zA-Z]+, matches a word.
- The characters in brackets, known as a character class, match any single upper- or lowercase letter, and the + sign means to match one or more of the preceding thing, which here means a string of letters or a word.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- In this program, there are only three patterns.
- The first one, [a-zA-Z]+, matches a word.
- The action code updates the number of words and characters seen.
- In any flex action, the variable yytext is set to point to the input text that the pattern just matched.
- In this case, all we care about is how many characters it was so we can update the character count appropriately.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- In this program, there are only three patterns.
- The second pattern, \n, just matches a new line.
- The action updates the number of lines and characters.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- In this program, there are only three patterns.
- The final pattern is a dot, which is regex-ese for any character.
- It's similar to a ? in shell scripts.
- The action updates the number of characters. And that's all the patterns we need.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
/* fb1-1.l - just like Unix wc */
%{
        int chars = 0;
        int words = 0;
        int lines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
        yylex();
        printf("%8d%8d%8d\n", lines, words, chars);
}
```

- The C code at the end is a main program that calls yylex(), the name that flex gives to the scanner routine, and then prints the results.
- In the absence of any other arrangements, the scanner reads from the standard input.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- First we tell flex to translate our program, and in classic Unix fashion since there are no errors, it does so and says nothing.

- Then we compile lex.yy.c, the C program it generated; link it with the flex library, -lfl.
- And then run it, and type a little input for it to count.

To run it

```
$ flex fb1-1.l
$ cc lex.yy.c –lfl

$ ./a.out
The boy stood on the burning deck
shelling peanuts by the peck
^D
2 12 63
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- The actual wc program uses a slightly different definition of a word, a string of non-whitespace characters.
- Once we look up what all the whitespace characters are, we need only replace the line that matches words with one that matches a string of non-whitespace characters:

```
[^ \t\n\r\f\v]+        { words++; chars += strlen(yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
[^ \t\n\r\f\v]+        { words++; chars += strlen(yytext); }
```

- The ^ at the beginning of the character class means to match any character other than the ones in the class
- The + once again means to match one or more of the preceding patterns.
- This demonstrates one of flex's strengths—it's easy to make small changes to patterns and let flex worry about how they might affect the generated code.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Some applications are simple enough that you can write the whole thing in flex, or in flex with a little bit of C.
- For example, Example 1-2 shows the skeleton of a translator from English to American. (Example 1-2. fb1-2.l)

```
/* English -> American */
%%
"colour"        { printf("color"); }
"flavour"       { printf("flavor"); }
"clever"        { printf("smart"); }
"smart"         { printf("elegant"); }
"conservative" { printf("liberal"); }
 /* … lots of other words …  */
.               { printf("%s", yytext); }
%%
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- It reads through its input, printing the American version when it matches an English word and passing everything else through.
- One may use flex to generate a scanner that divides the input into tokens that are then used by other parts of your program.

```
/* English -> American */
%%
"colour"          { printf("color"); }
"flavour"         { printf("flavor"); }
"clever"          { printf("smart"); }
"smart"           { printf("elegant"); }
"conservative" { printf("liberal"); }
 /* … lots of other words …  */
.                 { printf("%s", yytext); }
%%
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- The first program we'll write using both flex and bison is a desk calculator.
- First we'll write a scanner, and then we'll write a parser and splice the two of them together.
- To keep things simple, we'll start by recognizing only integers, four basic arithmetic operators, and a unary absolute value operator (Example 1-3).
- Example 1-3. A simple flex scanner fb1-3.l

```
/* recognize tokens for the calculator */
%%
"+"    { printf("PLUS\n"); }
"-"    { printf("MINUS\n"); }
"*"    { printf("TIMES\n"); }
"/"    { printf("DIVIDE\n"); }
"|"    { printf("ABS\n"); }
[0-9]+ { printf("NUMBER %s\n", yytext); }
\n     { printf("NEWLINE\n"); }
[ \t]  { }
.    { printf("Mystery character %s\n", yytext); }
%%
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- The first five patterns are literal operators, written as quoted strings, and the actions, for now, just print a message saying what matched.
- The quotes tell flex to use the strings as is, rather than interpreting them as regular expressions.

```
/* recognize tokens for the calculator */
%%
"+"      { printf("PLUS\n"); }
"-"      { printf("MINUS\n"); }
"*"      { printf("TIMES\n"); }
"/"      { printf("DIVIDE\n"); }
"|"      { printf("ABS\n"); }
[0-9]+ { printf("NUMBER %s\n", yytext); }
\n    { printf("NEWLINE\n"); }
[ \t]  { }
.    { printf("Mystery character %s\n", yytext); }
%%
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- The sixth pattern matches an integer.
- The bracketed pattern [0-9] matches any single digit, and the following + sign means to match one or more.
- The action prints out the string that's matched, using the pointer yytext that the scanner sets after each match.

```
/* recognize tokens for the calculator */
%%
"+"      { printf("PLUS\n"); }
"-"      { printf("MINUS\n"); }
"*"      { printf("TIMES\n"); }
"/"      { printf("DIVIDE\n"); }
"|"      { printf("ABS\n"); }
[0-9]+  { printf("NUMBER %s\n", yytext); }
\n       { printf("NEWLINE\n"); }
[ \t]    { }
.    { printf("Mystery character %s\n", yytext); }
%%
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- The seventh pattern matches a newline character, represented by the usual C \n sequence.
- The eighth pattern ignores whitespace. It matches any single space or tab (\t), and the empty action code does nothing.
- The final pattern is the catchall to match anything the other patterns didn't. Its action code prints a suitable complaint.

```
/* recognize tokens for the calculator */
%%
"+"     { printf("PLUS\n"); }
"-"     { printf("MINUS\n"); }
"*"     { printf("TIMES\n"); }
"/"     { printf("DIVIDE\n"); }
"|"     { printf("ABS\n"); }
[0-9]+  { printf("NUMBER %s\n", yytext); }
\n      { printf("NEWLINE\n"); }
[ \t]   { }
.    { printf("Mystery character %s\n", yytext); }
%%
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- In this simple flex program, there's no C code in the third section. The flex library (-lfl) provides a tiny main program that just calls the scanner.

```
$ flex fb1-3.l
$ cc lex.yy.c -lfl
$ ./a.out
12+34
NUMBER 12
PLUS
NUMBER 34
NEWLINE
 5 6 / 7q
NUMBER 5
NUMBER 6
DIVIDE
NUMBER 7
Mystery character q
NEWLINE
^D
```

```
/* recognize tokens for the calculator */
%%
"+"     { printf("PLUS\n"); }
"-"     { printf("MINUS\n"); }
"*"     { printf("TIMES\n"); }
"/"     { printf("DIVIDE\n"); }
"|"     { printf("ABS\n"); }
[0-9]+  { printf("NUMBER %s\n", yytext); }
\n   { printf("NEWLINE\n"); }
[ \t]  { }
.    { printf("Mystery character %s\n", yytext); }
%%
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- First we run flex, which translates the scanner into a C program called lex.yy.c, then we compile the C program, and finally we run it.
- The output shows that it recognizes numbers as numbers, it recognizes operators as operators, and the q in the last line of input is caught by the catchall pattern at the end.
- The last ^D is a Unix/Linux end-of-file character.
- On Windows you'd type ^Z.

```
$ flex fb1-3.l
$ cc lex.yy.c -lfl
$ ./a.out
12+34
NUMBER 12
PLUS
NUMBER 34
NEWLINE
 5 6 / 7q
NUMBER 5
NUMBER 6
DIVIDE
NUMBER 7
Mystery character q
NEWLINE
^D
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- The Scanner as Coroutine
- Most programs with flex scanners use the scanner to return a stream of tokens that are handled by a parser.
- Each time the program needs a token, it calls yylex(), which reads a little input and returns the token.
- When it needs another token, it calls yylex() again.
- The 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.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- The Scanner as Coroutine
- Within the scanner, when the action code has a token ready, it just returns it as the value from yylex().
- The next time the program calls yylex(), it resumes scanning with the next input characters.
- Conversely, if a pattern doesn't produce a token for the calling program and doesn't return, the scanner will just keep going within the same call to yylex(), scanning the next input characters.
- This incomplete snippet shows two patterns that return tokens, one for the + operator and one for a number, and a whitespace pattern that does nothing, thereby ignoring what it matched.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
"+"    { return ADD; }
[0-9]+ { return NUMBER; }
[ \t] { /* ignore whitespace */ }
```

- A whitespace pattern does nothing, thereby ignoring what it matched.
- This apparent casualness about whether action code returns often confuses new flex users, but the rule is actually quite simple:

  If action code returns, scanning resumes on the next call to yylex();
  If it doesn't return, scanning resumes immediately.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Tokens and Values
- When a flex scanner returns a stream of tokens, each token actually has two parts, the token and the token's value.
- The token is a small integer.
- The token numbers are arbitrary, except that token zero always means end-of-file.
- When bison creates a parser, bison assigns the token numbers automatically starting at 258 (this avoids collisions with literal character tokens, discussed later) and creates a .h with definitions of the tokens numbers.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Tokens and Values
- But for now, we'll just define a few tokens by hand:

  NUMBER = 258,
  ADD = 259,
  SUB = 260,
  MUL = 261,
  DIV = 262,
  ABS = 263,
  EOL = 264 end of line
- Actually, it's the list of token numbers that bison will create.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Tokens and Values
- A token's value identifies which of a group of similar tokens this one is.
- In our scanner, all numbers are NUMBER tokens, with the value saying what number it is.
- When parsing more complex input with names, floating-point numbers, string literals, and the like, the value says which name, number, literal, or whatever, this token is.
- Our first version of the calculator's scanner, with a small main program for debugging, is in Example 1-4.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- Example 1-4. Calculator scanner fb1-4.l

```
/* recognize tokens for the calculator */
%{
  enum yytokentype {
    NUMBER = 258,
    ADD = 259,
    SUB = 260,
    MUL = 261,
    DIV = 262,
    ABS = 263,
    EOL = 264
  };
  int yylval;
%}

%%
"+"    { return ADD; }
"-"    { return SUB; }
"*"    { return MUL; }
"/"    { return DIV; }
"|"    { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n     { return EOL; }
[ \t]  { /* ignore whitespace */ }
.      { printf("Mystery character %c\n", *yytext); }

%%

main(int argc, char **argv)
{
  int tok;

  while(tok = yylex()) {
    printf("%d", tok);
    if(tok == NUMBER) printf(" = %d\n", yylval);
    else printf("\n");
  }
}
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

```
%{
  enum yytokentype {
    NUMBER = 258,
    ADD = 259,
    SUB = 260,
    MUL = 261,
    DIV = 262,
    ABS = 263,
    EOL = 264
  };
  int yylval;
%}
```

- We define the token numbers in a C enum.
- Then we make yylval, the variable that stores the token value, an integer, which is adequate for the first version of our calculator.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- The list of patterns is the same as in the previous example, but the action code is different.
- For each of the tokens, the scanner returns the appropriate code for the token.
- For numbers, it turns the string of digits into an integer and stores it in yylval before returning.
- The pattern that matches whitespace doesn't return, so the scanner just continues to look for what comes next.

```
"+"    { return ADD; }
"-"    { return SUB; }
"*"    { return MUL; }
"/"    { return DIV; }
"|"    { return ABS; }
[0-9]+ { yylval = atoi(yytext);
                  return NUMBER; }
\n    { return EOL; }
[ \t]  { /* ignore whitespace */ }
.     { printf("Mystery character %c\n", *yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- For testing only, a small main program calls yylex(), prints out the token values, and, for NUMBER tokens, also prints yylval.

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
^D
```

```
enum yytokentype {
   NUMBER = 258,
   ADD = 259,
   SUB = 260,
   MUL = 261,
   DIV = 262,
   ABS = 263,
   EOL = 264
};
```

```
"+"       { return ADD; }
"-"       { return SUB; }
"*"       { return MUL; }
"/"       { return DIV; }
"|"       { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n    { return EOL; }
[ \t]  { /* ignore whitespace */ }
.     { printf("Mystery character %c\n", *yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- For testing only, a small main program calls yylex(), prints out the token values, and, for NUMBER tokens, also prints yylval.

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
^D
```

```
enum yytokentype {
   NUMBER = 258,
   ADD = 259,
   SUB = 260,
   MUL = 261,
   DIV = 262,
   ABS = 263,
   EOL = 264
};
```

```
"+"       { return ADD; }
"-"       { return SUB; }
"*"       { return MUL; }
"/"       { return DIV; }
"|"       { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n    { return EOL; }
[ \t]  { /* ignore whitespace */ }
.     { printf("Mystery character %c\n", *yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- For testing only, a small main program calls yylex(), prints out the token values, and, for NUMBER tokens, also prints yylval.

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
^D
```

```
enum yytokentype {
   NUMBER = 258,
   ADD = 259,
   SUB = 260,
   MUL = 261,
   DIV = 262,
   ABS = 263,
   EOL = 264
};
```

```
"+"       { return ADD; }
"-"       { return SUB; }
"*"       { return MUL; }
"/"       { return DIV; }
"|"       { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n    { return EOL; }
[ \t]  { /* ignore whitespace */ }
.     { printf("Mystery character %c\n", *yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- For testing only, a small main program calls yylex(), prints out the token values, and, for NUMBER tokens, also prints yylval.

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
^D
```

```
enum yytokentype {
   NUMBER = 258,
   ADD = 259,
   SUB = 260,
   MUL = 261,
   DIV = 262,
   ABS = 263,
   EOL = 264
};
```

```
"+"       { return ADD; }
"-"       { return SUB; }
"*"       { return MUL; }
"/"       { return DIV; }
"|"       { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n    { return EOL; }
[ \t]  { /* ignore whitespace */ }
.     { printf("Mystery character %c\n", *yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- For testing only, a small main program calls yylex(), prints out the token values, and, for NUMBER tokens, also prints yylval.

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
^D
```

```
enum yytokentype {
   NUMBER = 258,
   ADD = 259,
   SUB = 260,
   MUL = 261,
   DIV = 262,
   ABS = 263,
   EOL = 264
 };
```

```
"+"      { return ADD; }
"-"      { return SUB; }
"*"      { return MUL; }
"/"      { return DIV; }
"|"      { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n    { return EOL; }
[ \t]  { /* ignore whitespace */ }
.     { printf("Mystery character %c\n", *yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- For testing only, a small main program calls yylex(), prints out the token values, and, for NUMBER tokens, also prints yylval.

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
^D
```

```
enum yytokentype {
   NUMBER = 258,
   ADD = 259,
   SUB = 260,
   MUL = 261,
   DIV = 262,
   ABS = 263,
   EOL = 264
 };
```

```
"+"      { return ADD; }
"-"      { return SUB; }
"*"      { return MUL; }
"/"      { return DIV; }
"|"      { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n    { return EOL; }
[ \t]  { /* ignore whitespace */ }
.     { printf("Mystery character %c\n", *yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

- For testing only, a small main program calls yylex(), prints out the token values, and, for NUMBER tokens, also prints yylval.

```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
^D
```

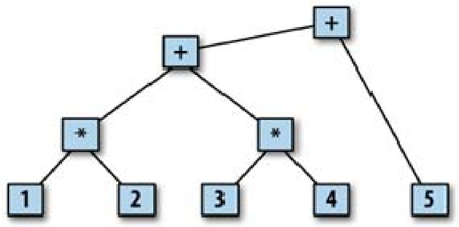
```
enum yytokentype {
   NUMBER = 258,
   ADD = 259,
   SUB = 260,
   MUL = 261,
   DIV = 262,
   ABS = 263,
   EOL = 264
  };
```

```
"+"      { return ADD; }
"-"      { return SUB; }
"*"      { return MUL; }
"/"      { return DIV; }
"|"      { return ABS; }
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
\n    { return EOL; }
[ \t]  { /* ignore whitespace */ }
.    { printf("Mystery character %c\n", *yytext); }
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

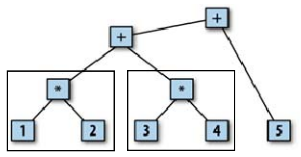
- Grammars and Parsing
- The parser’s job is to figure out the relationship among the input tokens.
- A common way to display such relationships is a parse tree.
- For example, under the usual rules of arithmetic, the arithmetic expression 1 * 2 + 3 * 4 + 5 would have the parse tree in Figure 1-1.



flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

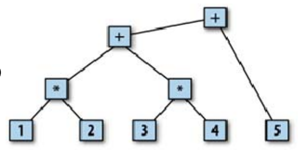


- Multiplication has higher precedence than addition, so the first two expressions are 1 * 2 and 3 * 4.
- Then those two expressions are added together, and that sum is then added to 5.
- Each branch of the tree shows the relationship between the tokens or subtrees below it.
- The structure of this particular tree is quite simple and regular with two descendants under each node (that's why we use a calculator as the first example), but any bison parser makes a parse tree as it parses its input.
- In some applications, it creates the tree as a data structure in memory for later use. In others, the tree is just implicit in the sequence of operations the parser does.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

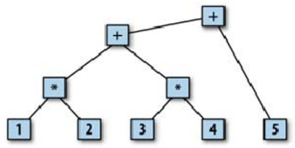


- BNF Grammars
- In order to write a parser, we need some way to describe the rules the parser uses to turn a sequence of tokens into a parse tree.
- The most common kind of language that computer parsers handle is a context-free grammar (CFG).
- The standard form to write down a CFG is Backus-Naur Form (BNF), created around 1960 to describe Algol 60 and named after two members of the Algol 60 committee.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



- BNF Grammars
- Here's BNF for simple arithmetic expressions enough to handle 1 * 2 + 3 * 4 + 5:

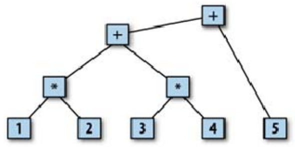
```
<exp> ::= <factor>
        | <exp> + <factor>
<factor> ::= NUMBER
        | <factor> * NUMBER
```

- Each line is a rule that says how to create a branch of the parse tree.
- In BNF, ::= can be read "is a" or "becomes," and | is "or," another way to create a branch of the same kind.
- The name on the left side of a rule is a symbol or term. By convention, all tokens are considered to be symbols, but there are also symbols that are not tokens.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



- Here's BNF for simple arithmetic expressions (e.g. 1 * 2 + 3 * 4 + 5):

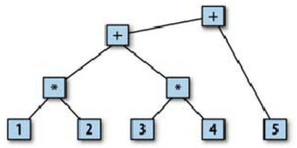
```
<exp> ::= <factor>
        | <exp> + <factor>
<factor> ::= NUMBER
        | <factor> * NUMBER
```

- Useful BNF is invariably quite recursive, with rules that refer to themselves directly or indirectly.
- These simple rules can match an arbitrarily complex sequence of additions and multiplications by applying them recursively.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

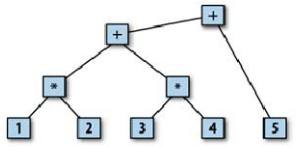


- Bison's Rule Input Language
- Bison rules are basically BNF, with the punctuation simplified a little to make them easier to type.
- Example 1-5 shows the bison code, including the BNF, for the first version of our calculator.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



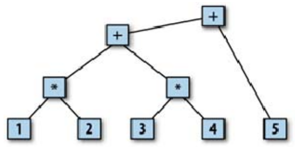
```
/* simplest version of calculator */
%{
#include <stdio.h>
%}

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL

%%
```

- Bison programs have (not by coincidence) the same three-part structure as flex programs, with declarations, rules, and C code.
- The declarations here include C code to be copied to the beginning of the generated C parser, again enclosed in %{ and %}.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



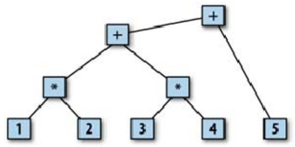
```
/* simplest version of calculator */
%{
#include <stdio.h>
%}

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL

%%
```

- Following that are %token token declarations, telling bison the names of the symbols in the parser that are tokens.
- By convention, tokens have uppercase names, although bison doesn't require it.
- Any symbols not declared as tokens have to appear on the left side of at least one rule in the program.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



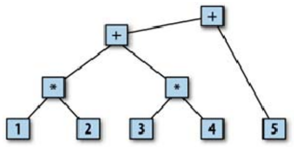
```
calclist:                  /* nothing - matches at beginning of input */
        | calclist exp EOL { printf("= %d\n", $2); }
                           /* EOL is end of an expression */
 ;

exp:    factor         /* default  $$ = $1 */
        | exp ADD factor { $$ = $1 + $3; }
        | exp SUB factor { $$ = $1 - $3; }
 ;
```

- The second section contains the rules in simplified BNF. Bison uses a single colon rather than ::=, and since line boundaries are not significant, a semicolon marks the end of a rule.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



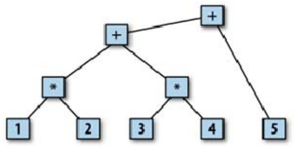
```
calclist:                       /* nothing - matches at beginning of input */
        | calclist exp EOL      { printf("= %d\n", $2); }
                                /* EOL is end of an expression */
 ;

exp:    factor          /* default  $$ = $1 */
        | exp ADD factor { $$ = $1 + $3; }
        | exp SUB factor { $$ = $1 - $3; }
 ;
```

- Each symbol in a bison rule has a value; the value of the target symbol (the one to the left of the colon) is called $$ in the action code, and the values on the right are numbered $1, $2, and so forth, up to the number of symbols in the rule.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

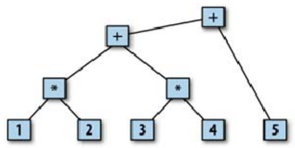



```
factor: term                    // default $$ = $1
        | factor MUL term { $$ = $1 * $3; }
        | factor DIV term { $$ = $1 / $3; }
 ;

term: NUMBER                    // default $$ = $1
        | ABS term   { $$ = $2 >= 0? $2 : - $2; }
;
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

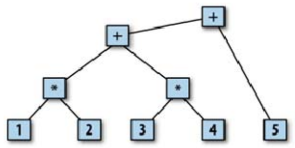


```
%%
main(int argc, char **argv)
{
        yyparse();
}

yyerror(char *s)
{
        fprintf(stderr, "error: %s\n", s);
}
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

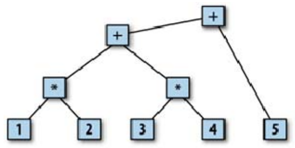


- Comment.
- Bison programs have (not by coincidence) the same three-part structure as flex programs, with declarations, rules, and C code.
- The declarations here include C code to be copied to the beginning of the generated C parser, again enclosed in %{ and %}.
- Following that are %token token declarations, telling bison the names of the symbols in the parser that are tokens.
- By convention, tokens have uppercase names, although bison doesn't require it.
- 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.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

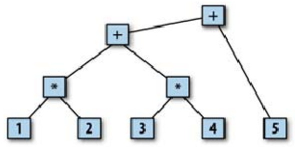


- Comment.
- The second section contains the rules in simplified BNF. Bison uses a single colon rather than ::=, and since line boundaries are not significant, a semicolon marks the end of a rule.
- Again, like flex, the C action code goes in braces at the end of each rule.
- Bison automatically does the parsing for you, remembering what rules have been matched, so the action code maintains the values associated with each symbol.
- Bison parsers also perform side effects such as creating data structures for later use or, as in this case, printing out results.
- The symbol on the left side of the first rule is the start symbol, the one that the entire input has to match.
- There can be, and usually are, other rules with the same start symbol on the left.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

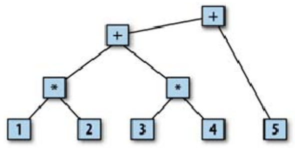


- Comment.
- Each symbol in a bison rule has a value; the value of the target symbol (the one to the left of the colon) is called $$ in the action code, and the values on the right are numbered $1, $2, and so forth, up to the number of symbols in the rule.
- The values of tokens are whatever was in yylval when the scanner returned the token; the values of other symbols are set in rules in the parser. In this parser, the values of the factor, term, and exp symbols are the value of the expression they represent.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

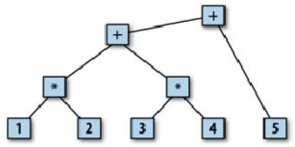


- Comment
- In this parser, the first two rules, which define the symbol calclist, implement a loop that reads an expression terminated by a newline and prints its value.
- The definition of calclist uses a common two-rule recursive idiom to implement a sequence or list: the first rule is empty and matches nothing; the second adds an item to the list.
- The action in the second rule prints the value of the exp in $2.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

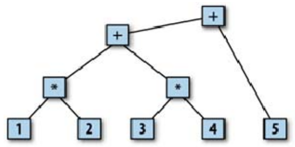


- Comment
- The rest of the rules implement the calculator.
- The rules with operators such as exp ADD factor and ABS term do the appropriate arithmetic on the symbol values.
- The rules with a single symbol on the right side are syntactic glue to put the grammar together; for example, an exp is a factor.
- In the absence of an explicit action on a rule, the parser assigns $1 to $$.
- This is a hack, albeit a very useful one, since most of the time it does the right thing.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

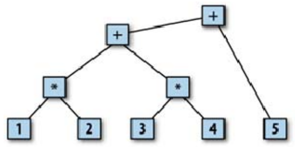


- Compiling Flex and Bison Programs Together
- Before we build the scanner and parser into a working program, we have to make some small changes to the scanner in Example 1-4 so we can call it from the parser.
- In particular, rather than defining explicit token values in the first part, we include a header file that bison will create for us, which includes both definitions of the token numbers and a definition of yylval.
- We also delete the testing main routine in the third section of the scanner, since the parser will now call the scanner.
- The first part of the scanner now looks like Example 1-6.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



- Example 1-6. Calculator scanner fb1-5.l

```
%{
# include "fb1-5.tab.h"
%}
%%  same rules as before, and no code in the third section
```

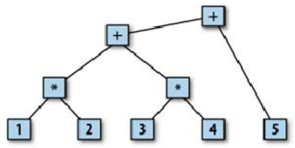
- The build process is now complex enough to be worth putting into a Makefile:

```
# part of the makefile
fb1-5: fb1-5.l fb1-5.y
        bison -d fb1-5.y
        flex fb1-5.l
        cc -o $@ fb1-5.tab.c lex.yy.c -lfl
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

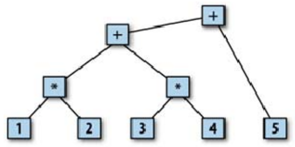


- First it runs bison with the -d (for “definitions” file) flag, which creates fb1-5.tab.c and fb1-5.tab.h, and it runs flex to create lex.yy.c.
- Then it compiles them together, along with the flex library.
- Try it out, and in particular verify that it handles operator precedence correctly, doing multiplication and division before addition and subtraction: 2 + 3 * 4

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

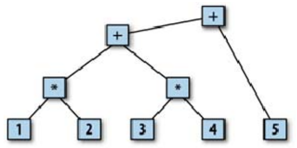


```
$ ./fb1-5
2 + 3 * 4
= 14
2 * 3 + 4
= 10
20 / 4 - 2
= 3
20 - 4 / 2
= 18
```

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5

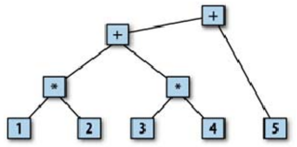


- Adding a Few More Rules
- One of the nicest things about using flex and bison to handle a program's input is that it's often quite easy to make small changes to the grammar.
- Our expression language would be a lot more useful if it could handle parenthesized expressions, and it would be nice if it could handle comments, using // syntax.
- To do this, we need only add one rule to the parser and three to the scanner.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



- In the parser we define two new tokens, OP and CP for open and close parentheses, and add a rule to make a parenthesized expression a term:

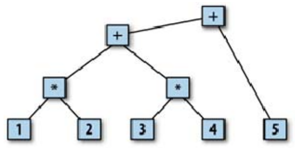
```
%token OP CP  in the declaration section
 ...
%%
term:  NUMBER
        | ABS term { $$ = $2 >= 0? $2 : - $2; }
        | OP exp CP { $$ = $2; } New rule
 ;
```

- Note the action code in the new rule assigns $2, the value of the expression in the parentheses, to $$.

flex & bison

flex & bison. John Levine. © 2009 O'Reilly Media, Inc.

1 * 2 + 3 * 4 + 5



- The scanner has two new rules to recognize the two new tokens and one new rule to ignore two slashes followed by arbitrary text.
- Since a dot matches anything except a newline, .* will gobble up the rest of the line.

```
"("    { return OP; }
")"    { return CP; }
"//".*  /* ignore comments */
```

- That's it—rebuild the calculator, and now it handles parenthesized expressions and comments.