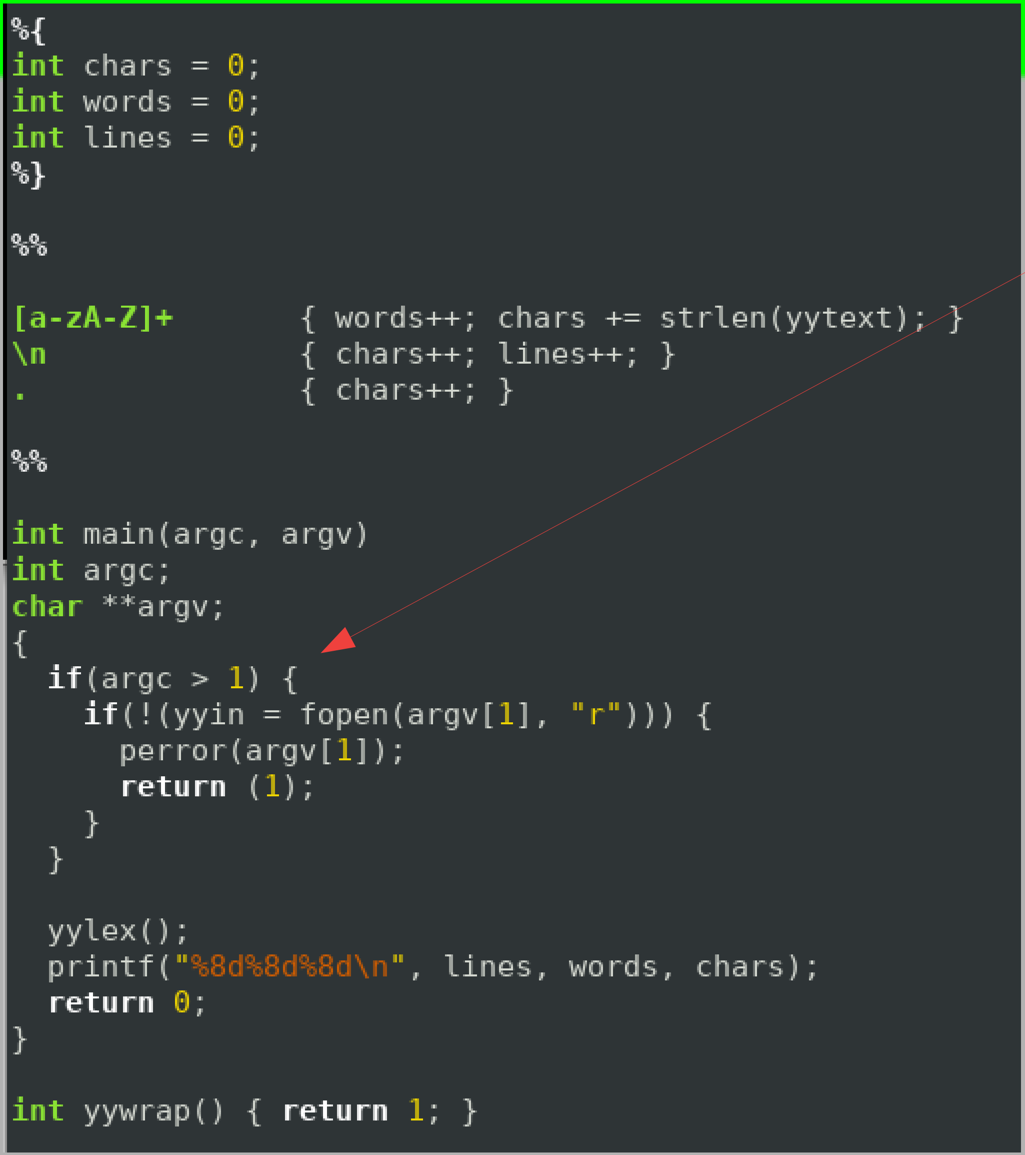
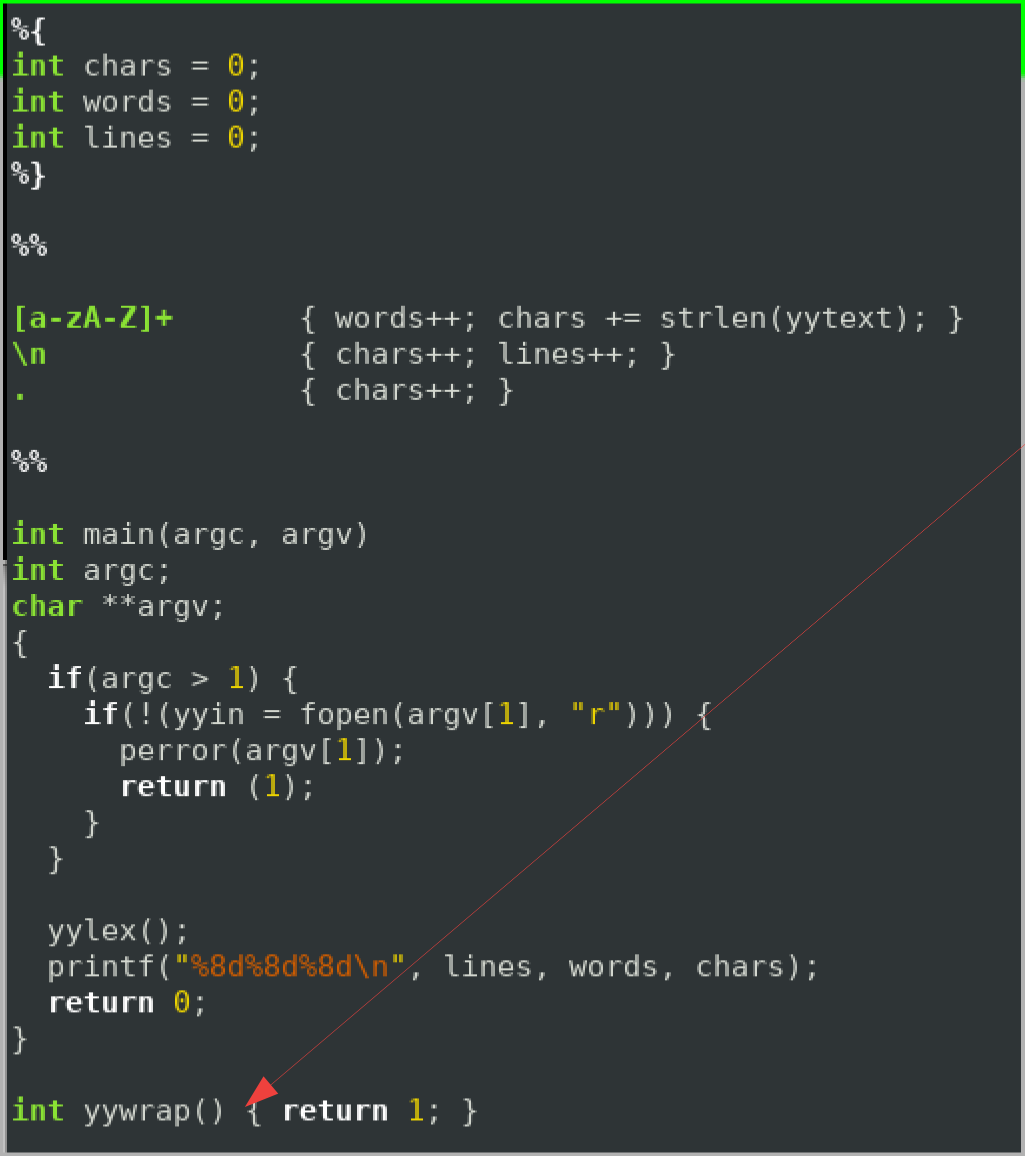
File I/O in Flex Scanners



```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}

%%

[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }

%%

int main(argc, argv)
int argc;
char **argv;
{
  if(argc > 1) {
    if(!(yyin = fopen(argv[1], "r"))) {
      perror(argv[1]);
      return (1);
    }
  }

  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
  return 0;
}

int yywrap() { return 1; }
```

Unless you make other arrangements, a scanner reads from the stdio FILE called yyin , so to read a single file, you need only set it before the first call to yylex .

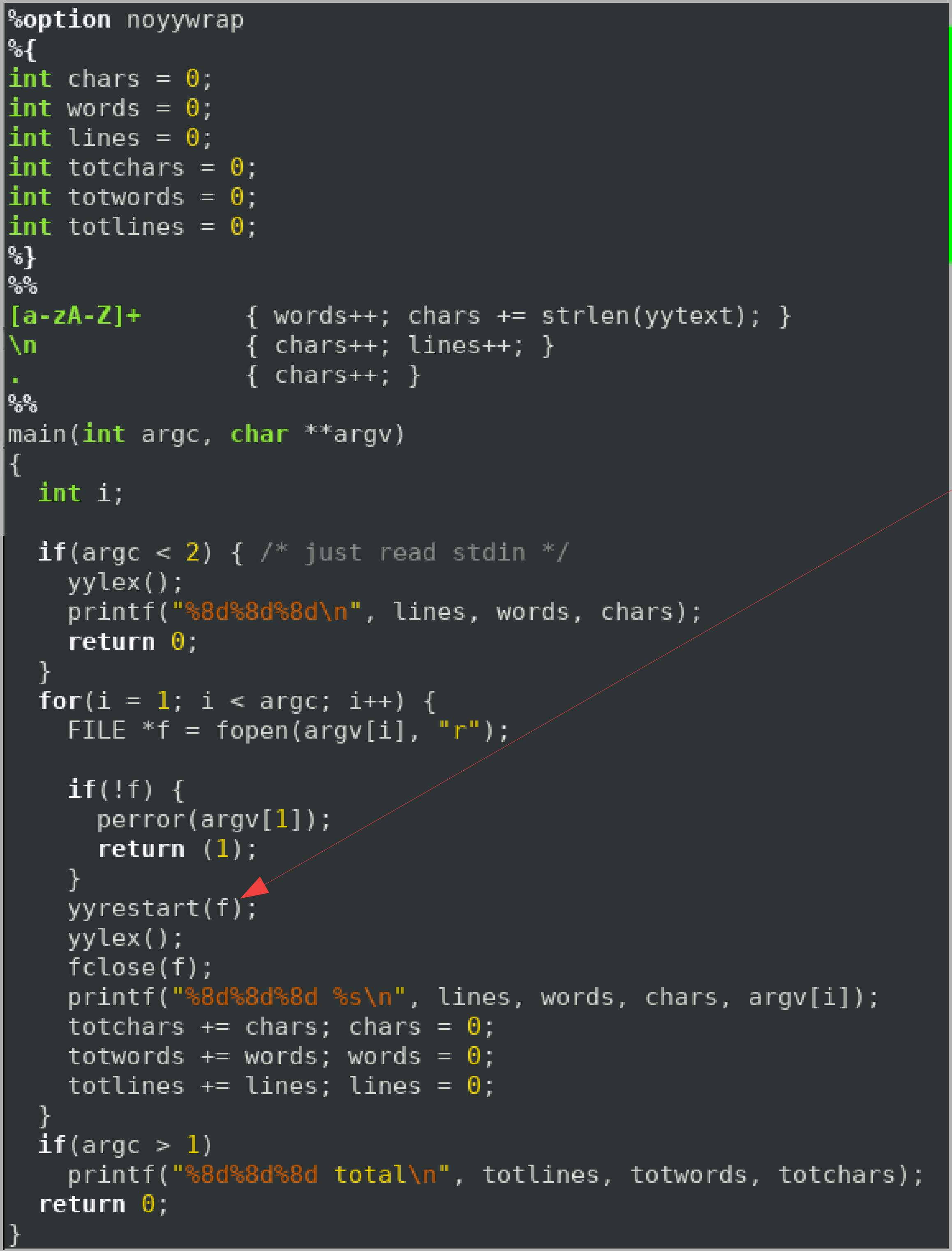
The main routine opens a filename passed on the command line, if the user specified one, and assigns the FILE to yyin . Otherwise, yyin is left unset, in which case yylex automatically sets it to stdin .



```
%{
int chars = 0;
int words = 0;
int lines = 0;
%}

%%

[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }

%%

int main(argc, argv)
int argc;
char **argv;
{
  if(argc > 1) {
    if(!(yyin = fopen(argv[1], "r"))) {
      perror(argv[1]);
      return (1);
    }
  }

  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
  return 0;
}

int yywrap() { return 1; }
```

When a lex scanner reached the end of yyin , it called yywrap() . The idea was that if
there was another input file, yywrap could adjust yyin and return 0 to resume scanning.
If that was really the end of the input, it returned 1 to the scanner to say that it was done.

Alternative
%option noyywrap



```
%option noyywrap
%{
int chars = 0;
int words = 0;
int lines = 0;
int totchars = 0;
int totwords = 0;
int totlines = 0;
%}
%%
[a-zA-Z]+       { words++; chars += strlen(yytext); }
\n              { chars++; lines++; }
.               { chars++; }
%%
main(int argc, char **argv)
{
  int i;

  if(argc < 2) { /* just read stdin */
    yylex();
    printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
  }
  for(i = 1; i < argc; i++) {
    FILE *f = fopen(argv[i], "r");

    if(!f) {
      perror(argv[1]);
      return (1);
    }
    yyrestart(f);
    yylex();
    fclose(f);
    printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
    totchars += chars; chars = 0;
    totwords += words; words = 0;
    totlines += lines; lines = 0;
  }
  if(argc > 1)
    printf("%8d%8d%8d total\n", totlines, totwords, totchars);
  return 0;
}
```

For each file, it opens the file, uses yyrestart() to make it the input to the scanner, and
calls yylex() to scan it.

To handle its input, a flex scanner uses a structure known as
a YY_BUFFER_STATE, which describes a single input source.

It contains a string buffer and a bunch of variables and
flags.

Usually it contains a FILE* for the file it’s reading from,
but it’s also possible to create a YY_BUFFER_STATE not connected
a file to scan a string already in memory.

The default input behavior of a flex scanner is approximately this:

```
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_SIZE );
//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
```

There are several other functions to create scanner buffers, including
yy_scan_string("string") to scan a null-terminated string and
yy_scan_buffer(char *base, size) to scan a buffer of known size.

We'll try out our knowledge of flex I/O with a simple program that handles nested include files and prints them out. To make it a little more interesting, it prints the input files with the line number of each line in its file.

To do that, the program keeps a stack of nested input files and line numbers, pushing an entry each time it encounters a #include and popping an entry off the stack when it gets to the end of a file.

We also use a very powerful flex feature called start states that let us control which patterns can be matched when.

The %x line near the top of the file defines IFILE as a start state that we'll use when we're looking for the filename in a #include statement.

At any point, the scanner is in one start state and can match patterns active in that start state only.
In effect, the state defines a different scanner, with its own rules.

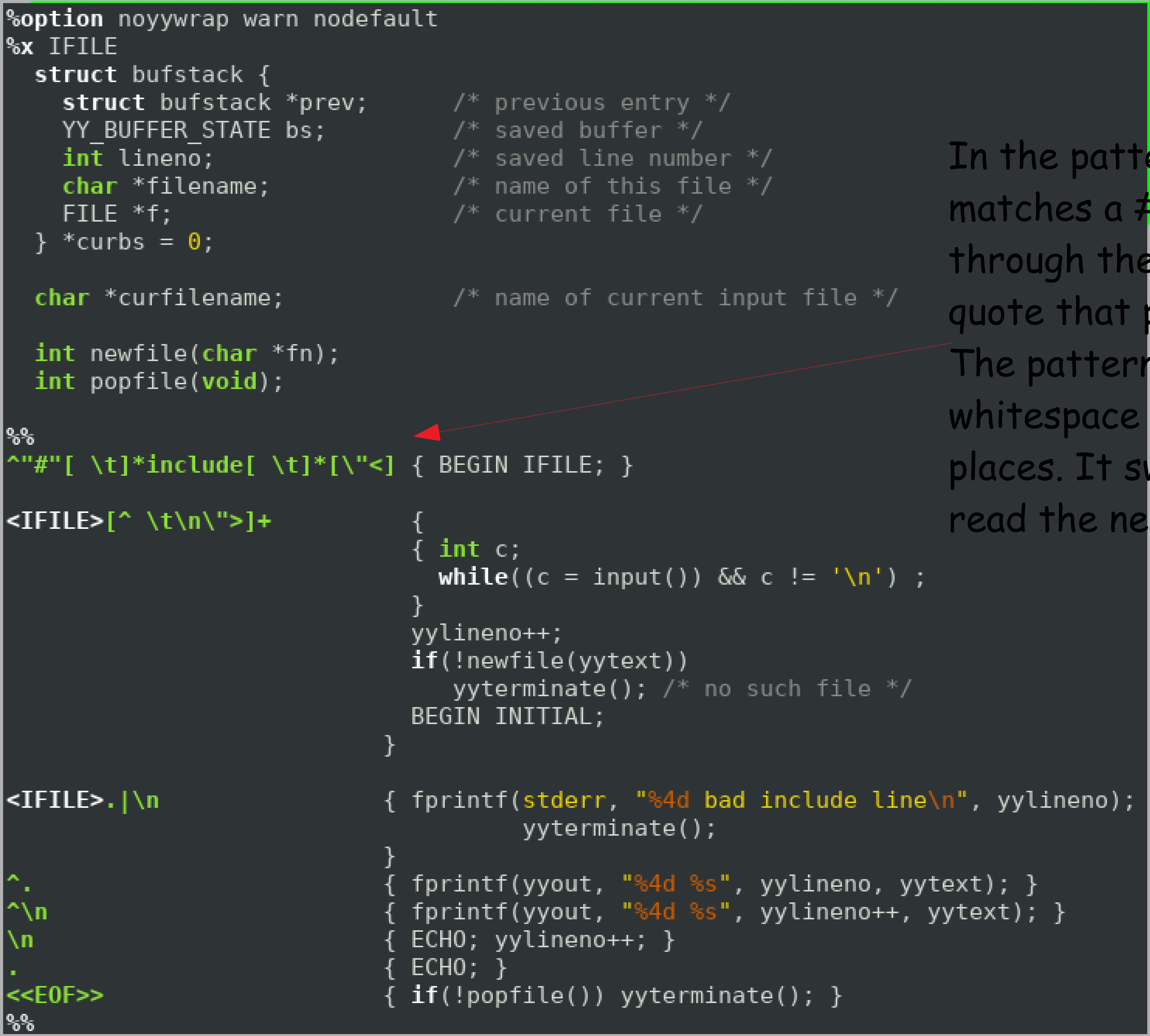
You can define as many start states as needed, but in this program we need only one in addition to the INITIAL state that flex always defines.

Patterns are tagged with start state names in angle brackets to indicate in which state(s) the pattern is active.

The %x marks IFILE as an exclusive start state, which means that when that state is active, only patterns specifically marked with the state can match.

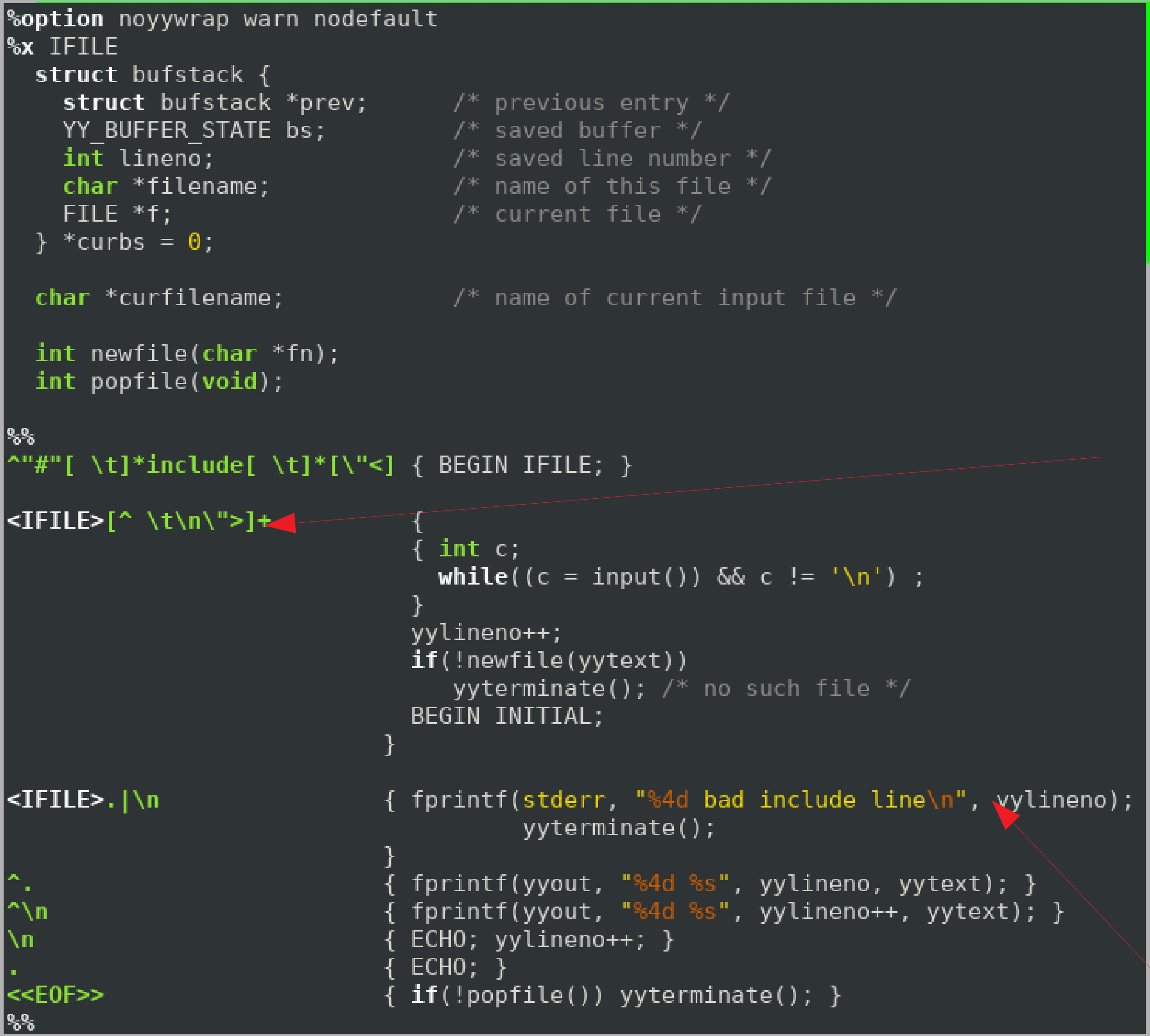
(There are also inclusive start states declared with %s , in which patterns not marked with any state can also match. Exclusive states are usually more useful.)

In action code, the macro BEGIN switches to a different start state.



```
%option noyywrap warn nodefault
%x IFILE
  struct bufstack {
    struct bufstack *prev;      /* previous entry */
    YY_BUFFER_STATE bs;         /* saved buffer */
    int lineno;                 /* saved line number */
    char *filename;             /* name of this file */
    FILE *f;                    /* current file */
  } *curbs = 0;

  char *curfilename;            /* name of current input file */

  int newfile(char *fn);
  int popfile(void);

%%
^"#"[ \t]*include[ \t]*[\"<] { BEGIN IFILE; }

<IFILE>[^ \t\n\">]+          {
                             { int c;
                               while((c = input()) && c != '\n') ;
                             }
                             yylineno++;
                             if(!newfile(yytext))
                                yyterminate(); /* no such file */
                             BEGIN INITIAL;
                           }

<IFILE>.|\n                { fprintf(stderr, "%4d bad include line\n", yylineno);
                                     yyterminate();
                           }
^.                         { fprintf(yyout, "%4d %s", yylineno, yytext); }
^\n                        { fprintf(yyout, "%4d %s", yylineno++, yytext); }
\n                         { ECHO; yylineno++; }
.                          { ECHO; }
<<EOF>>                    { if(!popfile()) yyterminate(); }
%%
```

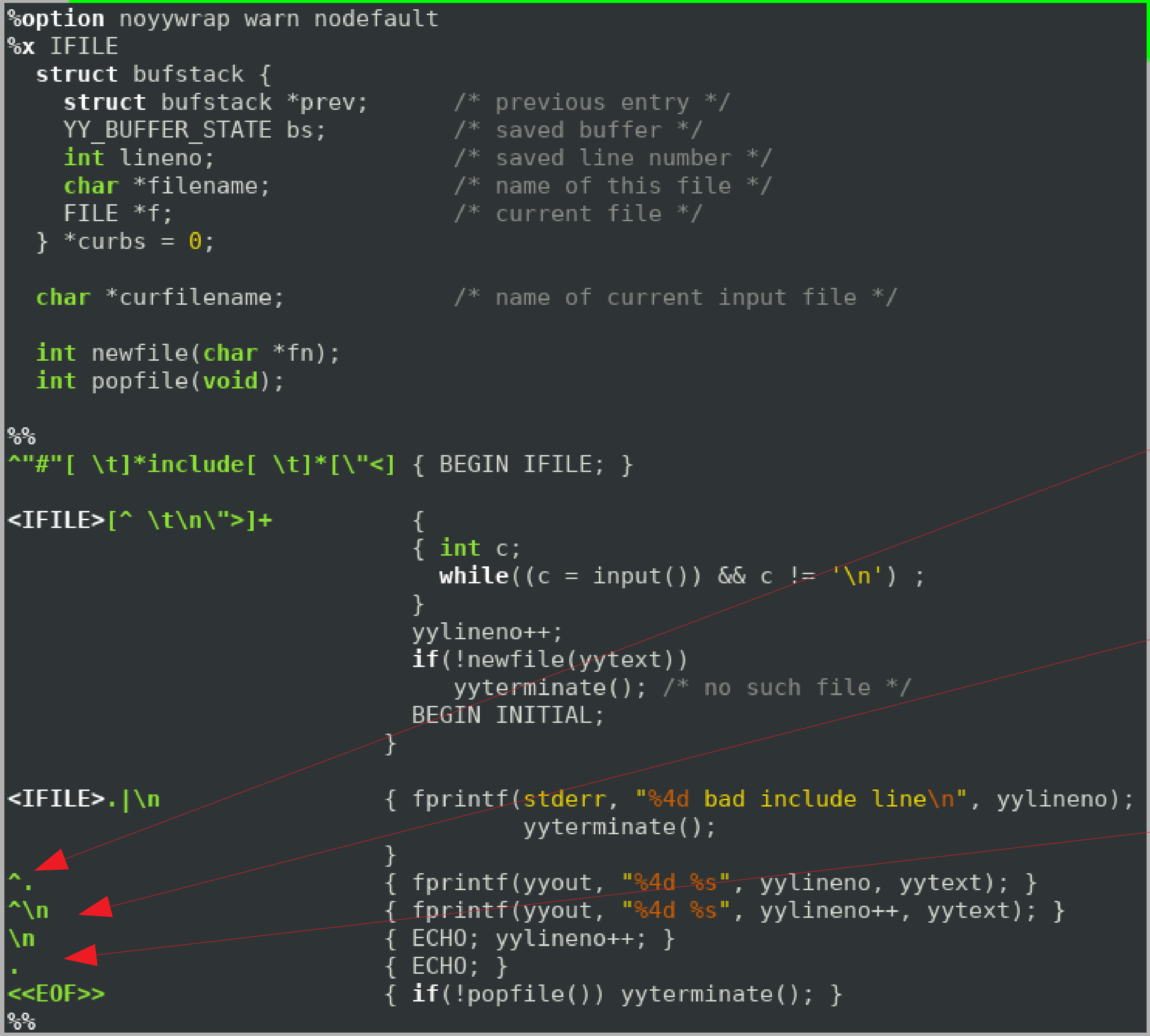
In the patterns, the first pattern matches a #include statement up through the double quote that precedes the filename. The pattern permits optional whitespace in the usual places. It switches to IFILE state to read the next input filename.



```
%option noyywrap warn nodefault
%x IFILE
  struct bufstack {
    struct bufstack *prev;      /* previous entry */
    YY_BUFFER_STATE bs;         /* saved buffer */
    int lineno;                 /* saved line number */
    char *filename;             /* name of this file */
    FILE *f;                    /* current file */
  } *curbs = 0;

  char *curfilename;            /* name of current input file */

  int newfile(char *fn);
  int popfile(void);

%%
^"#"[ \t]*include[ \t]*[\"<] { BEGIN IFILE; }

<IFILE>[^ \t\n\">]+          {
                             { int c;
                               while((c = input()) && c != '\n') ;
                             }
                             yylineno++;
                             if(!newfile(yytext))
                                yyterminate(); /* no such file */
                             BEGIN INITIAL;
                           }

<IFILE>.|\n                { fprintf(stderr, "%4d bad include line\n", yylineno);
                                     yyterminate();
                           }
^.                         { fprintf(yyout, "%4d %s", yylineno, yytext); }
^\n                        { fprintf(yyout, "%4d %s", yylineno++, yytext); }
\n                         { ECHO; yylineno++; }
.                          { ECHO; }
<<EOF>>                    { if(!popfile()) yyterminate(); }
%%
```

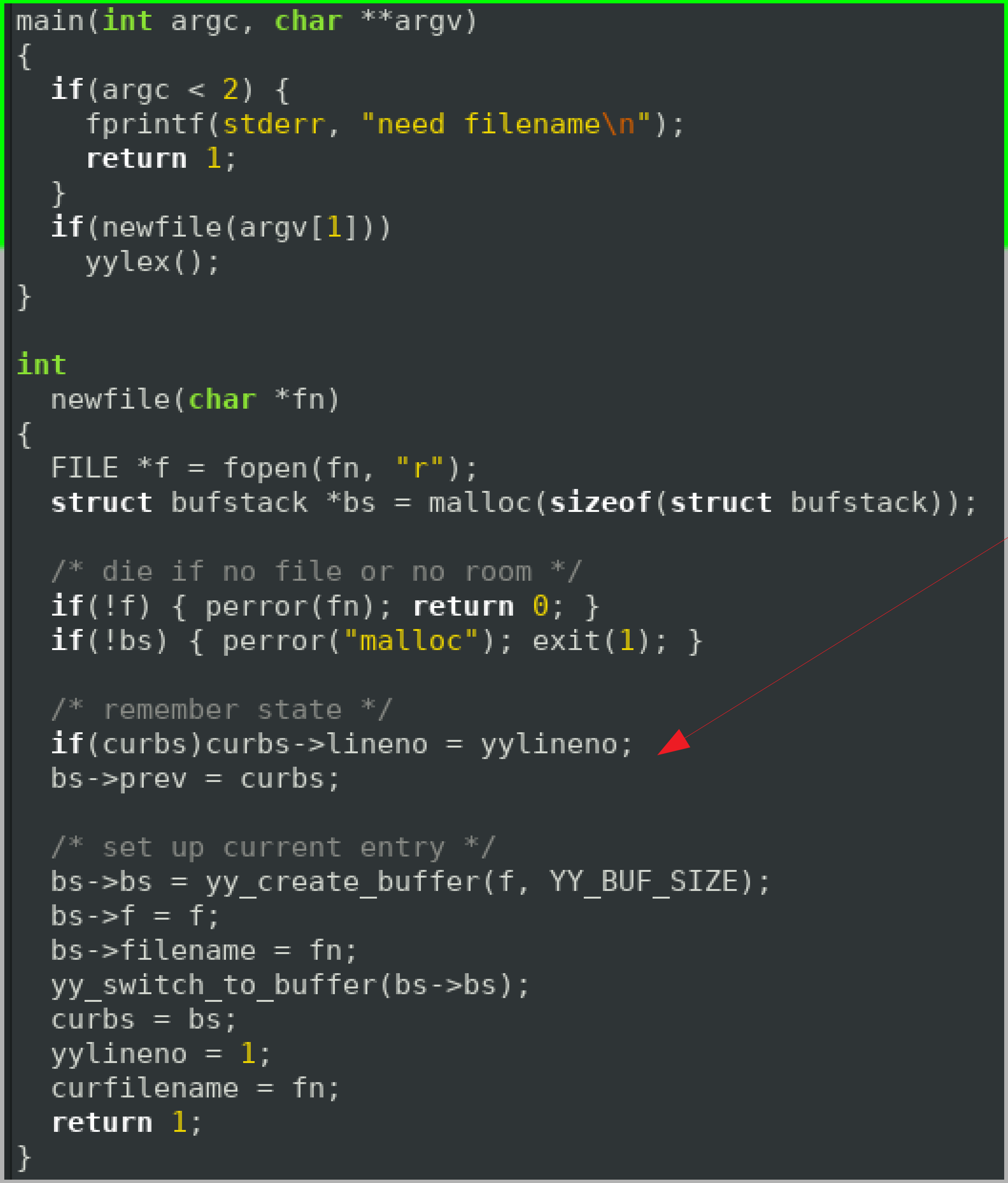
In IFILE state, the second pattern matches a filename, characters up to a closing quote, whitespace, or end-of-line. The filename is passed to newfile to stack the current input file and set up the next level of input, but first there's the matter of dealing with whatever remains of the #include line.

The next pattern deals with the case of an Ill-formed #include line that doesn't have a filename after the double quote.



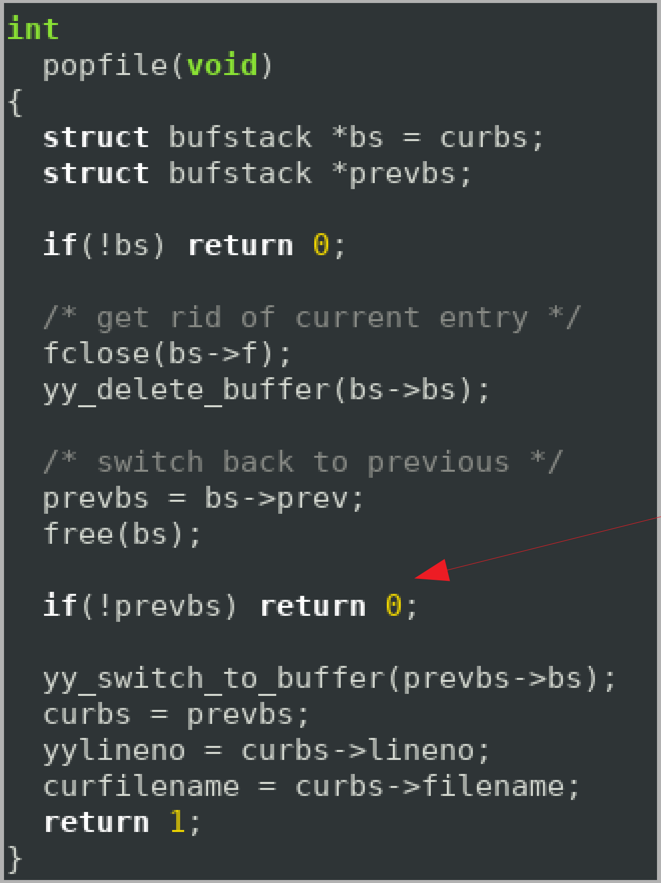
```
%option noyywrap warn nodefault
%x IFILE
  struct bufstack {
    struct bufstack *prev;      /* previous entry */
    YY_BUFFER_STATE bs;         /* saved buffer */
    int lineno;                 /* saved line number */
    char *filename;             /* name of this file */
    FILE *f;                    /* current file */
  } *curbs = 0;

  char *curfilename;            /* name of current input file */

  int newfile(char *fn);
  int popfile(void);

%%
^"#"[ \t]*include[ \t]*[\"<] { BEGIN IFILE; }

<IFILE>[^ \t\n\">]+          {
                             { int c;
                               while((c = input()) && c != '\n') ;
                             }
                             yylineno++;
                             if(!newfile(yytext))
                                yyterminate(); /* no such file */
                             BEGIN INITIAL;
                           }

<IFILE>.|\n                { fprintf(stderr, "%4d bad include line\n", yylineno);
                                    yyterminate();
                           }
^.                         { fprintf(yyout, "%4d %s", yylineno, yytext); }
^\n                        { fprintf(yyout, "%4d %s", yylineno++, yytext); }
\n                         { ECHO; yylineno++; }
.                          { ECHO; }
<<EOF>>                    { if(!popfile()) yyterminate(); }
%%
```

The last four patterns do the actual work of printing out each line with a preceding line number. Flex provides a variable called yylineno that is intended to track line numbers, so we might as well use it. The pattern ^. matches any character at the beginning of a line, so the action prints the current line number and the character. Since a dot doesn't match a newline, ^\n matches a newline at the beginning of a line, that is, an empty line, so the code prints out the line number and the new line and increments the line number. A newline or other character not at the beginning of the line is just printed out with ECHO , incrementing the line number for a new line.



```
main(int argc, char **argv)
{
  if(argc < 2) {
    fprintf(stderr, "need filename\n");
    return 1;
  }
  if(newfile(argv[1]))
    yylex();
}

int
  newfile(char *fn)
{
  FILE *f = fopen(fn, "r");
  struct bufstack *bs = malloc(sizeof(struct bufstack));

  /* die if no file or no room */
  if(!f) { perror(fn); return 0; }
  if(!bs) { perror("malloc"); exit(1); }

  /* remember state */
  if(curbs)curbs->lineno = yylineno;
  bs->prev = curbs;

  /* set up current entry */
  bs->bs = yy_create_buffer(f, YY_BUF_SIZE);
  bs->f = f;
  bs->filename = fn;
  yy_switch_to_buffer(bs->bs);
  curbs = bs;
  yylineno = 1;
  curfilename = fn;
  return 1;
}
```

For the first file, don't need to save a line number



```
int
  popfile(void)
{
  struct bufstack *bs = curbs;
  struct bufstack *prevbs;

  if(!bs) return 0;

  /* get rid of current entry */
  fclose(bs->f);
  yy_delete_buffer(bs->bs);

  /* switch back to previous */
  prevbs = bs->prev;
  free(bs);

  if(!prevbs) return 0;

  yy_switch_to_buffer(prevbs->bs);
  curbs = prevbs;
  yylineno = curbs->lineno;
  curfilename = curbs->filename;
  return 1;
}
```

Nothing on stack, end of first file