**Init and Destroy Functions**

yylex\_init and yylex\_destroy must be called before and after yylex, respectively.

int yylex\_init ( yyscan\_t \* ptr\_yy\_globals ) ;

int yylex\_init\_extra ( YY\_EXTRA\_TYPE user\_defined, yyscan\_t \* ptr\_yy\_globals ) ;

int yylex ( yyscan\_t yyscanner ) ;

int yylex\_destroy ( yyscan\_t yyscanner ) ;

The function yylex\_init must be called before calling any other function. The argument to yylex\_init is the address of an uninitialized pointer to be filled in by yylex\_init, overwriting any previous contents. The function yylex\_init\_extra may be used instead, taking as its first argument a variable of type YY\_EXTRA\_TYPE. See the section on yyextra, below, for more details.

The value stored in ptr\_yy\_globals should thereafter be passed to yylex and yylex\_destroy. Flex does not save the argument passed to yylex\_init, so it is safe to pass the address of a local pointer to yylex\_init so long as it remains in scope for the duration of all calls to the scanner, up to and including the call to yylex\_destroy.

The function yylex should be familiar to you by now. The reentrant version takes one argument, which is the value returned (via an argument) by yylex\_init. Otherwise, it behaves the same as the non-reentrant version of yylex.

Both yylex\_init and yylex\_init\_extra returns 0 (zero) on success, or non-zero on failure, in which case errno is set to one of the following values:

* ENOMEM Memory allocation error. See [memory-management](http://flex.sourceforge.net/manual/memory_002dmanagement.html#memory_002dmanagement).
* EINVAL Invalid argument.

The function yylex\_destroy should be called to free resources used by the scanner. After yylex\_destroy is called, the contents of yyscanner should not be used. Of course, there is no need to destroy a scanner if you plan to reuse it. A flex scanner (both reentrant and non-reentrant) may be restarted by calling yyrestart.

Below is an example of a program that creates a scanner, uses it, then destroys it when done:

int main ()

{

yyscan\_t scanner;

int tok;

yylex\_init(&scanner);

while ((tok=yylex(scanner)) > 0)

printf("tok=%d yytext=%s\n", tok, yyget\_text(scanner));

yylex\_destroy(scanner);

return 0;

}

https://books.google.co.in/books?id=3Sr1V5J9\_qMC&pg=PA212&lpg=PA212&dq=yyset\_in&source=bl&ots=WFGvl5kIGP&sig=jPtfDiGlqhsiPFv60ALf4pgLz4U&hl=en&sa=X&ei=Mnn8VN62ApGJuASfyYLQDA&redir\_esc=y#v=onepage&q=yyset\_in&f=false

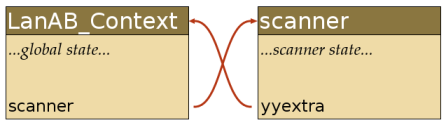
Before we start looking at the details, we will give a high level overview of the solution first. There is more than one way to solve this problem; the method I will present here is the one I believe is the least messy, but that is of course a matter of opinion; one alternative (using a C++ lexer) is discussed briefly [at the end](http://www.phpcompiler.org/articles/reentrantparser.html#alternative) of this tutorial.

Unfortunately, although the solutions offered by Flex and Bison are very similar, they are slightly different in the details, so it will be important to remember which tool we are talking about.

When Flex generates a reentrant scanner, the function yylex will get an additional argument scanner. This argument is a pointer to a data structure that represents the state of the scanner. Before we start parsing, we must initialise this state, and then pass the state in to yylex every time it is invoked. The scanner state has a data field, called yyextra, of a user-specified type, that can be used for additional state. We will use yyextra to determine the semantic value of each character in the input (according to the rules explained in the introduction).

The Bison generated parser yyparse also gets an additional argument, but this argument represents the user-defined state only. The Bison internal global state is stored in local variables inside yyparse, and is completely invisible to the user.

We will create a class LanAB\_Context (for “language **AB** context”) to hold the global (user-defined) state. We will pass in an object of type LanAB\_Context to yyparse. Since yyparse needs to call yylex, we will find it useful to store a reference to the scanner object inside LanAB\_Context. However, since we only have access to the scanner object from within yylex, we will use yyextra (mentioned above) inside the scanner object to point back to the LanAB\_Context object. Graphically:



**Making a reentrant (thread-safe) parser with Flex and Bison involves several stages.**

To eliminate global variables from Flex, use the following line:

%option reentrant

This changes yylex () to yylex (void \*). The argument to yylex contains initialized memory for the lexer which is initialized using yylex\_init:

void \* something;

yylex\_init (& something);

yylex (something);

and then release the memory after finishing the lexing:

yylex\_destroy (something);

In the flex documentation, this is given as yyscan\_t, but it's void \*.

If the lexer is combined with a Bison parser, add

%option bison-bridge

to the Flex input file options. This makes Flex add extra arguments to yylex to use instead of using the global variable yylval:

int yylex ( YYSTYPE \* lvalp, yyscan\_t scanner );

You can then use yylval in the Flex lexer, and it will refer to whatever is passed in as the first argument to yylex above.

The type of yylval, YYSTYPE, is also needed, so run Bison with the -d option to create the file y.tab.h which defines it for you, and include this file into the lex file using the top section:

%{

#include "y.tab.h"

%}

See [C Scanners with Bison Parsers - Flex manual](http://flex.sourceforge.net/manual/Bison-Bridge.html) for more details.

If your parser's value type is a union, any yyval. instances used to fill union members will need to change to yylval->.

When running flex, use the --header-file=something.h option to generate a header file to include in the parser file.

In the Bison input file, add

%pure-parser

to make a reentrant parser,

%lex-param {void \* scanner}

to tell it to send the lexer an extra argument, and

%parse-param {void \* scanner}

to add another argument to yyparse, which is the thing to pass in to Flex.

The above is already enough to create a reentrant parser. If you also need to pass in something to Bison, you can add a member

struct pass\_to\_bison {

....

void \* scanner;

} x;

with

%parse-param {struct pass\_to\_bison \* x}

then use a preprocessor macro like

#define scanner x->scanner

in the Bison preamble to make Bison send the scanner. In this case, use

struct pass\_to\_bison x;

yylex\_init (& x->scanner);

yyparse (& x);

yylex\_destroy (& x->scanner);

To use private data in the Flex lexer, set its value with yylex\_set\_extra:

struct user\_type {

int number;

};

struct user\_type \* user\_data;

yylex\_set\_extra (user\_data, scanner);

after calling yyinit\_lexer. Here scanner is the data passed to yyinit\_lexer. In the preamble of the Flex file, add

%{

#define YY\_EXTRA\_TYPE struct user\_type \*

%}

The data in user\_data is then available in the lexer as yyextra:

%%

.\* { yyextra->number++; }

%%