# The Lemon Parser Generator

Lemon is an LALR(1) parser generator for C or C++. It does the same job as ``bison'' and ``yacc''. But lemon is not another bison or yacc clone. It uses a different grammar syntax which is designed to reduce the number of coding errors. Lemon also uses a more sophisticated parsing engine that is faster than yacc and bison and which is both reentrant and thread-safe. Furthermore, Lemon implements features that can be used to eliminate resource leaks, making is suitable for use in long-running programs such as graphical user interfaces or embedded controllers.

This document is an introduction to the Lemon parser generator.

## Theory of Operation

The main goal of Lemon is to translate a context free grammar (CFG) for a particular language into C code that implements a parser for that language. The program has two inputs:

* The grammar specification.
* A parser template file.

Typically, only the grammar specification is supplied by the programmer. Lemon comes with a default parser template which works fine for most applications. But the user is free to substitute a different parser template if desired.

Depending on command-line options, Lemon will generate between one and three files of outputs.

* C code to implement the parser.
* A header file defining an integer ID for each terminal symbol.
* An information file that describes the states of the generated parser automaton.

By default, all three of these output files are generated. The header file is suppressed if the ``-m'' command-line option is used and the report file is omitted when ``-q'' is selected.

The grammar specification file uses a ``.y'' suffix, by convention. In the examples used in this document, we'll assume the name of the grammar file is ``gram.y''. A typical use of Lemon would be the following command:

lemon gram.y

This command will generate three output files named ``gram.c'', ``gram.h'' and ``gram.out''. The first is C code to implement the parser. The second is the header file that defines numerical values for all terminal symbols, and the last is the report that explains the states used by the parser automaton.

### Command Line Options

The behavior of Lemon can be modified using command-line options. You can obtain a list of the available command-line options together with a brief explanation of what each does by typing

lemon -?

As of this writing, the following command-line options are supported:

* -b
* -c
* -g
* -m
* -q
* -s
* -x

The ``-b'' option reduces the amount of text in the report file by printing only the basis of each parser state, rather than the full configuration. The ``-c'' option suppresses action table compression. Using -c will make the parser a little larger and slower but it will detect syntax errors sooner. The ``-g'' option causes no output files to be generated at all. Instead, the input grammar file is printed on standard output but with all comments, actions and other extraneous text deleted. This is a useful way to get a quick summary of a grammar. The ``-m'' option causes the output C source file to be compatible with the ``makeheaders'' program. Makeheaders is a program that automatically generates header files from C source code. When the ``-m'' option is used, the header file is not output since the makeheaders program will take care of generated all header files automatically. The ``-q'' option suppresses the report file. Using ``-s'' causes a brief summary of parser statistics to be printed. Like this:

Parser statistics: 74 terminals, 70 nonterminals, 179 rules  
 340 states, 2026 parser table entries, 0 conflicts

Finally, the ``-x'' option causes Lemon to print its version number and then stops without attempting to read the grammar or generate a parser.

### The Parser Interface

Lemon doesn't generate a complete, working program. It only generates a few subroutines that implement a parser. This section describes the interface to those subroutines. It is up to the programmer to call these subroutines in an appropriate way in order to produce a complete system.

Before a program begins using a Lemon-generated parser, the program must first create the parser. A new parser is created as follows:

void \*pParser = ParseAlloc( malloc );

The ParseAlloc() routine allocates and initializes a new parser and returns a pointer to it. The actual data structure used to represent a parser is opaque -- its internal structure is not visible or usable by the calling routine. For this reason, the ParseAlloc() routine returns a pointer to void rather than a pointer to some particular structure. The sole argument to the ParseAlloc() routine is a pointer to the subroutine used to allocate memory. Typically this means ``malloc()''.

After a program is finished using a parser, it can reclaim all memory allocated by that parser by calling

ParseFree(pParser, free);

The first argument is the same pointer returned by ParseAlloc(). The second argument is a pointer to the function used to release bulk memory back to the system.

After a parser has been allocated using ParseAlloc(), the programmer must supply the parser with a sequence of tokens (terminal symbols) to be parsed. This is accomplished by calling the following function once for each token:

Parse(pParser, hTokenID, sTokenData, pArg);

The first argument to the Parse() routine is the pointer returned by ParseAlloc(). The second argument is a small positive integer that tells the parse the type of the next token in the data stream. There is one token type for each terminal symbol in the grammar. The gram.h file generated by Lemon contains #define statements that map symbolic terminal symbol names into appropriate integer values. (A value of 0 for the second argument is a special flag to the parser to indicate that the end of input has been reached.) The third argument is the value of the given token. By default, the type of the third argument is integer, but the grammar will usually redefine this type to be some kind of structure. Typically the second argument will be a broad category of tokens such as ``identifier'' or ``number'' and the third argument will be the name of the identifier or the value of the number.

The Parse() function may have either three or four arguments, depending on the grammar. If the grammar specification file request it, the Parse() function will have a fourth parameter that can be of any type chosen by the programmer. The parser doesn't do anything with this argument except to pass it through to action routines. This is a convenient mechanism for passing state information down to the action routines without having to use global variables.

A typical use of a Lemon parser might look something like the following:

01 ParseTree \*ParseFile(const char \*zFilename){  
 02 Tokenizer \*pTokenizer;  
 03 void \*pParser;  
 04 Token sToken;  
 05 int hTokenId;  
 06 ParserState sState;  
 07  
 08 pTokenizer = TokenizerCreate(zFilename);  
 09 pParser = ParseAlloc( malloc );  
 10 InitParserState(&sState);  
 11 while( GetNextToken(pTokenizer, &hTokenId, &sToken) ){  
 12 Parse(pParser, hTokenId, sToken, &sState);  
 13 }  
 14 Parse(pParser, 0, sToken, &sState);  
 15 ParseFree(pParser, free );  
 16 TokenizerFree(pTokenizer);  
 17 return sState.treeRoot;  
 18 }

This example shows a user-written routine that parses a file of text and returns a pointer to the parse tree. (We've omitted all error-handling from this example to keep it simple.) We assume the existence of some kind of tokenizer which is created using TokenizerCreate() on line 8 and deleted by TokenizerFree() on line 16. The GetNextToken() function on line 11 retrieves the next token from the input file and puts its type in the integer variable hTokenId. The sToken variable is assumed to be some kind of structure that contains details about each token, such as its complete text, what line it occurs on, etc.

This example also assumes the existence of structure of type ParserState that holds state information about a particular parse. An instance of such a structure is created on line 6 and initialized on line 10. A pointer to this structure is passed into the Parse() routine as the optional 4th argument. The action routine specified by the grammar for the parser can use the ParserState structure to hold whatever information is useful and appropriate. In the example, we note that the treeRoot field of the ParserState structure is left pointing to the root of the parse tree.

The core of this example as it relates to Lemon is as follows:

ParseFile(){  
 pParser = ParseAlloc( malloc );  
 while( GetNextToken(pTokenizer,&hTokenId, &sToken) ){  
 Parse(pParser, hTokenId, sToken);  
 }  
 Parse(pParser, 0, sToken);  
 ParseFree(pParser, free );  
 }

Basically, what a program has to do to use a Lemon-generated parser is first create the parser, then send it lots of tokens obtained by tokenizing an input source. When the end of input is reached, the Parse() routine should be called one last time with a token type of 0. This step is necessary to inform the parser that the end of input has been reached. Finally, we reclaim memory used by the parser by calling ParseFree().

There is one other interface routine that should be mentioned before we move on. The ParseTrace() function can be used to generate debugging output from the parser. A prototype for this routine is as follows:

ParseTrace(FILE \*stream, char \*zPrefix);

After this routine is called, a short (one-line) message is written to the designated output stream every time the parser changes states or calls an action routine. Each such message is prefaced using the text given by zPrefix. This debugging output can be turned off by calling ParseTrace() again with a first argument of NULL (0).

### Differences With YACC and BISON

Programmers who have previously used the yacc or bison parser generator will notice several important differences between yacc and/or bison and Lemon.

* In yacc and bison, the parser calls the tokenizer. In Lemon, the tokenizer calls the parser.
* Lemon uses no global variables. Yacc and bison use global variables to pass information between the tokenizer and parser.
* Lemon allows multiple parsers to be running simultaneously. Yacc and bison do not.

These differences may cause some initial confusion for programmers with prior yacc and bison experience. But after years of experience using Lemon, I firmly believe that the Lemon way of doing things is better.

## Input File Syntax

The main purpose of the grammar specification file for Lemon is to define the grammar for the parser. But the input file also specifies additional information Lemon requires to do its job. Most of the work in using Lemon is in writing an appropriate grammar file.

The grammar file for lemon is, for the most part, free format. It does not have sections or divisions like yacc or bison. Any declaration can occur at any point in the file. Lemon ignores whitespace (except where it is needed to separate tokens) and it honors the same commenting conventions as C and C++.

### Terminals and Nonterminals

A terminal symbol (token) is any string of alphanumeric and underscore characters that begins with an upper case letter. A terminal can contain lower class letters after the first character, but the usual convention is to make terminals all upper case. A nonterminal, on the other hand, is any string of alphanumeric and underscore characters than begins with a lower case letter. Again, the usual convention is to make nonterminals use all lower case letters.

In Lemon, terminal and nonterminal symbols do not need to be declared or identified in a separate section of the grammar file. Lemon is able to generate a list of all terminals and nonterminals by examining the grammar rules, and it can always distinguish a terminal from a nonterminal by checking the case of the first character of the name.

Yacc and bison allow terminal symbols to have either alphanumeric names or to be individual characters included in single quotes, like this: ')' or '$'. Lemon does not allow this alternative form for terminal symbols. With Lemon, all symbols, terminals and nonterminals, must have alphanumeric names.

### Grammar Rules

The main component of a Lemon grammar file is a sequence of grammar rules. Each grammar rule consists of a nonterminal symbol followed by the special symbol ``::='' and then a list of terminals and/or nonterminals. The rule is terminated by a period. The list of terminals and nonterminals on the right-hand side of the rule can be empty. Rules can occur in any order, except that the left-hand side of the first rule is assumed to be the start symbol for the grammar (unless specified otherwise using the %start directive described below.) A typical sequence of grammar rules might look something like this:

expr ::= expr PLUS expr.  
 expr ::= expr TIMES expr.  
 expr ::= LPAREN expr RPAREN.  
 expr ::= VALUE.

There is one non-terminal in this example, ``expr'', and five terminal symbols or tokens: ``PLUS'', ``TIMES'', ``LPAREN'', ``RPAREN'' and ``VALUE''.

Like yacc and bison, Lemon allows the grammar to specify a block of C code that will be executed whenever a grammar rule is reduced by the parser. In Lemon, this action is specified by putting the C code (contained within curly braces {...}) immediately after the period that closes the rule. For example:

expr ::= expr PLUS expr. { printf("Doing an addition...\n"); }

In order to be useful, grammar actions must normally be linked to their associated grammar rules. In yacc and bison, this is accomplished by embedding a ``$$'' in the action to stand for the value of the left-hand side of the rule and symbols ``$1'', ``$2'', and so forth to stand for the value of the terminal or nonterminal at position 1, 2 and so forth on the right-hand side of the rule. This idea is very powerful, but it is also very error-prone. The single most common source of errors in a yacc or bison grammar is to miscount the number of symbols on the right-hand side of a grammar rule and say ``$7'' when you really mean ``$8''.

Lemon avoids the need to count grammar symbols by assigning symbolic names to each symbol in a grammar rule and then using those symbolic names in the action. In yacc or bison, one would write this:

expr -> expr PLUS expr { $$ = $1 + $3; };

But in Lemon, the same rule becomes the following:

expr(A) ::= expr(B) PLUS expr(C). { A = B+C; }

In the Lemon rule, any symbol in parentheses after a grammar rule symbol becomes a place holder for that symbol in the grammar rule. This place holder can then be used in the associated C action to stand for the value of that symbol.

The Lemon notation for linking a grammar rule with its reduce action is superior to yacc/bison on several counts. First, as mentioned above, the Lemon method avoids the need to count grammar symbols. Secondly, if a terminal or nonterminal in a Lemon grammar rule includes a linking symbol in parentheses but that linking symbol is not actually used in the reduce action, then an error message is generated. For example, the rule

expr(A) ::= expr(B) PLUS expr(C). { A = B; }

will generate an error because the linking symbol ``C'' is used in the grammar rule but not in the reduce action.

The Lemon notation for linking grammar rules to reduce actions also facilitates the use of destructors for reclaiming memory allocated by the values of terminals and nonterminals on the right-hand side of a rule.

### Precedence Rules

Lemon resolves parsing ambiguities in exactly the same way as yacc and bison. A shift-reduce conflict is resolved in favor of the shift, and a reduce-reduce conflict is resolved by reducing whichever rule comes first in the grammar file.

Just like in yacc and bison, Lemon allows a measure of control over the resolution of paring conflicts using precedence rules. A precedence value can be assigned to any terminal symbol using the %left, %right or %nonassoc directives. Terminal symbols mentioned in earlier directives have a lower precedence that terminal symbols mentioned in later directives. For example:

%left AND.  
 %left OR.  
 %nonassoc EQ NE GT GE LT LE.  
 %left PLUS MINUS.  
 %left TIMES DIVIDE MOD.  
 %right EXP NOT.

In the preceding sequence of directives, the AND operator is defined to have the lowest precedence. The OR operator is one precedence level higher. And so forth. Hence, the grammar would attempt to group the ambiguous expression

a AND b OR c

like this

a AND (b OR c).

The associativity (left, right or nonassoc) is used to determine the grouping when the precedence is the same. AND is left-associative in our example, so

a AND b AND c

is parsed like this

(a AND b) AND c.

The EXP operator is right-associative, though, so

a EXP b EXP c

is parsed like this

a EXP (b EXP c).

The nonassoc precedence is used for non-associative operators. So

a EQ b EQ c

is an error.

The precedence of non-terminals is transferred to rules as follows: The precedence of a grammar rule is equal to the precedence of the left-most terminal symbol in the rule for which a precedence is defined. This is normally what you want, but in those cases where you want to precedence of a grammar rule to be something different, you can specify an alternative precedence symbol by putting the symbol in square braces after the period at the end of the rule and before any C-code. For example:

expr = MINUS expr. [NOT]

This rule has a precedence equal to that of the NOT symbol, not the MINUS symbol as would have been the case by default.

With the knowledge of how precedence is assigned to terminal symbols and individual grammar rules, we can now explain precisely how parsing conflicts are resolved in Lemon. Shift-reduce conflicts are resolved as follows:

* If either the token to be shifted or the rule to be reduced lacks precedence information, then resolve in favor of the shift, but report a parsing conflict.
* If the precedence of the token to be shifted is greater than the precedence of the rule to reduce, then resolve in favor of the shift. No parsing conflict is reported.
* If the precedence of the token it be shifted is less than the precedence of the rule to reduce, then resolve in favor of the reduce action. No parsing conflict is reported.
* If the precedences are the same and the shift token is right-associative, then resolve in favor of the shift. No parsing conflict is reported.
* If the precedences are the same the the shift token is left-associative, then resolve in favor of the reduce. No parsing conflict is reported.
* Otherwise, resolve the conflict by doing the shift and report the parsing conflict.

Reduce-reduce conflicts are resolved this way:

* If either reduce rule lacks precedence information, then resolve in favor of the rule that appears first in the grammar and report a parsing conflict.
* If both rules have precedence and the precedence is different then resolve the dispute in favor of the rule with the highest precedence and do not report a conflict.
* Otherwise, resolve the conflict by reducing by the rule that appears first in the grammar and report a parsing conflict.

### Special Directives

The input grammar to Lemon consists of grammar rules and special directives. We've described all the grammar rules, so now we'll talk about the special directives.

Directives in lemon can occur in any order. You can put them before the grammar rules, or after the grammar rules, or in the mist of the grammar rules. It doesn't matter. The relative order of directives used to assign precedence to terminals is important, but other than that, the order of directives in Lemon is arbitrary.

Lemon supports the following special directives:

* %code
* %default\_destructor
* %default\_type
* %destructor
* %extra\_argument
* %include
* %left
* %name
* %nonassoc
* %parse\_accept
* %parse\_failure
* %right
* %stack\_overflow
* %stack\_size
* %start\_symbol
* %syntax\_error
* %token\_destructor
* %token\_prefix
* %token\_type
* %type

Each of these directives will be described separately in the following sections:

#### The %code directive

The %code directive is used to specify addition C/C++ code that is added to the end of the main output file. This is similar to the %include directive except that %include is inserted at the beginning of the main output file.

%code is typically used to include some action routines or perhaps a tokenizer as part of the output file.

#### The %default\_destructor directive

The %default\_destructor directive specifies a destructor to use for non-terminals that do not have their own destructor specified by a separate %destructor directive. See the documentation on the %destructor directive below for additional information.

In some grammers, many different non-terminal symbols have the same datatype and hence the same destructor. This directive is a convenience way to specify the same destructor for all those non-terminals using a single statement.

#### The %default\_type directive

The %default\_type directive specifies the datatype of non-terminal symbols that do no have their own datatype defined using a separate %type directive. See the documentation on %type below for addition information.

#### The %destructor directive

The %destructor directive is used to specify a destructor for a non-terminal symbol. (See also the %token\_destructor directive which is used to specify a destructor for terminal symbols.)

A non-terminal's destructor is called to dispose of the non-terminal's value whenever the non-terminal is popped from the stack. This includes all of the following circumstances:

* When a rule reduces and the value of a non-terminal on the right-hand side is not linked to C code.
* When the stack is popped during error processing.
* When the ParseFree() function runs.

The destructor can do whatever it wants with the value of the non-terminal, but its design is to deallocate memory or other resources held by that non-terminal.

Consider an example:

%type nt {void\*}  
 %destructor nt { free($$); }  
 nt(A) ::= ID NUM. { A = malloc( 100 ); }

This example is a bit contrived but it serves to illustrate how destructors work. The example shows a non-terminal named ``nt'' that holds values of type ``void\*''. When the rule for an ``nt'' reduces, it sets the value of the non-terminal to space obtained from malloc(). Later, when the nt non-terminal is popped from the stack, the destructor will fire and call free() on this malloced space, thus avoiding a memory leak. (Note that the symbol ``$$'' in the destructor code is replaced by the value of the non-terminal.)

It is important to note that the value of a non-terminal is passed to the destructor whenever the non-terminal is removed from the stack, unless the non-terminal is used in a C-code action. If the non-terminal is used by C-code, then it is assumed that the C-code will take care of destroying it if it should really be destroyed. More commonly, the value is used to build some larger structure and we don't want to destroy it, which is why the destructor is not called in this circumstance.

By appropriate use of destructors, it is possible to build a parser using Lemon that can be used within a long-running program, such as a GUI, that will not leak memory or other resources. To do the same using yacc or bison is much more difficult.

#### The %extra\_argument directive

The %extra\_argument directive instructs Lemon to add a 4th parameter to the parameter list of the Parse() function it generates. Lemon doesn't do anything itself with this extra argument, but it does make the argument available to C-code action routines, destructors, and so forth. For example, if the grammar file contains:

%extra\_argument { MyStruct \*pAbc }

Then the Parse() function generated will have an 4th parameter of type ``MyStruct\*'' and all action routines will have access to a variable named ``pAbc'' that is the value of the 4th parameter in the most recent call to Parse().

#### The %include directive

The %include directive specifies C code that is included at the top of the generated parser. You can include any text you want -- the Lemon parser generator copies it blindly. If you have multiple %include directives in your grammar file the value of the last %include directive overwrites all the others.

The %include directive is very handy for getting some extra #include preprocessor statements at the beginning of the generated parser. For example:

%include {#include <unistd.h>}

This might be needed, for example, if some of the C actions in the grammar call functions that are prototyed in unistd.h.

#### The %left directive

The %left directive is used (along with the %right and %nonassoc directives) to declare precedences of terminal symbols. Every terminal symbol whose name appears after a %left directive but before the next period (``.'') is given the same left-associative precedence value. Subsequent %left directives have higher precedence. For example:

%left AND.  
 %left OR.  
 %nonassoc EQ NE GT GE LT LE.  
 %left PLUS MINUS.  
 %left TIMES DIVIDE MOD.  
 %right EXP NOT.

Note the period that terminates each %left, %right or %nonassoc directive.

LALR(1) grammars can get into a situation where they require a large amount of stack space if you make heavy use or right-associative operators. For this reason, it is recommended that you use %left rather than %right whenever possible.

#### The %name directive

By default, the functions generated by Lemon all begin with the five-character string ``Parse''. You can change this string to something different using the %name directive. For instance:

%name Abcde

Putting this directive in the grammar file will cause Lemon to generate functions named

* AbcdeAlloc(),
* AbcdeFree(),
* AbcdeTrace(), and
* Abcde().

The %name directive allows you to generator two or more different parsers and link them all into the same executable.

#### The %nonassoc directive

This directive is used to assign non-associative precedence to one or more terminal symbols. See the section on precedence rules or on the %left directive for additional information.

#### The %parse\_accept directive

The %parse\_accept directive specifies a block of C code that is executed whenever the parser accepts its input string. To ``accept'' an input string means that the parser was able to process all tokens without error.

For example:

%parse\_accept {  
 printf("parsing complete!\n");  
 }

#### The %parse\_failure directive

The %parse\_failure directive specifies a block of C code that is executed whenever the parser fails complete. This code is not executed until the parser has tried and failed to resolve an input error using is usual error recovery strategy. The routine is only invoked when parsing is unable to continue.

%parse\_failure {  
 fprintf(stderr,"Giving up. Parser is hopelessly lost...\n");  
 }

#### The %right directive

This directive is used to assign right-associative precedence to one or more terminal symbols. See the section on precedence rules or on the %left directive for additional information.

#### The %stack\_overflow directive

The %stack\_overflow directive specifies a block of C code that is executed if the parser's internal stack ever overflows. Typically this just prints an error message. After a stack overflow, the parser will be unable to continue and must be reset.

%stack\_overflow {  
 fprintf(stderr,"Giving up. Parser stack overflow\n");  
 }

You can help prevent parser stack overflows by avoiding the use of right recursion and right-precedence operators in your grammar. Use left recursion and and left-precedence operators instead, to encourage rules to reduce sooner and keep the stack size down. For example, do rules like this:

list ::= list element. // left-recursion. Good!  
 list ::= .

Not like this:

list ::= element list. // right-recursion. Bad!  
 list ::= .

#### The %stack\_size directive

If stack overflow is a problem and you can't resolve the trouble by using left-recursion, then you might want to increase the size of the parser's stack using this directive. Put an positive integer after the %stack\_size directive and Lemon will generate a parse with a stack of the requested size. The default value is 100.

%stack\_size 2000

#### The %start\_symbol directive

By default, the start-symbol for the grammar that Lemon generates is the first non-terminal that appears in the grammar file. But you can choose a different start-symbol using the %start\_symbol directive.

%start\_symbol prog

#### The %token\_destructor directive

The %destructor directive assigns a destructor to a non-terminal symbol. (See the description of the %destructor directive above.) This directive does the same thing for all terminal symbols.

Unlike non-terminal symbols which may each have a different data type for their values, terminals all use the same data type (defined by the %token\_type directive) and so they use a common destructor. Other than that, the token destructor works just like the non-terminal destructors.

#### The %token\_prefix directive

Lemon generates #defines that assign small integer constants to each terminal symbol in the grammar. If desired, Lemon will add a prefix specified by this directive to each of the #defines it generates. So if the default output of Lemon looked like this:

#define AND 1  
 #define MINUS 2  
 #define OR 3  
 #define PLUS 4

You can insert a statement into the grammar like this:

%token\_prefix TOKEN\_

to cause Lemon to produce these symbols instead:

#define TOKEN\_AND 1  
 #define TOKEN\_MINUS 2  
 #define TOKEN\_OR 3  
 #define TOKEN\_PLUS 4

#### The %token\_type and %type directives

These directives are used to specify the data types for values on the parser's stack associated with terminal and non-terminal symbols. The values of all terminal symbols must be of the same type. This turns out to be the same data type as the 3rd parameter to the Parse() function generated by Lemon. Typically, you will make the value of a terminal symbol by a pointer to some kind of token structure. Like this:

%token\_type {Token\*}

If the data type of terminals is not specified, the default value is ``int''.

Non-terminal symbols can each have their own data types. Typically the data type of a non-terminal is a pointer to the root of a parse-tree structure that contains all information about that non-terminal. For example:

%type expr {Expr\*}

Each entry on the parser's stack is actually a union containing instances of all data types for every non-terminal and terminal symbol. Lemon will automatically use the correct element of this union depending on what the corresponding non-terminal or terminal symbol is. But the grammar designer should keep in mind that the size of the union will be the size of its largest element. So if you have a single non-terminal whose data type requires 1K of storage, then your 100 entry parser stack will require 100K of heap space. If you are willing and able to pay that price, fine. You just need to know.

### Error Processing

After extensive experimentation over several years, it has been discovered that the error recovery strategy used by yacc is about as good as it gets. And so that is what Lemon uses.

When a Lemon-generated parser encounters a syntax error, it first invokes the code specified by the %syntax\_error directive, if any. It then enters its error recovery strategy. The error recovery strategy is to begin popping the parsers stack until it enters a state where it is permitted to shift a special non-terminal symbol named ``error''. It then shifts this non-terminal and continues parsing. But the %syntax\_error routine will not be called again until at least three new tokens have been successfully shifted.

If the parser pops its stack until the stack is empty, and it still is unable to shift the error symbol, then the %parse\_failed routine is invoked and the parser resets itself to its start state, ready to begin parsing a new file. This is what will happen at the very first syntax error, of course, if there are no instances of the ``error'' non-terminal in your grammar.