FMQ -- AN LL(1) ERROR-CORRECTING-PARSER GENERATOR USER GUIDE

by

Jon Mauney
Charles N. Fischer

Computer Sciences Technical Report #449

November 1981

FMQ -- an LL(1) Error-Correcting-Parser Generator
User Guide

Jon Mauney
Charles N. Fischer

FMQ accepts a context-free grammar specification and a list of correction costs, and produces tables for parsing and correcting sentences of the language so specified. It will produce tables for any LL(1) grammar, and provides a simple conflict resolution mechanism for grammars which are not LL(1). The error correction technique is described in [1] and [2].

FMQ was written, in Pascal, at the University of Wisconsin by Jon Mauney. It is designed to be easily transportable.

Research supported in part by National Science Foundation Grant MCS78-02570

FMQ -- An LL(1) Parser and Error Corrector Generator

Introduction

This report describes a pair of tools for language translation -- specifically, for parsing, and for correcting syntax errors encountered during parsing. FMQ is a table generator. It accepts an LL(1) grammar specified in the format described below, and produces tables which can be used for parsing and correcting. LLParse is a program which uses the tables produced by FMQ to parse, and perhaps correct, programs in the language so specified. It produces an output listing of the program, showing any corrections made. Although these programs perform only syntactic analysis, they provide an interface to (user supplied) semantic actions through semantic action numbers. These numbers are specified in the input to FMQ and appear in the resulting tables. LLParse has a facility for calling a semantic routine when the appropriate point in the parse is reached. Both FMQ and LLParse use the technique described in [3] to insure that the parser never performs erroneous parse actions.

This package is very similar to the LALR(1) parsing/correction package, ECP and LRParse. The difference is in the grammars accepted (and tables produced) and in the interface with semantic actions.

The corrector implemented in LLParse performs 'locally least-cost' corrections, based on the correction costs specified in the input to FMQ. Each terminal in the language has an insertion cost and a deletion cost; any time the corrector adds or deletes a symbol, it incurs the associated cost. When an error is detected, the corrector chooses the lowest cost modification to the program which will allow the parser to accept one more symbol. Note that this model of correction depends on the language to be parsed and on the costs, not on the parsing method used. Thus it is not necessary to understand the parsing method or the details of the corrector in order to anticipate the actions of the corrector, and to control them via costs. Furthermore, the correction chosen is completely independent of the parsing method. If a good set of costs is determined for an LL grammar, those costs can be used with a corresponding LALR grammar with exactly the same results.

Correction costs are heuristically determined. There is no algorithm for selecting the 'best' set of costs; there are some rules of thumb which provide a starting point. Symbols which begin a construct, such as "IF" or "BEGIN", should have relatively high costs, since inserting or deleting such a symbol could cause many more subsequent errors. Symbols which close constructs, such as "END" or ")", may have lower costs. Very low costs may be given to symbols which appear only in limited contexts, for example, ".." in Pascal. With some experimentation, costs can be adjusted to give high quality corrections in almost all situations. Sample correction costs for Pascal and Algol are given in [1].

A typical session with FMQ and LLParse, for experimentation with correction costs, might go as follows (see also the appendix):

1. create a file containing the appropriate grammar.
2. run FMQ, directing the grammar file to standard input. FMQ will send optional output and error messages, if any, to the terminal (or whatever the standard output is), and create two files, 'ptableout' and 'etableout' (or 'ptablebin' and 'etablebin', see section on output).
3. If the grammar is not acceptable to FMQ, repeat 1 and 2.
4. Run LLParse, typing in a test program, or directing a program file to standard input. LLParse will read the files created by FMQ and print a corrected listing of the program.
5. Note the corrections made by LLParse, adjust costs accordingly, and repeat 1 through 4 until the corrections made by LLParse are acceptable.

Input to FMQ

The input to FMQ has three main sections: options desired for the run, terminal symbols of the grammar, and production rules of the grammar. The general form of the input is:

```
   <comments>
*fmq
   <options>
*define
   <constant definitions>
*terminals
   <terminal specifications>
*productions
   <production specifications>
*end
   <comments>
```

An example is given in the appendix.

In the following, "symbol" will refer to a symbol in the grammar for which tables are to be generated, and "token" will refer to an entity in the input to FMQ. Double quotes (") will denote a literal string -- for example, a string as it appears in the input to FMQ. Single quotes (') will denote a logical entity, such as an option.

Symbols and tokens

The input to FMQ is divided into 'tokens' by three simple rules:

1) All tokens must be separated by one or more blanks, tabs, or end of line.
2) Tokens may not contain blanks or tabs, unless the token is surrounded by angle brackets, "<" and ">". Tokens may not run across line boundaries.
3) If a token begins with a "<", then it must end with a ">".

That is, everything in the input -- option names, reserved words, grammar symbols -- must be surrounded by white space. Angle

brackets may be used when a symbol contains white space, but they are special if and only if the first character is "<". Angle brackets appearing in any other circumstances are legal, but not special (a warning will be issued in such cases). The only other character endowed with special properties is "#", which begins all 'action symbols'. Action symbols consist of "#" followed (no spaces) by an unsigned integer or a defined constant (described below).

Upper and lower case letters are considered distinct; however, the reserved tokens and the options are recognized in either case (or a mixture). The following examples illustrate the above rules:

token	comments
ABC	OK
abc	OK, different from ABC
123	OK
< Expr >	OK
<id list>	OK
<>	OK
&:=	OK
"<"	legal, gets a warning
-->	legal, gets a warning
<not<>equal>	legal, gets a warning
much<<lessthan	legal, gets a warning
<LHS>::=<RHS>	legal, gets a warning. this is one token, not three
2<two tokens>	legal, two tokens, two warnings ("<" is only special if first)
*fmq	reserved
*FMQ	reserved, same as *fmq
*Fmq	reserved, same as *fmq
#13	legal, action symbol for routine 13
#add	legal, "add" should have been defined in the *define section
*fmq*terminals	legal, one token, no warning
<=	illegal, no closing bracket
<	ditto
<bad>token	ditto, (">" must be followed by space)

The following tokens are reserved:

*fmq	*define	*terminals	*productions	*end
::=	...	--	<Goal>	$$$

End of line is required as a terminator for specifications of terminals, productions, and constant definitions as described below. The input to FMQ is otherwise free-format.

Comments

Anything before "*fmq" or after "*end" will be considered a comment, and ignored. However, these comments must not contain

any of the above reserved tokens. Comments may also be placed at the end of any line; all text between the token "--" and the end of the line will be ignored.

Options

Following "*fmq" is a list of zero or more options, separated (as always) by blanks, tabs, or end of line. All options have the form of switches, and are enabled by including the name in the option list. An enabled option may be disabled by placing "no" before the name, without a space; e.g. to prevent construction of error correction tables, type "noerrortables". All options are initially disabled, except for 'errortables', 'checkreduce', and 'text'. Options are recognized in upper or lower case. The available options are

bnf: Print the grammar rules.

first: Print the first sets for all nonterminals.

follow: Print the follow sets of all nonterminals.

parsetable: Print the parse action table in tabular form. A number in the table indicates prediction of that production; blank entries indicate error. This table can be quite large.

checkreduce: Check whether the grammar is reduced; report all symbols which cannot produce a terminal string, and those which are not reachable from the start symbol. If the grammar is not reduced, and checkreduce is enabled, tables will not be produced. Checkreduce is normally enabled.

resolve: If the grammar given is not LL(1), generate the error tables anyway, and resolve parse conflicts pairwise in favor of the production which appeared earlier in the input. This option should be used with caution; see discussion under "Error Handling within FMQ". If 'resolve' is not enabled, computation of error-tables will be suppressed in the presence of parse conflicts.

shortline:

longline: Control the length of printed lines in human-oriented output (vocab, parsetable, etc.) Shortline causes lines to be less than 80 characters (suitable for a screen), longline is 132 (for printer). Shortline is synonymous with nolongline, and vice versa. The default is shortline.

statistics: Print assorted statistics on the grammar.

vocab: Print the symbols of the language, along with the insert and delete costs of the terminals.

text:

binary: The tables created by FMQ can be written as text (file of char) or as binary (file of integer), or both. Text output is written on files 'ptableout' and 'etableout'; binary is written on 'ptablebin' and 'etablebin'. Binary files tend to be larger, at least on a 32-bit machine (about 30% larger on the VAX under UNIX(tm)), but are usually faster to read. LLParse can read either kind. The default is 'text' and 'nobinary'.

errortables: Create the tables needed for least-cost error correction. Normally enabled.
If errortables are computed, either of the tables involved may be printed. Notice that individual tables may be printed only if they have been computed, thus the effect of the following options depends on 'errortables'. Printing of each table is controlled by an individual option:

s: Least cost-string derivable from each nonterminal.

e: Least-cost prefix to derive terminal from nonterminal. This table is rather large.

Constant definitions

The constant definition section is optional. If present, it begins with the reserved token "*define", and consists of a list of definitions, each on a separate line. Each definition has the form:

<const name> <integer value>

where <const name> is a token as described above, and <integer value> is an unsigned integer (i.e., a token containing only digits). This constant can then be used whenever an integer constant is called for: in subsequent constant definitions, in terminal insert and delete costs, and for semantic routine numbers. Note that this feature is not as nice as it seems at first, because the output listing of FMQ will use the numeric value, not the constant name.

Terminals

The reserved token "*terminals" begins the list of terminal symbols and their insert and delete costs. The specification for one terminal symbol has the form:

<terminal symbol> <insert cost> <delete cost>

where <terminal symbol> is a token as described above, and <insert cost> and <delete cost> are unsigned integers or defined constants. (Correction costs are user-supplied values used for controlling and 'fine-tuning' the actions of the corrector.) The terminal specification section consists of a list of such specifications, each on a separate line. All terminals must appear in this list.

Productions

The token "*productions" separates the terminals from the productions. The productions are specified by a list of rules, each on a separate line. Specification of one production has the form:

<lhs> ::= <rhs>

Either of <lhs> and <rhs> may be absent. <lhs> is one token representing a nonterminal symbol. If it is absent, the <lhs> of the preceding production is used. <rhs> is a string of tokens, containing the grammar symbols of the production, and action symbols indicating semantic routines to be called when the appropriate point in the production is reached. An action symbol consists of a "#" followed by an unsigned integer or defined constant, without intervening blanks. If <rhs> is absent, or contains only action symbols, then <lhs> derives the null string. <rhs> may be continued on subsequent lines by beginning those lines with the reserved token "..." (only productions may be so continued).

End

The productions are terminated by "*end". After all of the productions have been processed, the augmenting production is added. Two symbols, <Goal> and $$$, and one production

<Goal> ::= <S> $$$

are added to the grammar, where <S> is the left-hand side of the first production specified, <Goal> is the start symbol, and $$$ the end-marker. $$$ is a terminal symbol, and is assigned very high insert and delete costs, 'infinity'.

Output from FMQ

The output controlled by the above options is written to the standard Pascal file 'output'. In addition, files of tables are created. The tables for parsing are written to 'ptableout' (text) and/or 'ptablebin' (binary), and the error-correction tables are written to 'etableout' (text) and/or 'etablebin' (binary). These files may be assigned or redirected, depending on the operating system. A driver routine is provided for using the tables. Only those who wish to use the tables starting from scratch need study the table file formats, which are detailed in the appendix.

Using the Tables

LLParse is a complete program which will parse, correcting if necessary, a program, using tables generated by FMQ. It produces a formated source listing showing any corrections made. The parsing and correction tables are read from the same files as written by FMQ, and can be used immediately by LLParse. LLParse can read tables from either binary (ptablebin, etablebin) or text (ptableout, etableout) files, under compile-time control. LLParse may be used 'as is' for experimentation with FMQ and tuning of correction costs, but with the addition of a better lexi-

cal scanner and file access, it can be used as the 'front-end' of a useful system.

LLParse has a very simple lexical scanner which uses the symbol table provided by FMQ. Thus, all symbols must be typed exactly as seen by FMQ. One simple-minded exception to this is provided. An alphanumeric string which is not recognized will be considered to be token number 1; a string of digits (integer constant) will be returned as token number 2. For best results, the first two terminals in the list given to FMQ should be 'identifier' and 'constant'. The scanner divides characters into three classes: alphanumeric, spaces, and other. It looks for the longest meaningful string of characters in one class. This is fine for most languages, but symbols consisting of a mixture of these, such as ".LE." in FORTRAN, cannot be recognized. Note that angle brackets have no special meaning in LLParse.

A specialized scanner can be installed by replacing one procedure, 'scan'. In order to use the line formatting routines provided, 'scan' should get characters from the input by 'readchar'. Character lookahead can be accomplished by returning 'peeked at' characters through 'unreadchar'. If all the input routines are replaced, several procedures must be provided for interface with the error corrector: 'peek' looks ahead at input symbols without consuming them, 'deletetokens' deletes a number of symbols from the input, 'inserttokens' adds a string of symbols to the input. For more details, see the code.

The error-correction tables tend to be moderately large, about 40K bytes for Pascal. Since only a portion of the tables are needed for any one correction, we wish to leave the tables in a file, and read only the portion immediately required. Unfortunately, there is no standard way to quickly position the file pointer at a random place in the file. LLParse postions the file pointer by reading from the beginning of the file until the desired section is reached. This is slow, but acceptable for experimentation purposes, especially if 'binary' files are used. If a faster random access method is available, it can be installed by replacing procedure 'seekE' in LLParse. An index, giving the start location in the file for the tables relevant to each error-symbol, can be created by program 'makeindex'. Makeindex looks at file 'etablebin' and creates file 'index'. It will work only with binary table files.

Semantic routines may be installed by adding the appropriate calls to procedure 'announceaction'. The parser calls 'announceaction', passing the index of an action symbol, whenever an appropriate point in the parse is reached. LLParse uses the technique described in [3] to insure the immediate error detection property.

Error Handling within FMQ

Syntax errors in the input to FMQ will be handled by a locally least-cost recovery routine. Table generation will be suppressed if errors are present in the input.

An attempt to use the symbols "<Goal>" and "$$$" (the start symbol and the end-marker) will be treated as a syntax error.

If error tables are to be generated, and the insert and delete costs for a terminal are omitted, the value 1 will be supplied for both.

All terminals must be listed in the *terminals section. If any terminal is not listed, or if a nonterminal does not appear on the left of any production, the symbol will be flagged, and no tables will be generated. Similarly, a symbol declared as a terminal may not appear on the left of a production.

If the grammar specified is not LL(1), all conflicts will be reported. If the option 'resolve' is enabled, productions will be given precedence in the order of appearance (first production specified is highest). Thus the "dangling else" of Pascal and other languages can be parsed by:

```
<if stmt> ::= IF <expr> THEN <stmt> <else part>
<else part> ::= ELSE <stmt>
              ::=
```

The conflict will be resolved in favor of the first form of the statement, matching the ELSE with the most recent IF.

This resolution mechanism should be used with caution. Conflicts must be carefully examined to insure that the parse action taken is the action desired. For example, reversing the order of the above <else part> productions would be perfectly acceptable to FMQ, but would have a disastrous effect. When ELSE appears in the look-ahead, the parse action taken would be always to predict <else part> deriving the null string; the ELSE would never be accepted. Furthermore, the error corrector, which does not use the parse tables, would not provide a correction since it finds the ELSE to be acceptable. Neither FMQ nor LLParse is capable of detecting such a situation. It is the responsibility of the user to insure correctness.

Size Limits

If the grammar specified proves too large for the limits of FMQ, the program will print a message describing the limit which was exceeded, and terminate. FMQ must then be recompiled with increased limits. Normally, exceeding one limit suggests that others will also be exceeded, and increasing them all at once will save on recompilations. Notice though, that FMQ processes in order terminals, productions, parse table, error tables. Therefore, if the number of productions in the grammar is exceeded, the number of terminals must be within limits, as they have been completely processed already. Some of the dimensions of a particular grammar are easy to discover; others must be tackled by rule of thumb and trial. Easily determined are the number of terminals, number of symbols (terminals + nonterminals) and number of productions. Less simple, but still easy to estimate are the total number of symbols in the productions, and the total number of characters in all the distinct symbols.

Using FMQ

In order to run, FMQ requires two files: 'ptablein' and 'etablein'. These files provide the tables to parse and correct the input to FMQ. 'Etablein' is only used if there is a syntax error in the input. The grammar specification is read from the standard 'input' file, and human-oriented output is written to 'output'. Parse tables produced are written to 'ptableout'; error correction tables (if computed) are written to 'etableout'. The above names are the internal names as declared in the program header, and may be modified by the system environment in which the program is run.

Appendix A -- examples

Sample FMQ input

```
grammar for DCL -- desk calculator language
*fmq
statistics
vocab bnf
notext binary
*define
  one 1       -- costs
  two 2
  <do assn> 5 -- semantic action numbers
  add 12
  subtract 13
*terminals
        id          two two
        constant 1 one
        end         two 2
        ;           1 1
        :=          1 3
        (           3 4
        )           1 1
        +           1 2
        -           2 2
        *           2 2
        /           2 2
        write       3 4
        read        3 4
        ,           1 1

*productions

<prog> ::= <st list> end
<st list> ::= <st> #1 <st list tail>
        ::=
<st list tail> ::= ; <st list>
<st> ::= id #2 := <expr> #<do assn>
     ::= read  #3 ( <id list> #6 #14 )
     ::= write  #4 ( <expr list> #6 )
<expr> ::= <term> <e tail>
<e tail> ::= <add op> <term> #20 <e tail>
        ::=
<term> ::= <primary> <t tail>
<t tail> ::= <mult op>
 ... <primary> #21 <t tail>
        ::=
<primary> ::= - <primary> #7
<primary> ::= ( <expr> )
        ::= id #8
        ::= constant #9
<mult op> ::= * #10
        ::= / #11
<add op> ::= + #add
        ::= - #subtract
<id list> ::= id #15 <id list tail>
```

```
<id list tail> ::= , <id list>
        ::=
<expr list> ::= <expr> #30 <e list tail>
<e list tail> ::= , <expr list>
        ::=
*end
```

Sample output of FMQ (from input above)

```
 FMQ version 1.1, (Nov 28 1980),  date: 18 May 81

     Options for this run:
 vocab  bnf  statistics
 errortables
 Binary output files

      Vocabulary:
  terminals                costs     nonterminals
   1: id                  2    2      16: <prog>
   2: constant            1    1      17: <st list>
   3: end                 2    2      18: <st>
   4: ;                   1    1      19: <st list tail>
   5: :=                  1    3      20: <expr>
   6: (                   3    4      21: <id list>
   7: )                   1    1      22: <expr list>
   8: +                   1    2      23: <term>
   9: -                   2    2      24: <e tail>
  10: *                   2    2      25: <add op>
  11: /                   2    2      26: <primary>
  12: write               3    4      27: <t tail>
  13: read                3    4      28: <mult op>
  14: ,                   1    1      29: <id list tail>
  15: $$$               Inf  Inf      30: <e list tail>
                                      31: <Goal>


          PRODUCTIONS :
   1:    <prog>          ::= <st list> end
   2:    <st list>       ::= <st> #1 <st list tail>
   3:                    ::=
   4:    <st list tail>  ::= ; <st list>
   5:    <st>            ::= id #2 := <expr> #5
   6:                    ::= read #3 ( <id list> #6 #14 )
   7:                    ::= write #4 ( <expr list> #6 )
   8:    <expr>          ::= <term> <e tail>
   9:    <e tail>        ::= <add op> <term> #20 <e tail>
  10:                    ::=
  11:    <term>          ::= <primary> <t tail>
  12:    <t tail>        ::= <mult op> <primary> #21 <t tail>
  13:                    ::=
  14:    <primary>       ::= - <primary> #7
  15:                    ::= ( <expr> )
  16:                    ::= id #8
  17:                    ::= constant #9
  18:    <mult op>       ::= * #10
  19:                    ::= / #11
```

```
 20:   <add op>        ::= + #12
 21:                   ::= - #13
 22:   <id list>       ::= id #15 <id list tail>
 23:   <id list tail>  ::= , <id list>
 24:                   ::=
 25:   <expr list>     ::= <expr> #30 <e list tail>
 26:   <e list tail>   ::= , <expr list>
 27:                   ::=
 28:   <Goal>          ::= <prog> $$$
```

```
The grammar is LL(1).
```

```
statistics for this grammar:
  15 terminals in grammar
  31 symbols in all
  28 productions
prodspcptr =    96
stringptr  =   241; biggest insertspaceptr =   32
Timing :
    Input  took            2.83 seconds.
    Parser tables took     0.50 seconds.
    Error tables  took     0.50 seconds.
whole thing took     3.88 seconds
```

Sample input to llparse (for dcl grammar)

```
    j := i + x / y ;
    i := x y + / z ( ) ;
    write ( i , j ; )
    a ( i , j ) ;
    read ( x y , ) ;
    write := n ;
end
```

Sample output of llparse (on input above)
Insertions are underlined with "*",
deletions enclosed in "{" and "}"

```
 LL Parse, using binary files
version 1.1, (Oct 7 1980)
     1:     j := i + x / y ;
**   2*     i := x + y + constant / z + ( constant ) ;
error             *     ********     *   ********
**   3*     write ( i , j ) ; {)}
error                        *
**   4*     a := ( i {,}+   j ) ;
error         **         *
**   5*     read ( x , y , id ) ;
error                  *     **
**   6*     write {:=}(   n ) ;
error                  *     *
     7: end
 accepted
    7 lines in program
   12 errors ( calls to corrector)
   11 tokens inserted;    3 tokens deleted.
```

Appendix B -- tables

Parsing Tables

The formats of the output files are the same, whether they are created as 'text' or 'binary' files. In 'binary' files, character values are written as 'ord' of the character.

The file 'ptableout' or 'ptablebin' contains the following tables: the right hand sides of all productions, the parser action table, a list of symbols which can derive the null string, symbolic representations of all symbols in the grammar.

The format of the file is:

header line: The first line gives the sizes of the various tables. It contains: the number of terminals in the language (numterms), the number of symbols in the grammar (numsymbols), the number of productions in the grammar (numprods), the size of the character string for the symbol images (stringsize), a flag indicating whether error-correction tables were produced. The value of the flag is a character 'T' if the tables were produced, and 'F' if not.

productions: The right hand sides of all the productions are given; all right hand sides are reversed, and can be pushed onto the parse stack in the order given. Each production consists of a length, followed by the corresponding number of symbols. Action symbols are included in the right hand sides, encoded as the negative of the number given in the grammar.

parse table: The LL(1) predictions for each nonterminal symbol are given by the parse table. The predictions for each nonterminal are stored in a list of pairs of integers. The first pair in a list is a zero followed by the number of the nonterminal symbol. The subsequent pairs have the form

terminal prediction

where 'prediction' is the production predicted when 'terminal' appears in the lookahead. The list is terminated by the start of the next list. The table is terminated by "0 0". In order to make immediate error detection possible, predictions which will could be erroneous are marked; they are written as the negative of the production predicted. When such a prediction is encountered, the stack should be checked to insure that the prediction is, in fact, correct. (LLParse does this.) The details of this algorithm are explained in [3]. If error-correction tables are not generated, parse actions will not be marked.

epsilon productions: A list of those productions which are "epsilon productions", i.e. which have no symbols (except action symbols) on the right hand side. The list consists of a length, followed by that many integers representing the production numbers.

string table: The symbol table information is in two parts. First is an index, consisting of numsymbols pairs of integers. The first integer of each pair is the starting point of the symbol in the character string, the second is the length of the symbol. Following the index are string-size characters, 132 per line.

Error-Correction Tables

The file etableout, if created, contains the additional information necessary to find the locally least-cost correction to any error.

The error tables have the following form:

header line: one line containing 3 integers: the number of terminals in the language, the number of symbols (terminals + nonterminals), and the maximum correction cost.

delete costs: The deletion costs of the terminal symbols

S table: The least cost strings derivable from the nonterminals

E table: The least cost prefix to derive a terminal from a nonterminal.

The format of the error table file is summarized in the following chart. In the chart, a name corresponds to an integer in the file. (string)*<name> means that the contents of the parentheses (string) are repeated <name> times. (string)* means that string is repeated an unknown number of times (the list is terminated by -1). "name:" labels a logical division of the file, and does not appear physically in the file. Comments to the chart (don't appear in the file) are enclosed in '{' and '}'. "-1" represents -1.

```
header : Numterminals numsymbols infinity
delete : ( deletecost )*numterminals (insertcost?)
S table: ( cost length ( insertsymbol )*length
      or   -1   {if same as previous}
         )*numsymbols-numterminals { number of nonterminals }
E table: (
                0 terminal
                ( symbol cost length (string)*length
                )*
         )*numterminals
            0 0
```

Index Table

The program 'makeindex' reads file 'etablebin' and creates file 'index'. 'Index' is a file of integer, containing the starting point within 'etablebin' of the E entries for each nonterminal.

References

[1] Fischer, Charles N., Donn R. Milton, and Jon Mauney, "A locally least-cost LL(1) error corrector," Tech. Report #371, University of Wisconsin (August 1979).

[2] Fischer, Charles N., Donn R. Milton, and Sam B. Quiring, "Efficient LL(1) error correction and recovery using only insertions," Acta Informatica 13, 2, pp. 141-154 (1980).

[3] Mauney, Jon and Charles N. Fischer, "An Improvement to Immediate Error Detection in Strong LL(1) Parsers," Information Processing Letters 12, 5, (October 1981).