ECP -- AN ERROR-CORRECTING-PARSER GENERATOR USER GUIDE

by

Jon Mauney
Charles N. Fischer

Computer Sciences Technical Report #450

November 1981

ECP -- an Error-Correcting-Parser Generator User Guide

Jon Mauney
Charles N. Fischer

ECP 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 LALR(1) grammar, and provides a simple conflict resolution mechanism for grammars which are not LALR(1). The error correction technique is described in [1].

ECP was written at the University of Wisconsin by Jon Mauney. The source code is in Pascal, and designed to be readily transportable.

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

ECP -- an Error-Correcting-Parser Generator

Introduction

This report describes a pair of tools for language translation -- specifically, for parsing and for correcting syntax errors encountered during parsing. ECP is a table generator. It accepts an LALR(1) grammar specified in the format given below, and produces tables which can be used for parsing and correcting. LRParse is a program which uses the tables produced by ECP 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 routine numbers. These numbers are specified in the input to ECP and appear in the resulting tables. LRParse has a facility for calling a semantic routine when the associated language construct is recognized.

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

The corrector implemented in LRParse performs ´locally least-cost´ corrections, based on the correction costs specified in the input to ECP. 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 LALR grammar, those costs can be used with a corresponding LL 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 ECP and LRParse, for experimentation with correction costs, might go as follows (see also the appendix):

1. create a file containing the appropriate grammar.
2. run ECP, directing the grammar file to standard input. ECP 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 the section on output).
3. If the grammar is not acceptable to ECP, repeat 1 and 2.
4. Run LRParse, typing in a test program, or directing a program file to standard input. LRParse will read the files created by ECP and print a corrected listing of the program.
5. Note the corrections made by LRParse, adjust costs accordingly, and repeat 1 through 4 until the corrections made by LRParse are acceptable.

Input to ECP

The input to ECP 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>
   *ecp
      <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 ECP. Double quotes (") will denote a literal string, for example, a string as it appears in the input to ECP. Single quotes (´) will denote a logical entity, such as an option.

Symbols and tokens

The input to ECP 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).

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)
*ecp	reserved
*ECP	reserved, same as *ecp
*Ecp	reserved, same as *ecp
*ecp*terminals	legal, one token
<=	illegal, no closing bracket
<	ditto
<bad>token	ditto, (">" must be followed by space)

The following tokens are reserved:

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

End of line is required as a terminator for specifications of terminals, productions and constant definitions (see description below). The input to ECP is otherwise free-format.

Comments

Anything before "*ecp" 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 "*ecp" 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´. Be warned that most of the output options will create a large amount of output for a grammar the size of a real programming language. The figures in parentheses after each output option give the order of magnitude of the lines printed, and the actual number of lines for a Pascal grammar, with 69 terminals, 258 productions, and 226 states. Options are recognized in upper or lower case. The available options are

bnf: Print the grammar rules. (number of productions, Pascal=262)

cfsm: Print the characteristic finite-state machine for the grammar, with LALR(1) lookahead sets. (number of configurations, Pascal=2893)

links: If links and cfsm are both enabled, then for each item of a state, list the successor items. (number of configurations, Pascal=5809 + size of cfsm)

first: Print the first sets for all nonterminals. (number of nonterminals, Pascal=175)

parsetable: Print the parse action table in tabular form. In the parse table, an unmarked entry signifies a transition, and gives the number of the succeeding state. An entry marked "L" means a lookahead reduction by the production number given, and "R" means a simple reduction. A blank indicates error. (number of states times number of terminals, Pascal=1855)

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 LALR(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´. 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, cfsm, 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 in gory detail. In any case the number of productions, symbols and states is reported. If statistics is enabled, additional information such as average number of basis items, length of paths through the closure graph, and execution times will also be printed. (constant, 25)

vocab: Print the symbols of the language, along with the insert and delete costs of the terminals. (number of symbols, Pascal=124)

text:

binary: The tables created by ECP 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. LRParse can read either kind. The default is ´text´ and ´nobinary´.

errortables: Create the tables needed for least-cost error correction. If errortables are computed, any 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´. All the tables are large. Printing of each table is controlled by an individual option:

s: Least cost-string derivable from each nonterminal. (number of symbols, pascal=149)

e: Least-cost prefix to derive terminal from nonterminal. (number of nonterminals times number of terminals, pascal=2483)

graph: Least cost paths through the closure graph, and completers for basis items. (number of items squared, pascal=7631)

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 ECP 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). The terminal section consists of a list of such specifications, each on a separate line. The terminal symbols will be assigned numeric encodings in the order they are listed, beginning with 1. All terminals must appear in this list.

Productions

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

```
    <lhs> ::= <rhs> <semantic routine #>
```

Any of <lhs>, <rhs>, and <semantic routine #> 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 (separated by blanks). If <rhs> is absent, 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).
<semantic routine #> has the form:

```
    ## <number>
```

and specifies the semantic routine to be called when the production is recognized. <number> is either an unsigned integer or a defined constant. If absent, zero will be used.

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´. The augmenting production is given a semantic routine number of -1.

Output from ECP

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 provided for using the tables is described in the next section. 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

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

LRParse has a very simple lexical scanner which uses the symbol table provided by ECP. Thus, all symbols must be typed exactly as seen by ECP. 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 ECP 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 are not special characters to this scanner.

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 are typically very large, over 100K bytes for Pascal. Since only a small portion of the tables are in use at any time, we wish to leave the tables in the file, and only read the portion required. Unfortunately, there is no standard way to quickly position the file pointer to a random place in the file. LRParse simulates this capability by reading from the beginning of the file up to the proper point. This is slow, but acceptable for experimentation, especially if ´binary´ files are used. Genuine random file access, installed in procedures ´seekE´ and ´seekState´, should increase the speed of the corrector. Makeindex is a program which creates an index file (called ´index´) for the correction tables, for use by LRParse with random file access. Makeindex works only with ´binary´ files.

Semantic routines may be installed by added the appropriate calls to procedure "callsemantics". After a production is recognized, the parser calls "callsemantics", passing the associated semantic routine number, as specified in the input to ECP. LRParse insures that no actions need ever be undone.

Error Handling within ECP

Syntax errors in the input to ECP 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 LALR(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 <stmt>
               ::= IF <expr> THEN <stmt>
```

The conflict will be resolved in favor of the first form of the statement, matching the ELSE with the most recent IF. A conflict between two configurations with the same underlying production will be resolved in favor of the reduction. This ambiguous grammar, then:

```
     E ::= E + E
       ::= id
```

will parse expressions involving + and id, enforcing left association, since reduction will always be chosen over shifting.

This resolution mechanism should be used with caution. Conflicts should be examined carefully to insure that the action taken by the parser is the action desired. In the above IF statement grammar, for example, a reversal of the two productions would be acceptable to the generator, causing a precedence of reduction over shifting. This would have a disastrous effect on parsing, as the ELSE would <u>never</u> be accepted. Furthermore, since the error corrector uses the original grammar, and not the LALR parse tables, it finds the ELSE acceptable and it would not provide a correction. Neither ECP nor LRParse is capable of detecting such a situation, so it is the responsibility of the user to insure correctness.

Size Limits

If the grammar specified proves too large for the limits of ECP, the program will print a message describing the limit which was exceeded, and terminate. ECP 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 ECP processes in order terminals, productions, parse table, error tables. Therefore, if the number of states in the CFSM is exceeded, the number of terminals and productions 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 not difficult to estimate are the total number of symbols in the productions, the number of paths through the closure graph, and the total number of characters in all the distinct symbols. Candidates for rule of thumb are number of states (approx. number of productions), number of items (for pascal, 12 times number of states (=2500)), number of links (for pascal, 2 times number of items (=5000)). The size of the insert table is about 300 for Pascal.

LRParse has similar limits. The actual requirements for these limits are included in the ´statistics´ printed by ECP.

Using ECP

In order to run, ECP requires two files: ´ptablein´ and ´etablein´. These files provide the tables to parse and correct the input to ECP. ´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´ (or ´ptablebin´); error correction tables (if computed) are written to ´etableout´ (or ´etablebin´). 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.

LRParse gets its tables from ´ptableout´ and ´etableout´ (or ´ptablebin and ´etablebin´, determined at compile time); tables written by ECP can be immediately read by LRParse. LRParse reads a program from standard ´input´ and produces a formated listing on ´output´.

Appendix A -- Examples

Sample ECP input

```
grammar for DCL, Desk Calculator Language
*ecp
     vocab bnf
     binary notext  -- only gen. binary files
*define
     one 1   -- costs
     two 2
     three 3
     <do assn> 2 -- semantic actions
     add 5
     subtract 6
*terminals
        id       two two
        constant one 1
        end      2 two
        ;        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 list> ; <st>
          ::= <st>
<st> ::= id := <expr> ## <do assn>
     ::= <read> ( <id list> ) ## 3
     ::= <write> ( <expr list> ) ## 4
     ::=
<expr> ::= <expr> + <term> ## add
     ::= <expr> - <term>
         ... ## subtract
     ::= <term> ## 7
<term> ::= <term> * <primary> ## 8
     ::= <term> / <primary> ## 9
     ::= <primary> ## 10
<primary> ::= - <primary> ## 11
     ::= ( <expr> )
     ::= id ## 1
     ::= constant ## 12
<write> ::= write ## 20
<read> ::= read ## 21
<id list> ::= <id list> , id ## 23
     ::= id ## 24
```

```
<expr list> ::= <expr list>
 ... , <expr>
     ::= <expr>
*end
```

<u>Sample</u> <u>ECP</u> <u>Output</u> (<u>from</u> <u>above</u> <u>input</u>)

```
 ECP version 1.1 (Nov 14 1980),  date: 18 May 81
   Options for this run:
 vocab  bnf
 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: <expr>
   5: :=              1    3      20: <read>
   6: (               3    4      21: <id list>
   7: )               1    1      22: <write>
   8: +               1    2      23: <expr list>
   9: -               2    2      24: <term>
  10: *               2    2      25: <primary>
  11: /               2    2      26: <Goal>
  12: write           3    4
  13: read            3    4
  14: ,               1    1
  15: $$$           Inf  Inf

 [semantics]           Productions:
         1: <prog>      ::= <st list> end
         2: <st list>   ::= <st list> ; <st>
         3:             ::= <st>
 [   2]  4: <st>        ::= id := <expr>
 [   3]  5:             ::= <read> ( <id list> )
 [   4]  6:             ::= <write> ( <expr list> )
         7:             ::=
 [   5]  8: <expr>      ::= <expr> + <term>
 [   6]  9:             ::= <expr> - <term>
 [   7] 10:             ::= <term>
 [   8] 11: <term>      ::= <term> * <primary>
 [   9] 12:             ::= <term> / <primary>
 [  10] 13:             ::= <primary>
 [  11] 14: <primary>   ::= - <primary>
        15:             ::= ( <expr> )
 [   1] 16:             ::= id
 [  12] 17:             ::= constant
 [  20] 18: <write>     ::= write
 [  21] 19: <read>      ::= read
 [  23] 20: <id list>   ::= <id list> , id
 [  24] 21:             ::= id
        22: <expr list> ::= <expr list> , <expr>
        23:                 ::= <expr>
 [ -1]  24: <Goal>          ::= <prog> $$$

 The grammar is LALR(1).

 statistics for this grammar:
   15 terminals in grammar
   26 symbols in all
   24 productions
   27 states in CFSM, with   130 configurations
 whole thing took     5.94 seconds
```

Sample Input to LRParse (for DCL tables)

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

Output of LRParse on above program
Insertions are underlined with ´*´, deletions are enclosed in ´{´ and ´}´.

```
 LR Parse, using binary tables
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 -- table formats

Parsing Tables

The file 'ptableout' (or 'ptablebin') contains the following tables: the encoded parse action table, the lengths of the right-hand sides of the productions, the left-hand side symbols of the productions, the semantic routine numbers associated with the productions, a symbol table giving the character representations of the symbols of the grammar, and the entry symbol for each state of the cfsm. The symbols of the grammar are encoded as integers. The terminals are numbered, starting with one, in the order they are listed in the terminal specification section. The end marker, "$$$", is the highest numbered terminal. The nonterminals are assigned numbers in the order they are encountered in the grammar, beginning one higher than the end marker. The goal symbol, "<Goal>", is the highest numbered nonterminal.

The format of the file is:

<u>header line</u>: The first line gives the sizes of the various tables. It contains: number of states of the cfsm (numstates), number of symbols in the grammar (numsymbols), number of productions (numprods), size of the character string for the symbol table (stringsize), number of non-error parse table entries, and a flag (errortables) indicating whether error-correction tables were created for this grammar. The flag is one character, a "T" if correction tables were created and "F" if not.

<u>parse actions</u>: The parse are given as lists of symbol/action pairs for each state. Each list is headed by a pair zero/state-number; the action table is terminated by a pair of zeroes. If a symbol does not appear in the list for some state, then the parse action for that state and symbol is error. The actions are encoded as follows:

```
   n > 2000: lookahead reduction by production p=n-2000
          pop rhslength(p) states from the stack
          and do not consume the current input symbol.
   2000 > n > 1000: simple reduction by production p=n-1000.
          the current symbol completes production p.
          pop rhslength(p)-1 states and consume the symbol.
   1000 > n > 0: transition to state n.
          push n onto the stack. Consume the symbol.
```

<u>rhs lengths</u>: Following the action matrix are numprods integers, indicating the number of symbols on the right-hand side of the corresponding productions.

<u>lhs</u>: Next are numprods integers giving the symbols on the left-hand side of each production.

<u>semantic numbers</u>: Numprods integers, giving the semantic rou-

tine numbers associated with each production.

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 stringsize characters, 80 per line. In the binary format, the characters are written one per word, using ´ord´.

entry symbols: Finally, there are numstates integers, giving the symbol which was shifted on entry to each state.

Error-Correction Tables

The file ´etableout´ (or ´etablebin´), 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 6 integers: the number of terminals in the language, the number of symbols (terminals + nonterminals), the number of states in the cfsm, the maximum number of items in any one state, the maximum number of basis items in any one state, and the integer which represents an "infinite" cost.

correction costs: The insertion and 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.

state information: For each state: the number of items and number of basis items in the state; for each basis item in the state, the state and item number of the ´successor´, the item reached by shifting the next symbol, and the ´completer´, the symbols which follow the dot in the configuration; for each closure item in the state, the state and item number of the ´successor´, the number of ´path labels´ through the closure graph, followed by strings of symbols representing those paths.

The closure graph or labeled path information is written as a series of strings of symbols. Each such string represents a least-cost path from a closure item to some other item in the state; the symbols correspond to the labels along the path. A set of several strings is needed to represent all the distinct paths from one item. The first n paths, where n is the number of basis items in the state, represent paths to a basis item. If a basis item is unreachable, a -1 appears in place of the string. Since each path is least-cost, if a derivation of the error symbol is found, no cheaper derivation can be found on this path; if the path leads to a basis item, then it represents the least-cost path to that basis item.

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 numstates
                      maxnumbasis maxnumitems infinity
costs  : ( insertcost deletecost )*numterminals
S table: ( cost length ( insertsymbol )*length
      or   -1   {if same as previous}
         )*numsymbols-numterminals { number of nonterminals }
E table: ( 0 terminal 0
           ( nonterminal cost length
                  ( insertsymbol )*length
           )*
         )*numterminals
            0 0 0
states : ( numbasis numitems
          ( succstate succitem length
               (string)*length
          )*numbasis
          ( succstate succitem numpaths
                 or numpaths = -1
                          if all paths same as previous item
            ( length (string)*length
               or -2 if unreachable item
            )*numpaths {if numpaths <> -1}
          )*numitems-numbasis
         )*numstates
```

References

[1] Fischer, Charles N., Bernard A. Dion, and Jon Mauney, "A Locally Least-Cost LR Error-Corrector," Tech. Report 363, to appear in ACM TOPLAS, University of Wisconsin-Madison (August 1979).