LR(1) Parser Generator Hyacc

X. Chen[1], **D. Pager**[1]
[1]Department of Information and Computer Science, University of Hawaii at Manoa, Honolulu, HI, USA

Abstract - *The space and time cost of LR parser generation is high. Robust and effective LR(1) parser generators are rare to find. This work employed the Knuth canonical algorithm, Pager's practical general method, lane-tracing algorithm, and other relevant algorithms, implemented an efficient, practical and open-source parser generator Hyacc in ANSI C, which supports full LR(0)/LALR(1)/LR(1) and partial LR(k), and is compatible with Yacc and Bison in input format and command line user interface. In this paper we introduce Hyacc, and give a brief overview on its architecture, parse engine, storage table, precedence and associativity handling, error handling, data structures, performance and usage.*

Keywords: Hyacc, LR(1), Parser Generator, Compiler, Software tool

1 Introduction

The canonical LR(k) algorithm [1] proposed by Knuth in 1965 is a powerful parser generation algorithm for context-free grammars. It was potentially exponential in time and space to be of practical use. Alternatives to LR(k) include the LALR(1) algorithm used in parser generators such as Yacc and later Bison, and the LL algorithm used by parser generators such as ANTLR. However, LALR and LL are not as powerful as LR. Good LR(k) parser generator remains scarce, even for the case k = 1.

This work has developed Hyacc, an efficient and practical open source full LR(0)/LALR(1)/LR(1) and partial LR(k) parser generation tool in ANSI C. It is compatible with Yacc and Bison. The LR(1) algorithms employed are based on 1) the canonical algorithm of Knuth [1], 2) the lane-tracing algorithm of Pager [2][3], which reduces parsing machine size by splitting from a LALR(1) parsing machine that contains reduce-reduce conflicts, and 3) the practical general method of Pager [4], which reduces parsing machine size by merging compatible states from a parsing machine obtained by Knuth's method. The LR(0) algorithm used in Hyacc is the traditional one. The LALR(1) algorithm used in Hyacc is based on the first phase of the lane-tracing algorithm. LR(0) and LALR(1) are implemented because Pager's lane-tracing algorithm depends on these as the first step. The LR(k) algorithm is called the edge-pushing algorithm [5] based on recursively applying the lane-tracing process, and works for a subclass of LR(k) grammars. As a side optimization, Hyacc also implemented the unit production elimination algorithm of Pager [6] and its extension [5].

2 The Hyacc Parser Generator

2.1 Overview

Hyacc is pronounced as "HiYacc". It is an efficient and practical parser generator written from scratch in ANSI C, and is easy to port to other platforms. Hyacc is open source. Version 0.9 was released in January 2008 [7]. Version 0.95 was released in April 2009. Version 0.97 was released in January 2011.

Hyacc is released under the GPL license. But the LR(1) parse engine file hyaccpar and LR(k) parse engine file hyaccpark are under the BSD license so that the parser generators created by Hyacc can be used in both open source and proprietary software. This addresses the copyright problem that Richard Stallman discussed in "Conditions for Using Bison" of his Bison manuals [8][9].

The algorithms employed by Hyacc are listed in the introduction.

Hyacc is compatible to Yacc and Bison in its input file format, ambiguous grammar handling and error handling. These directives from Yacc and Bison are implemented in Hyacc: %token, %left, %right, %expect, %start, %prec. Hyacc can be used together with the lexical analyser Lex. It can generate rich debug information in the parser generation process, and store these in a log file for review.

If specified, the generated parser can record the parsing steps in a file, which makes it easy for debugging and testing. It can also generate a Graphviz input file for the parsing machine. With this input, Graphviz can draw an image of the parsing machine.

2.2 Architecture

Hyacc first gets command line switch options, then reads from the grammar input file. Next, it creates the parsing machine according to different algorithms as specified by the command line switches. Then it writes the generated parser to y.tab.c, and optionally, y.output and y.gviz. y.tab.c is the parser with the parsing machine stored in arrays. y.output contains all kinds of information needed by the compiler developer to understand the parser generation process and the parsing machine. y.gviz can be used as the input file to Graphviz to generate a graph of the parsing machine.

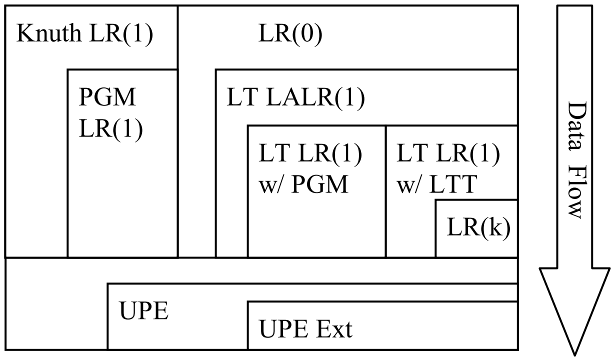

Fig. 1. Relationship of algorithms from the point of view of grammar processing [1]

Fig. 1 shows how the algorithms used in Hyacc are structured from the point of view of grammar processing. Input grammars can be processed by the merging path on the left, first by the Knuth canonical algorithm and stop here, or be further processed by Pager's PGM algorithm.

Input grammars can also be processed by the splitting path on the right. First the LR(0) parsing machine is generated. Next the LALR(1) parsing machine is generated by the first phase of the lane-tracing algorithm. If reduce-reduce conflicts exist, this is not a LALR(1) grammar, and the second phase of lane-tracing is applied to generate the LR(1) parsing machine. There are two methods for the second phase of lane-tracing. One is based on the PGM method [4], the other is based on the lane table method [10]. If LR(1) cannot resolve all the conflicts, this may be a LR(k) grammar and the LR(k) process is applied.

The generated LR(1) parsing machine may contain unit productions that can be eliminated by applying the UPE algorithm and its extension.

Fig. 2 shows the relationship of the algorithms from the point of view of implementation, i.e., how one algorithm is based on the other.

[1] Knuth LR(1) – Knuth canonical algorithm, PGM LR(1) – Pager's practical general method, LT LALR(1) – LALR(1) based on lane-tracing phase 1, LT LR(1) w/ PGM – lane-tracing LR(1) algorithm based on Pager's practical general method, LT LR(1) w/ LTT – lane-tracing LR(1) algorithm based on Pager's lane table method, UPE – Pager's unit production elimination algorithm, UPT Ext – Extension algorithm to Pager's unit production elimination algorithm.

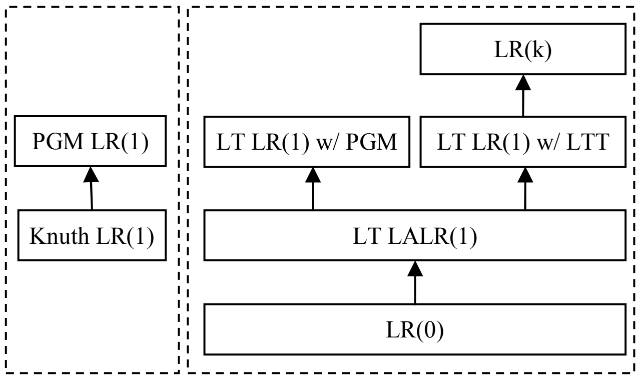

Fig. 2. Relationship of algorithms from the point of view of implementation

2.3 The LR(1) Parse Engine

Similar to the yaccpar file of Yacc, the hyaccpar file is the parse engine of Hyacc. The parser generation process embeds the parsing table into hyaccpar. How the hyaccpar LR(1) parse engine works is shown in Algorithm 1.

Algorithm 1: The hyaccpar LR(1) parse engine algorithm.[2]

```
Initialization:
push state 0 onto state_stack;

while next token is not EOF do {
  S ← current state;
  L ← next token/lookahead;
  A ← action of(S, L) in parsing table;
  if A is shift then {
    push target state on state_stack,
      pop lookahead symbol;
    update S and L;
  } else if A is reduce then {
    output code for this reduction;
    r1 ← LHS symbol of reduction A;
    r2 ← RHS symbol count of A;
    pop r2 states from state_stack,
      update current state S;
    A_tmp ← action for (S, r1);
    push target goto state A_tmp to
      state_stack;
  } else if A is accept then {
    if next token is EOF then {
      is valid accept, exit;
    } else {
      is error, error recovery or exit;
    }
  } else {
    is error, do error recovery;
  }
}
```

In Algorithm 1, a state stack is used to keep track of the current status of traversing the state machine. The parameter 'S' or current state is the state on the top of the

[2] LHS – Left Hand Side, RHS – Right Hand Side.

state stack. The parameter 'L' or lookahead is the symbol used to decide the next action from the current state. The parameter 'A' or action is the action to take, and is found by checking the parsing table entry (S, L). '$\leftarrow$' denotes assignment operation. This parse engine is similar to the one used in Yacc, but there are variation in the details, such as the storage parsing table, as discussed in the next section.

TABLE 1. STORAGE ARRAYS FOR THE PARSING MACHINE IN HYACC PARSE ENGINE.

Array name	Explanation
yyfs[]	List the default reduction for each state. If a state has no default reduction, its entry is 0. Array size = *n*.
yyrowoffset[]	The offset of parsing table rows in arrays yyptblact[] and yyptbltok[]. Array size = *n*.
yyptblact[]	Destination state of an action (shift/goto/reduce/ accept). If yyptblact[i] is positive, action is 'shift/goto', If yyptblact[i] is negative, action is 'reduce', If yyptblact[i] is 0, action is 'accept'. If yyptblact[i] is -10000000, labels array end. Array size = *p*.
yyptbltok[]	The token for an action. If yyptbltok[i] is positive, token is terminal, If yyptbltok[i] is negative, token is non-terminal. If yyptbltok[i] is -10000001, is place holder for a row. If yyptbltok[i] is -10000000, labels array end. Array size = *p*.
yyr1[]	If the LHS symbol of rule i is a non-terminal, and its index among non-terminals (in the order of appearance in the grammar rules) is *x*, then yyr1[i] = -*x*. If the LHS symbol of rule i is a terminal and its token value is *t*, then yyr1[i] = *t*. Note yyr1[0] is a placeholder and not used. Note this is different from yyr1[] of Yacc or Bison, which only have non-terminals on the LHS of its rules, so the LHS symbol is always a non-terminal, and yyr1[i] = x, where x is defined the same as above. Array size = *r*.
yyr2[]	Same as Yacc yyr2[]. Let x[i] be the number of RHS symbols of rule i, then yyr2[i] = x[i] << 1 + y[i], where y[i] = 1 if production i has associated semantic code, y[i] = 0 otherwise. Note yyr2[0] is a placeholder and not used. This array is used to generate semantic actions. Array size = *r*.
yynts[]	List of non-terminals. This is used only in debug mode. Array size = number of non-terminals + 1.
yytoks[]	List of tokens (terminals). This is used only in debug mode. Array size = number of terminals + 1.
yyreds[]	List of the reductions. Note this does not include the augmented rule. This is used only in debug mode. Array size = *r*.

2.4 Storing the Parsing Table

2.4.1 Storage tables

The following describes the arrays that are used in hyaccpar to store the parsing table.

Let the parsing table have *n* rows (states) and m columns (number of terminals and non-terminals). Assuming there are *r* rules (including the augmented rule), and the number of non-empty entries in the parsing table is *p*. Table 1 lists all the storage arrays and explains their usage.

2.4.2 Complexity

Suppose in state i there is a token j, we can find if an action exists by looking at the yyptbltok table from yyptbltok[yyrowoffset[i]] to yyptbltok[yyrowoffset[i+1]-1]:

i) If yyptbltok[k] == j, yyptblact[k] is the associated action;
ii) If yyptblact[k] > 0, this is a 'shift/goto' action;
iii) If yyptblact[k] < 0, is a reduction, then use yyr1 and yyr2 to find number of states to pop and the next state to goto;
iv) If yyptblact[k] == 0 then it is an 'accept' action, which is valid when j is the end of an input string.

The space used by the storage is: 2n + 2p + 3r + (m + 2). In most cases the parsing table is a sparse matrix. In general, 2n + 2p + 3r + (m + 2) < n*m.

For the time used, the main factor is when searching through the yyptbltok array from yyptbltok[yyrowoffset[i]] to yyptbltok[yyrowoffset[i+1]-1]. Now it is linear search and takes O(n) time. This can be made faster by binary search, which is possible if terminals and non-terminals are sorted alphabetically. Then time complexity will be O(ln(n)). It can be made such that time complexity is O(1), by using the double displacement method which stores the entire row of each state. That would require more space though.

2.4.3 Example

An example is given to demonstrate the use of these arrays to represent the parsing table. Given grammar G1:

E $\rightarrow$ E + T | T
T $\rightarrow$ T * a | a

The parsing table is shown in Table 2. Here the parsing table has n = 8 rows, m = 6 columns, and r = 5 rules (including the augmented rule). The actual storage arrays in the hyaccpar parse engine are shown in Table 3.

Array yyfs[] lists the default reduction for each state: state 3 has default reduction on rule 4, and state 7 has default reduction on rule 3.

Array yyrowoffset[] defines the offset of parsing table rows in arrays yyptblact[] and yyptbltok[]. E.g., row 1 starts at offset 0, row 2 starts at offset 3.

Array yyptblact[] is the destination state of an action. The first entry is 97, which can be seen in the yytoks[] array. The second entry is 1, which stands for non-terminal E. And as we see in the parsing table, entry (0, a) has action s3, entry (0, E) has action g1, thus in yyptblact[] we see correspondingly the first entry is 3, and the second entry is 1. Entry -10000000 in both yyptblact[] and yyptbltok[] labels the end of the array. Entry 0 in yyptblact[] labels the accept action. Entry 0 in yyptbltok[] stands for the token end marker $. Entry -10000001 in yyptbltok[] labels that this state (row in parsing table) has no other actions but the default reduction. -10000001 is just a dummy value that is never used, and servers as a place holder so yyrowoffset[] can have a corresponding value for this row.

Entries of array yyr1[] and array yyr2[] are defined as in Table 1, and it is easy to see the correspondence of the values.

2.5 Handling Precedence and Associativity

The way that Hyacc handles precedence and associativity is the same as Yacc and Bison. By default, in a shift/reduce conflict, shift is chosen; in a reduce/reduce conflict, the reduction whose rule appears first in the grammar is chosen. But this may not be what the user wants. So %left, %right and %nonassoc are used to declare tokens and specify customized precedence and associativity.

2.6 Error Handling

Error handling is the same as in Yacc. There have been abundant complaints about the error recovery scheme of Yacc. We are concentrating on LR(1) algorithms instead of better error recovery. Also we want to keep compatible with Yacc and Bison. For these reasons we keep the way that Yacc handles errors.

2.7 Data Structures

A symbol table is implemented by hash table, and uses open-chaining to store elements in a linked list at each bucket. The symbol table is used to achieve O(1) performance for many operations. All the symbols used in the grammar are stored as a node in this symbol table. Each node also contains other information about each symbol. Such information are calculated at the time of parsing the grammar file and stored for later use.

TABLE 2. PARSING TABLE FOR GRAMMAR G1

State	*$*	+	*	*a*	*E*	*T*
0	0	0	0	s3	g1	g2
1	a0	s4	0	0	0	0
2	r2	r2	s5	0	0	0
3	r4	r4	r4	0	0	0
4	0	0	0	s3	0	g6
5	0	0	0	s7	0	0
6	r1	r1	s5	0	0	0
7	r3	r3	r3	0	0	0

TABLE 3. STORAGE TABLES IN HYACC LR(1) PARSE ENGINE FOR GRAMMAR G1

```
#define YYCONST const
typedef int yytabelem;

static YYCONST yytabelem yyfs[] = {0, 0,
0, -4, 0, 0, 0, -3};

static YYCONST yytabelem yyptbltok[] = {
97, -1, -2, 0, 43, 0, 43, 42, -10000001, 97,
-2, 97, 0, 43, 42, -10000001, -10000000};

static YYCONST yytabelem yyptblact[] = {
3, 1, 2, 0, 4, -2, -2, 5, -4, 3,
6, 7, -1, -1, 5, -3, -10000000};

static YYCONST yytabelem yyrowoffset[] = {
0, 3, 5, 8, 9, 11, 12, 15, 16};

static YYCONST yytabelem yyr1[] = {
     0,    -1,    -1,    -2,    -2};
static YYCONST yytabelem yyr2[] = {
     0,     6,     2,     6,     2};

#ifdef YYDEBUG

typedef struct {char *t_name; int t_val;}
yytoktype;

yytoktype yynts[] = {
  "E", -1,
  "T", -2,
  "-unknown-", 1  /* ends search */
};
yytoktype yytoks[] = {
  "a", 97,
  "+", 43,
  "*", 42,
  "-unknown-", -1  /* ends search */
};
char * yyreds[] = {
  "-no such reduction-"
  "E : 'E' '+' 'T'",
  "E : 'T'",
  "T : 'T' '*' 'a'",
  "T : 'a'",
};
#endif /* YYDEBUG */
```

Linked list, static arrays, dynamic arrays and hash tables are used where appropriate. Sometimes multiple data structures are used for the same object, and which one to use depends on the particular circumstance.

In the parsing table, the rows index the states (e.g., row 1 represents actions of state 1), and the columns stand for the lookahead symbols (both terminals and non-terminals) upon which shift/goto/reduce/accept actions take place. The parsing table is implemented as a one dimensional integer array. Each entry [row, col] is accessed as entry [row * column size + col]. In the parsing table positive numbers are for 'shift', negative numbers are for 'reduce', -10000000 is for 'accept' and 0 is for 'error'.

There is no size limit for any data structures. They can grow until they consume all the memory. However, Hyacc artificially sets an upper limit of 512 characters for the maximal length of a symbol.

2.8 Performance

The performance of Hyacc is compared to other LR(1) parser generators. Menhir [11] and MSTA [12] are both very efficiently and robustly implemented. Table 4 and Table 5 show the running time comparison of the three parser generators on C++ and C grammars. The speeds are similar. MSTA, implemented in C++, is a handy choice for industry users. It does not use reduced-space LR(1) algorithms though, thus always results in larger parsing machines. Menhir uses Pager's PGM algorithm, but is implemented in Caml, which is not so popular in industry. Therefore Hyacc should be a favorable choice.

TABLE 4. RUNNING TIME (SEC) COMPARISON OF MENHIR, MSTA AND HYACC ON C++ GRAMMAR.

	Knuth LR(1)	PG MLR(1)	LALR(1)
Menhir	1.97	1.48	N/A
MSTA	5.32	N/A	1.17
Hyacc	3.53	1.78	1.10

TABLE 5. RUNNING TIME (SEC) COMPARISON OF MENHIR, MSTA AND HYACC ON C GRAMMAR.

	Knuth LR(1)	PG MLR(1)	LALR(1)
Menhir	1.64	0.56	N/A
MSTA	0.92	N/A	0.13
Hyacc	1.05	0.42	0.19

2.9 Usage

From the sourceforge.net homepage of Hyacc [7] a user can download the source packages for unix/linux and windows, and the binary package for windows. All the instructions on installation and usage are available in the included readme file. It is very easy to use, especially for users familiar with Yacc and/or Bison.

Hyacc is a command line utility. To start hyacc, use: "hyacc input_file.y [-bcCdDghKlmnoOPQRStvV]". The input grammar file input_file.y has the same format as those used by Yacc/Bison.

The meanings of some of the command line switches are briefly introduced here. '-b' specifies the prefix to use for all hyacc output file names. The default is y.tab.c as in Yacc. If '-c' is specified, no parser files (y.tab.c and y.tab.h) will be generated. This is used when the user only wants to use the -v and -D options to parse the grammar and check the y.output log file. '-D' is used with a number from 0 to 15 (e.g., -D7) to specify the details to be included into the y.output log file during the parser generation process. '-g' says that a Graphviz input file should be generated. '-S' means to apply LR(0) algorithm. '-R' applies LALR(1) algorithm. '-Oi' where i = 0 to 3 applies the Knuth canonical algorithm and the practical general method with different optimizations. '-P' applies the lane-tracing algorithm based on the practical general method. '-Q' applies the lane-tracing algorithm based on the lane table method. '-K' applies the LR(k) algorithm. '-m' shows man page.

For more usage of the Hyacc parser generator, interested users can refer to the Hyacc usage manual.

3 Related Work

Pager's practical general method has been implemented in some other parser generators. Some examples are LR (1979, in Fortran 66, at Lawrence Livermore National Laboratory) [13], LRSYS (1985, in Pascal, at Lawrence Livermore National Laboratory) [14], LALR (1988, in MACRO-11 under RSX-11) [15], Menhir (2004, in Caml) [11] and the Python Parsing module (2007, in Python) [16].

The lane-tracing algorithm was implemented by Pager (1970s, in Assembly under OS 360) [3]. But no other available implementation is known.

For other LR(1) parser generators, the Muskox parser generator (1994) [17] implemented Spector's LR(1) algorithm [18][19]. MSTA (2002) [12] took a splitting approach but the detail is unknown. Commercial products include Yacc++ (LR(1) was added around 1990, using splitting approach loosely based on Spector's algorithm) [20][21] and Dr. Parse (detail unknown) [22]. Most recently an IELR(1) algorithm [23][24] was proposed to provide LR(1) solution to non-LR(1) grammars with specifications to solve conflicts, and the authors implemented this as an extension of Bison.

4 Conclusions

In this work we investigated LR(1) parser generation algorithms and implemented a parser generator Hyacc, which supports LR(0)/LALR(1)/LR(1) and partial LR(k). It has been released to the open source community. The usage of Hyacc is highly similar to the widely used LALR(1) parser generators Yacc and Bison, which makes it easy to learn and use. Hyacc is unique in its wide span of algorithms coverage, efficiency, portability, usability and availability.

5 References

[1] Donald E. Knuth. On the translation of languages from left to right. Information and Control, 8(6):607 – 639, 1965.

[2] David Pager. The lane tracing algorithm for constructing LR(k) parsers. In Proceedings of the fifth annual ACM symposium on Theory of computing, pages 172 – 181, Austin, Texas, United States, 1973.

[3] David Pager. The lane-tracing algorithm for constructing LR(k) parsers and ways of enhancing its efficiency. Information Sciences, 12:19 – 42,

[4] David Pager. A practical general method for constructing LR(k) parsers. Acta Informatica, 7:249 – 268, 1977.

[5] Xin Chen. Measuring and Extending LR(1) Parser Generation. PhD thesis, University of Hawaii, August 2009.

[6] David Pager. Eliminating unit productions from LR parsers. Acta Informatics, 9:31 – 59, 1977.

[7] Xin Chen. LR(1) Parser Generator Hyacc, January 2008. http://hyacc.sourceforge.net.

[8] Charles Donnelly, Richard Stallman. Bison, The YACC-compatible Parser generator (for Bison Version 1.23), 1993.

[9] Charles Donnelly, Richard Stallman. Bison, The YACC-compatible Parser generator (for Bison Version 1.24), 1995.

[10] David Pager. The Lane Table Method Of Constructing LR(1) Parsers. Technical Report No. ICS2009-06-02, University of Hawaii, Information and Computer Sciences Department, 2008. http://www.ics.hawaii.edu/research/tech-reports/LaneTableMethod.pdf/view

[11] Francois Pottier and Yann Regis-Gianas. Parser Generator Menhir, 2004. http://cristal.inria.fr/~fpottier/menhir/

[12] Vladimir Makarov. Toolset COCOM & scripting language DINO, 2002. http://sourceforge.net/projects/cocom

[13] Charles Wetherell and A. Shannon. LR automatic parser generator and LR(1) parser. Technical Report UCRL-82926 Preprint, July 1979.

[14] LRSYS, 1991. http://www.nea.fr/abs/html/nesc9721.html

[15] Algirdas Pakstas, 1992. http://compilers.iecc.com/comparch/article/92-08-109

[16] Parser Generator Parsing.py: An LR(1) parser generator with CFSM/GLR drivers, 2007. http://compilers.iecc.com/comparch/article/07-03-076

[17] Boris Burshteyn. MUSKOX Algorithms, 1994. http://compilers.iecc.com/comparch/article/94-03-067

[18] David Spector. Full LR(1) parser generation. ACM SIGPLAN Notices, p.58 – 66, 1981.

[19] David Spector. Efficient full LR(1) parser generation. ACM SIGPLAN Notices, 23(12):143 – 150, 1988.

[20] Yacc++ and the Language Objects Library, 1997 – 2004. http://world.std.com/~compres

[21] Chris Clark, 2005. http://compilers.iecc.com/comparch/article/05-06-124

[22] Parser Generator Dr. Parse. http://www.downloadatoz.com/software-development_directory/dr-parse

[23] Joel E. Denny, Brian A. Malloy. IELR(1): practical LR(1) parser tables for non-LR(1) grammars with conflict resolution. Proceedings of the 2008 ACM symposium on Applied computing, p.240 – 245, 2008.

[24] Joel E. Denny, Brian A. Malloy, The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution, Science of Computer Programming, v.75 n.11, p.943 – 979, November, 2010.