A Flexible Table Driven LR(1) Parser

Stoyan Bonev

Abstract: *The implementation of a flexible table driven LR(1) parser is presented in this paper. The LR(1) parsing algorithm is based on modified bottom up strategy described in [1] and it stays the same for all grammars. A parser with control tables that are initialized within the source text has no flexibility because each new LR(1) parser implementation needs to recompile the source text in order to adjust its data control structures like terminal symbols, non terminal symbols, grammar rules and control table. To avoid this drawback a configuration text edited file is to be created and read as input to adapt the parser before the syntax analysis process starts execution.*

Key words: *Syntax Analysis, Table Driven LR(1) Parser, LR-situation, Configuration File.*

INTRODUCTION

There exist a variety of top-down and bottom-up parsing strategies applicable in the area of theory and practice of compiler writing. The LR(k) approaches like LR(1), SLR(1), LALR(1) are known to be the most powerful syntax directed table controlled parsing strategies [2, 3 and 4]. A modified LR(1) parsing method was presented in [1] and its program implementation was discussed in [5]. Central term in this approach is the concept of LR-situation or LR-position – a structure composed of two components X,m. X stays for any terminal or non terminal character and m is a specific subscript indicating the location of the X character within the productions of the grammar. The usefulness of this subscript is that it helps to distinguish during the parsing process the same concrete terminal or non terminal character in case it appears more than once within the simple phrases (right sides of the grammar productions). To illustrate this, the context free grammar (CFG) of simplified arithmetic expressions and its modification as a grammar with LR-situations are presented in the three-column *Table 1*. The grammar rule number is to be used as a subscript attached to all terminal and/or non terminal symbols in the right side of the corresponding production.

Table 1

Rule No	Arithmetic expressions grammar	Modified arithmetic expressions grammar with LR-situations
0	E' −> ^ E #	E' −> ^ E0 #
1	E −> E + T	E −> E1 +1 T1
2	E −> T	E −> T2
3	T −> T * F	T −> T3 *3 F3
4	T −> F	T −> F4
5	F −> a	F −> a5
6	F −> (E)	F −> (6 E6)6

The advantage of this enriched CFG grammar notation is that when dealing with LR-situations grammar during the parsing process and operating the non terminal F for example, one can differ it as the last character of the phrase T * F (grammar rule 3) or as a sentential form presenting the sole phrase F (grammar rule 4).

The way of enumerating numbers to all terminal and non terminal symbols follows the only condition: assignment of a unique number. The above example illustrates that there are no duplicated symbols in the right sides of the grammar rules. That is why the grammar rule number was selected to be assigned to each of the symbols as a component of the LR-situation.

In this way the bottom-up parsing algorithm instead of pushing/popping terminal

and/or non terminal symbols will operate – pushing and popping LR-situations that constitute the stack alphabet. The example above supposes the following set { ^, E0, E1, +1, T1, T2, T3, *3, F3, F4, (6, E6,)6 } to be considered as a stack alphabet.

For more details on the algorithm and its implementation see [1, 5].

PROGRAM IMPLEMENTATION

The drawback of the LR(1) parser implementation presented in [5] is that all grammar data structures for symbol presentation of terminal symbols, non terminal symbols, grammar rules, starting non terminal, bottom stack symbol and LR situations control table were initialized within the source text of the parser. This approach has no flexibility and leads to loss of generalization because each new parser implementation requires recompilation of the parser source text substituting the control data structures before that.

In order to avoid this shortcoming the following was done. Instead of initializing the grammar data structures mentioned above, within the source text of the parser, a configuration text edited file was created to include and store the symbol presentation of the grammar structures. During execution time configuration file should initially be read before the parsing process starts. So, we get an advantage: to implement a new LR(1) parser, we will need to create a new configuration file and no recompilation of the parser source text is required. The skeleton parser may be adapted to operate as LR(1) syntax analyzer for any formal language whose grammar is specified in accordance with the requirements to compose the configuration file. So, the parsing program from [5] is transferred to more general reference as an adjustable table-driven parser that operates based on bottom up LR(1) syntax analysis strategy.

The following *Table 2* illustrates the configuration file components and their correspondence to the data structures utilized in the parser.

Table 2

Configuration file component	Data structure
[alphabet – set of terminal symbols]	char TermCh[]; int NumTerm;
[syntax categories – set of non terminal symbols]	char NonTermCh[]; int NumNonTerm;
[starting non terminal symbol]	char StartNonTerm;
[productions – set of grammar rules]	struct GramRule { char LeftPart; char *RightPart; int ProdLen; }; GramRule gr[]; int NumGrRules;
[LR situations control table]	struct CtrlTabElm { char action; // actions 'S'-Shift 'R'-Reduce // 'E'-Error 'A'-Accept int LRposOrRuleNum; // LR_situation nmr or //Grammar rule number }; CtrlTabElm CtrlTab[][]; int NumRowCtrlTab; int NumColCtrlTab;
[bootom stack marker]	char BottomStack;
[input file terminal delimiter]	char EofDelimiter;

The basic algorithm is spilt in two principle functions: to read the configuration file and to activate the parser.

```
int main(int argc, char *argv[])
{
  ReadCfgFle(argc, argv);
  LrkParse();
  return 0;
}
```

The configuration file contents that is to be used to adjust the syntax analyzer as a simplified arithmetic expressions parser follows:

```
[terminal characters]
5
a      +      *      (      )

[nonterminal characters]
3
E      T      F

[starting nonterminal]
E

[grammar rules]
6
 E  −>  E+T,  ,3  ,
 E  −>  T,    ,1  ,
 T  −>  T*F,  ,3  ,
 T  −>  F,    ,1  ,
 F  −>  a,    ,1  ,
 F  −>  (E),  ,3

[control table   rows x columns]
12   9
//       a        +            *            (         )          E          T          F          #
{ SHIFT5,       ERR,       ERR, SHIFT6,       ERR, SHIFT8, SHIFT8, SHIFT4,       ERR }, // #,0
{    ERR,    SHIFT1,       ERR,    ERR,       ERR,    ERR,    ERR,    ERR, ACCEPT }, // E,8
{ SHIFT5,       ERR,       ERR, SHIFT6,       ERR,    ERR, SHIFT9, SHIFT4,       ERR }, // +,1
{    ERR,REDUCE2,   SHIFT3,    ERR,REDUCE2,   ERR,    ERR,    ERR,REDUCE2 },// T,8
{    ERR,REDUCE1,   SHIFT3,    ERR,REDUCE1,   ERR,    ERR,    ERR,REDUCE1 },// T,9
{ SHIFT5,       ERR,       ERR, SHIFT6,       ERR,    ERR,    ERR, SHIFT3,       ERR },// *,3
{    ERR,REDUCE3,REDUCE3,    ERR,REDUCE3,   ERR,    ERR,    ERR,REDUCE3 },// F,3
{    ERR,REDUCE4,REDUCE4,    ERR,REDUCE4,   ERR,    ERR,    ERR,REDUCE4 },// F,4
{    ERR,REDUCE5,REDUCE5,    ERR,REDUCE5,   ERR,    ERR,    ERR,REDUCE5 },// a,5
{ SHIFT5,       ERR,       ERR, SHIFT6,       ERR, SHIFT9, SHIFT8, SHIFT4,       ERR },// (,6
{    ERR,    SHIFT1,       ERR,    ERR,    SHIFT6,    ERR,    ERR,    ERR,       ERR },// E,9
{    ERR,REDUCE6,REDUCE6,    ERR,REDUCE6,   ERR,    ERR,    ERR,REDUCE6 },// ),6

[bottom stack marker]
#

[input file terminal delimiter]
#
```

The input data for the control table only is to be commented: The number of rows (12) and number of columns (9) are explicitly declared. The control table columns (array

ColumnTitle[]) are labeled with all terminal symbols (array TermCh[]) followed by all non terminal symbols (array NonTermCh[]) followed by the terminal delimiter of the input stream (EofDelimiter).

The control table rows (array RowTitle[]) are labeled using LR situations X,m that appear at the end of each line with data for the control table rows.

The data elements for the control table entries have the form

SHIFTxx or
REDUCEyy or
ERR or
ACCEPT

SHIFT, REDUCE, ERR and ACCEPT are recognized as case insensitive reserved words to specify the Shift,xx or Reduce,yy or Error or Accept actions of the parsing algorithm.

xx is a string concatenated to SHIFT and composed of digits specifying the LR situation to create pushing to the stack the symbol, naming the control table column.

yy is a string concatenated to REDUCE and composed of digits specifying the grammar production number to apply in order to substitute (reduce) the right side of rule numbered yy configured on the top of the stack with its left hand side non terminal symbol.

The scanning process of the input data file and the configuring of the control table is based on the *strtok* run time library function which operates in the following loop:

```
fgets(InpBuf, 120, fp);      // read the number of rows and columns
  NumRowCtrlTab = atoi(strtok(InpBuf," "));
  NumColCtrlTab = atoi(strtok(NULL," "));
  fgets(InpBuf, 120, fp);  // read the comment line
  for (i=1; i<=NumRowCtrlTab; i++)
  {
      fgets(InpBuf, 120, fp); // read the current row of the control table
      ptr = strtok(InpBuf," {},/");   // the first component to be processed
      if(strnicmp(ptr, "SHIFT", 5)==0){  ptr=ptr+5;
                                         CtrlTab[i][1].action = 'S';
                                         CtrlTab[i][1].LRposOrRuleNum = atoi(ptr);
                                         }
      else if(strnicmp(ptr, "REDUCE", 6)==0){ ptr=ptr+6;
                                         CtrlTab[i][1].action = 'R';
                                         CtrlTab[i][1].LRposOrRuleNum = atoi(ptr);
                                         }
      else if(strnicmp(ptr, "ERR", 3)==0){
                                         CtrlTab[i][1].action = 'E';
                                         CtrlTab[i][1].LRposOrRuleNum = -5;
                                         }
      else if(strnicmp(ptr, "ACCEPT", 6)==0){
                                         CtrlTab[i][1].action = 'A';
                                         CtrlTab[i][1].LRposOrRuleNum = 0;
                                         }

      for (j=2; j<=NumColCtrlTab; j++)  // the rest part of the line includes all components
      {
            ptr = strtok(NULL," {},/");   // inclues all components  minus  1
            if(strnicmp(ptr, "SHIFT", 5)==0){ ptr=ptr+5;
                                         CtrlTab[i][j].action = 'S';
                                         CtrlTab[i][j].LRposOrRuleNum = atoi(ptr);
```

```
                                        }
            else if(strnicmp(ptr, "REDUCE", 6)==0){ ptr=ptr+6;
                                                CtrlTab[i][j].action = 'R';
                                                CtrlTab[i][j].LRposOrRuleNum = atoi(ptr);
                                                }
            else if(strnicmp(ptr, "ERR", 3)==0) {
                                                CtrlTab[i][j].action = 'E';
                                                CtrlTab[i][j].LRposOrRuleNum = -5;
                                                }
            else if(strnicmp(ptr, "ACCEPT", 6)==0){
                                                CtrlTab[i][j].action = 'A';
                                                CtrlTab[i][j].LRposOrRuleNum = 0;
                                                }
        }  // end of j loop

        // now  to read and store the RowTitle[i].Symb and RowTitle[i].LRpos componennts
        ptr = strtok(NULL," {},/");
        RowTitle[i].Symb = *ptr;
        ptr = strtok(NULL," {},/");
        RowTitle[i].LRpos = atoi(ptr);

    }  // end of i loop
```

The parser is implemented in two versions based on the MS Visual C++ analogy: Release and Debug.

In case of input strings accepted as valid sentences:

The Release version output results the reverse order for the right canonical analysis of the recognized input string.

The Debug version output is composed as a detailed report tracing the parsing process step by step and displaying the top stack value, the row and column indexes for the control table, the control table entry and a marker (*) indicating a grammar rule was reduced and registered as an element of the final result – sequence of production numbers constituting the right canonical analysis in reverse order.

In case of input strings rejected as invalid sentences:

Both Release and Debug versions generate error messages diagnosing the errors reasons like "Invalid input character", or "Control table Error entry", or "Control table Shift,I expected".

The protocol of the parsing process for the elementary a + a input string follows:

```
Enter input string to parse:   a   +   a
Compressed input string to parse is:a+a#

Top_stack     Pair_to_compare     Ctrl_Table_value     Output
-------------------------------------------------------------------
#0              #0,a                SHIFT ,5
a5              a5,+                REDUCE,5          *
#0              #0,F                SHIFT ,4
F4              F4,+                REDUCE,4          *
#0              #0,T                SHIFT ,8
T8              T8,+                REDUCE,2          *
#0              #0,E                SHIFT ,8
E8              E8,+                SHIFT ,1
+1              +1,a                SHIFT ,5
```

a5	a5,#	REDUCE,5	*
+1	+1,F	SHIFT ,4	
F4	F4,#	REDUCE,4	*
+1	+1,T	SHIFT ,9	
T9	T9,#	REDUCE,1	*
#0	#0,E	SHIFT ,8	
E8	E8,#	ACCEPT	

Successful parsing, input string accepted
The reverse order right canonical syntax analysis result is: 5 4 2 5 4 1
Enter new input string to parse or CTRL/Z to quit:

FUTURE DEVELOPMENT

The potential possible improvement is to implement a control table generator using the symbol presentation of the grammar rules as input. In this way the user will not need to manually synthesize the control table. The adjustable LR(1) parser discussed above will convert into a user friendly computer aided software tool similar to the well known popular YACC, Bison etc. utilities reading symbol grammar presentation and generating the source text of bottom up LR parsers. For the compatibility reasons the input grammar specification should be modified to follow the format required by YACC.

CONCLUSION

The LR(1) parser presented was written in C++ and executes as MS-DOS application as well as a console application under Windows. It serves as a sample demo program illustrating details of bottom up-syntax analysis strategy when teaching students in language processors and compiler theory courses.

REFERENCES

[1] Yankov B., Translators and Operating Systems, Sofia, Tehnika Publ., 1993, (in Bulgarian).

[2] Tremblay J.P., P.Sorenson, The Theory and Practice of Compiler Writing, McGraw Hill Book Company, 1985.

[3] Aho A., R.Sethi, J.Ullman, Compilers, Principles, Techniques and Tools, Addison Wesley Publishing Company, 1986.

[4] Aho A., J.Ullman, The Theory of Parsing, Translation and Computing, vol. 1,2, Prentice Hall, 1973.

[5] Bonev S., Implementation of LR(1) Parsers, Proc of the 16th Int. Conf. Systems for Automation of Engineering and Research SAER 2002, Varna, Bulgaria, pp 149-153.

ABOUT THE AUTHOR

Assoc.Prof. Stoyan Bonev, PhD, Department of Computer Science, The American University in Bulgaria, Phone: +359 73 888 416, E-mail: sbonev@aubg.bg.