A Method for Efficient Compactation of LALR(1) Parsing Tables

Roberto da Silva Bigonha
(bigonha@dcc.ufmg.br)
Departament of Computer Science, Federal University of Minas Gerais
Caixa Postal, 702
30.161 - Belo Horizonte - Minas Gerais - Brazil
and

Mariza Andrade da Silva Bigonha
(mariza@dcc.ufmg.br)
Departament of Computer Science, Federal University of Minas Gerais
Caixa Postal, 702
30.161 - Belo Horizonte - Minas Gerais - Brazil

Summary

A new compactation method for LALR(1) parsing tables is presented and discussed. The proposed method is based on intrinsics properties of the parsing method, and allows LALR(1) tables needs of space to be substantially reduced without compromising the table accessing time.

Keywords: Parsing tables, compactation, LALR(1) methods.

Introduction

An SLR(1), LALR(1) and possibly LR(1) table [5, 6] for a programming language of real size like **PASCAL** has more than 300 rows (states) and 100 columns (terminal and nonterminal symbols). Assuming, as an example, a grammar with 100 productions, at least 11 bits would be necessary to encode each entry of the LALR(1) table.

The resulting 300x100 matrix would require at least 330.000 bits, which is about 40 Kbytes of working memory! In practice, the required memory would be even greater because the size of each entry is not necessarily multiple of the smallest addressable memory unit of most existing architectures. In this paper, we present a method for encoding LALR(1) parsing table that allows an expressive reduction on memory requirements without compromising accessing time.

LALR(1) Parsing Structure

LALR(1) parsers consist of a driver program, a stack and a parsing table. The stack is used to store states of the parser. The LALR(1) parsing table is a bi-dimensional array, where the row's indices represent states names, usually integer numbers, and the column's indices are terminal and nonterminal symbols of the underlying grammar.

Usually, these grammar symbols are also represented as integer numbers to increase array indexing efficiency in real implementations. For technical reasons, LALR(1) tables are divided in two parts, namely, **ACTION** and **GOTO** [3]. In the **ACTION** part, indices of columns represent terminal symbols, and in the **GOTO** part, they represent

	ACTION						GOTO		
	a	b	d	e	f	$	A	B	C
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		e6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figure 1: LALR(1) Parsing Table

nonterminal symbols. Each entry in the **ACTION** part may be one of the following actions:

1. `shift s`, where `s` is the number of a state. This action means *push state* `s`.

2. `reduce p`, where `p` is the number of a production of the form `A` $\rightarrow \omega$, `A` is a non-terminal symbol and ω a sequence of grammar symbols. This action means *reduce according the* `p`th *production.*

3. `accept`, which indicates successful completion of the parsing.

4. `error`, which indicates syntactic error and is represented by a blank entry in the table.

The entries in the **GOTO** part of the LALR(1) table contains either a state number or is empty. Figure 1 shows an LALR(1) parsing table, where a, b, d, e, f are terminal symbols; A, B, C are nonterminal symbols, and $ is a special symbol, which denotes the end of input file; **sk** represents the `shift k` action; **rp**, the `reduce p` action , where `p` is the number of a production, and **acc** is the `accept` action.

Figure 2 presents the basic parsing algorithm which assumes that the LALR(1) table is stored as the two matrices **ACTION** and **GOTO**. The procedure `Lex` returns, at each invocation, the next symbol of the input as a pair `(u, v)`, where `u` identifies the symbol and `v` denotes its associated value.

Encoding of LALR(1) Parsing Tables

LALR(1) tables are generally sparse in the sense that most of their entries are empty, i.e., they denote `error` actions. Thus, a compactation strategy that takes this fact into account may produce greater space efficiency. Actually, the problem to be solved is the design of an encoding scheme that makes possible to represent a sparse matrix `A(i,j)` in the smallest amount of memory space possible without compromising accessing time. A possible solution would be to use a hashing technique, in which the access key would be the pair `(i,j)`. Another solution is a ring representation as described by Knuth [4]. These solutions are certainly feasible. However, the use of the intrinsic properties of LALR(1) tables leads to better results.

For instance, the compactation scheme proposed by Aho and Ullman [2, 3] follows this approach. Aho e Ullman [2, 3] suggest a method that makes possible to achieve an expressive reduction in memory requirements with respect to the direct representation of LALR(1) table as a matrix.

Their method is based on the fact that most of the entries in an LALR(1) table represents `error` actions, and that LALR(1) parsers have the **viable prefix property**. A syntactic analysis method is said to have the property of the viable prefix, if all syntatic errors are detected as soon as the sequence of terminal and nonterminal symbols formed by the elements in the parse stack and the next input token does not establish a prefix of a valid sentencial form of the language being analised [1]. Strictly speaking, SLR(1) and LALR(1) parsers do not have this property because after the LR(1) parser has indicated a syntactic error in a given input, these two methods still can perform some reductions in the contents of the stack before they detect the error.

However, it can be shown that, even in this case, the input symbol that caused the LR(1) parser to report the syntactic error is never shifted. So, an error condition detectable by an LR(1) parser is not removed if extra reductions are allowed to happen. It is guaranteed that the first `shift` state (a state containing only `shift` actions) that follows

```
 var parsing : boolean;
     u       : token;
     v       : value;
     ACTION  : array[state, token] of action;
     GOTO    : array[state, nonterminal] of state;
     SIN     : array[0 .. max] of state;
     top     : 0 .. max;
     s, k    : state;
     A       : nonterminal;
     p       : production_number;
begin
    top      := 0;
    SIN[top] := initial_state;
    parsing  := TRUE;
    Lex(u,v);
    s := initial_state;
    while parsing loop
        case ACTION[s,u] of
             shift k  : top := top + 1;
                        SIN[top] := k;
                        s := k;
                        Lex(u,v);
             reduce p : A   := left hand side of production p;
                        top := top - size of the RHS of production p;
                        s   := GOTO[SIN[top],A];
                        top := top + 1;
                        SIN[top] := s
             accept   : parsing := FALSE;
             error    : error recovery routine
        end
    end
end
```

Figure 2: LALR(1) Parser

these reductions will detect the error. Therefore, at the expense of possibly complicating the error recovery algorithm, `error` entries may be eliminated and replaced by one of the `reduce` actions occurring in the state. In fact, in order to save space, the more frequent `reduce` action in a given row should be encoded only once, and should be selected only when any other cannot be applied for a given input symbol.

Moreover, in those states in which there exist only `shift` actions, `error` actions can be encoded just once, provided it is selected only when no other action is aplicable. In this encoding scheme, the **ACTION** table is stored by a collection of lists, each of them corresponding to a state in the table. A list consists of sequence of pairs of the form `(column, action)`, which associate terminal symbols to parsing actions. Each list is ended by a special pair of the form `(any, action)`. The element `column` denotes a (lookahead) terminal symbol, `any` represents all terminal symbols that are not on the list, and `action` describes the associated parsing action. The pair `(any, action)` specifies an action to be accomplished no matter what the current input symbol is. In states containing only `shift` actions, a pair of the form `(any, error)` must end the associated list; and in states containing `reduce` actions, the last pair must have the form `(any, reduce p)`, where `reduce p` represents the most frequent `reduce` action in the current row (state). The high degree of memory space compression achieved with this encoding scheme results from the fact that, in general, more than 90% of an LALR(1) table entries are `error` actions, and that `reduce` actions in a same row usually refer to a same production. Since execution of `reduce` actions does not cause any problems — at least, from the point of view of syntax — when the next input symbol is syntactically invalid, all occurrences of the most frequent `reduce` action in a row can be replaced by a single pair of the form `(any, reduce p)`, and no `error` actions are needed.

Another important opportunity for space optimization, which is explored by this method, is the elimination of repeated lists. Identical rows are encoded only once, and the resulting list is shared by the associated states. Although additional memory space is necessary to store pointers that associate lists to corresponding states or to nonterminal symbols, and to encode row and columns numbers in each pair, the reduction in the memory occupancy is claimed to be greater than 90% of the area of the original matrix representation [2].

The disadvantage of this scheme, when compared to the direct method, is related to the accessing time to the encoded table. Now, each parser transition needs to perform an indirect addressing operation to obtain the address of the list associated to a state (or to the nonterminal symbol), and the lists must be searched sequentially. However, in practice, the lists are small, and the increase in accessing time represents only a small fraction of the total compilation time, which is certainly a very low price in view of the economy of space attained.

The Proposed Method

The new method here proposed for compacting LALR(1) tables takes the Aho and Ullman's scheme as a starting point, and aims to decrease even more memory needs without affecting accessing time. The first step consists of removing the pointers that relate states to lists of pairs. In the direct method, states are used as indices of rows of the LALR(1) matrix. Consequently, it was convenient to encode states as consecutive integer numbers, starting from a base value, usually zero. Using lists, on the other hand, all that is needed is that from a given state the address of the its associated list could be determined.

Since each state corresponds a unique list, and if one guarantees that each list corresponds to a unique state, addresses of lists can be used to identify states. This guaranty can be easily achieved if repeated lists are not eliminated from the encoded representation of the parsing tables. In addition to expressive economy of space that results from the elimination of the pointers, there also exists an extra gain regarding the access time to the lists, since a level of indirect addressing has been abolished. In fact, a yet bigger reduction in memory demand comes from a property inherent to the underlying parsing method.

In an LALR(1) parser, every transition reaching a state has the same label (column). This property, which can be easily derived from the definition of the **GOTO** set [2, 3], makes possible to remove labels from transitions, that is, from pairs of the form `(column, shift k)`, and to associate them to the corresponding destination states. To reduce even further the memory requirements, the type of the action (`shift`, `reduce`, `accept` or `error` is also removed from the transitions, and a way to retrieve this information from the destination state is provided in the sequel. Thus, transitions can be completely identified by the destination states numbers, which give the associated labels and the type of action.

Taking into account that, generally, most states of an `LALR(1)` parser have more than one predecessor state, the space saved is substantial. The proposed encoding of LALR(1) matrices consists, basically, of a vector, here named `LALR`, where lists containing parsing actions — in fact, addresses of states — are stored. Each list corresponds to a parse state, and its first element always holds the state access symbol, i.e., the grammar symbol (or columns) that labels all the transitions reaching the state. The remainded elements in the list are addresses of successor states. In order to make the encoding process uniform, and to provide an easy mechanism to determine the type of the actions from the address of the state, entries containing `error` and `accept` are treated as transitions to the `distinguished` states `E` and `F`, and `reduce p` are viewed as transitions to `special` states. The `accept` action corresponds to a transition under the token $ to the `final` state `F`, whose list contains a single element, the $ symbol. All transitions for `error` actions always have the same destination state `E`. In almost all cases, `error` action is the most frequent entry in most LALR(1) rows. Thus, transitions to state `E` are encoded only once at the end of each list.

In fact, with the goal of keeping the parser algorithm simple, a transition to `E` will always be encoded at the end of every list that does not contain `reduce` actions. In this fashion, transitions to `E` will be selected only when no other is eligible.

The list of pairs associated to state `E` has always a single element, the access symbol, which must always contain the next input symbol `u`. The state `E` is called `error` state. A `special` state contains the transition label associated to a given `reduce` action, i.e., the lookahead symbol, and the number of the production involved. Each production `p` in the grammar corresponds to one or more `special` states, one for each entry of the form `reduce p`. The successor state of given state `s` is determined by the piece of code shown in Figure 3, in which `u` holds the next input symbol.

Note that, if `s` is equal to `E` at the ending of the loop statement of Figure 3, then a syntactic error was detected, otherwise `s` contains the address of the successor state. As discussed before, `error` actions are entirely removed from the rows containing `reduce` actions. In these states, the most frequent `reduce` action is selected only when no others are aplicable. The program fragment in Figure 3 assumes the presence of a flag state at the end of the list. Consequently, states having `reduce` actions must also be encoded according to the same pattern as that used in `shift` states. For this reason, it was created another `distinguished` state, named `R`, that works in a similar way to `E`, i.e., the state `R` is encoded at the end of all lists containing `reduce` actions, but the element that precedes

```
i := s + 1;   --- skip access symbol
s := LALR[i];
while u <> LALR[s] loop
    i := i + 1;
    s := LALR[i]
end
```

Figure 3: Finding the Successor State

R in the list must always represents the most frequent `reduce` action of the state. Thus, if, at the ending of the loop statement in Figure 3, `s` is equal to `R`, then the successor state is in fact the one that precedes `R` in the searched list. The state `R` is called `default reduction` state. The following statement

```
if s = R then s := LALR[i-1] end
```

must then be added at some point after the loop of Figure 3 to guarantee that `s` has the correct value. Since `R` works as a flag for the list searching, position `LALR[R]` must always contain the next input symbol. Lists do not contain pairs of the form `(column, reduce p)` anymore. These pairs were replaced by the addresses of `special` states, which themselves store the pairs.

Note that each pair `(column, reduce p)` needs only be stored once in the present method; its address may be used in more than one list. The type of the action, i.e., `shift`, `reduce`, `error` or `accept`, associated to each transition is implicitly determined by the parsing from the address of the destination state, as shown in the sequel. There are five types of states in this encoding: the `normal` states, which are those that correspond to rows of the original LALR(1) matrix, the `error` state (`E`), which is used as a flag to end lists containing only `shift` actions, the `final` state (`F`), the `default reduction` state (`R`), which is used as flag to end lists containing at least one `reduce` action, and the `special` states, which represent pairs of the form `(column, reduce p)`. These states are organized in the `LALR` vector in the way indicated in Figure 4.

In the data structure of Figure 4, transitions to states, say `k`, whose addresses are smaller than `E` represent actions of the form `shift k`; transitions to states whose addresses

	...			$			...	
←`normal` states		→	E	F	R	←`special` states		→

Figure 4: LALR Vector

are greater than `R` denote actions `reduce p`, where `p` is given by `LALR[k + 1]`. The `final` and `error` states have known addresses: `F` and `E`, respectively. Note that the `LALR` vector also incorporates the **GOTO** part of the LALR(1) table. This was achieved by allowing nonterminal symbols to be labels of transitions. In another words, in this method, the **ACTION** part of the LALR(1) table has been extended to encompass the **GOTO** part, whose entries for states numbers `k` are replaced by actions of the form `shift k`. In fact, the **GOTO** table was created only for efficiency purposes.

As a matter of fact, the automata from which the LALR(1) tables are constructed have transitions under terminal and nonterminal symbols. This new way of viewing LALR(1) matrices implies in a small change in the underlying parsing algorithm with respect to the semantic of `reduce` actions. Originally, an action of the form `reduce p`, where `p` is a production of the form `A` $\rightarrow \omega$, causes the removal of `m` (size of ω) elements from the stack of states followed by an access to the **GOTO** table, given the state currently on the top of the stack and the nonterminal `A`, to determine next state to be entered. In the proposed encoding scheme, this mechanism is equivalent to pop `m` elements from the stack, and to execute one transition under the nonterminal symbol `A`, as showed in Figure 5.

Note that it can be shown that error entries in **GOTO** table are never consulted, and that it is guaranteed that a transition under the nonterminal symbol `A` always exists for the state that appears in the top of the stack just after popping the stack. Consequently, the search always ends successfully.

Finally, it should be pointed out that as a consequence of the fusion of the **ACTION** and **GOTO** tables , the internal codes for terminal and nonterminal symbols must be disjoint. To conclude, in this encoding scheme, there exists another table, named `PROD`, that gives the internal code of left side nonterminal symbol and the size of the right hand side of each production. The new algorithm of the LALR(1) parser is presented in Figure 6, and Figure 7 illustrates the encoding of the table showed in Figure 1.

```
A    := PROD[p].LE;
top  := top - PROD[p].SIZE;
i    := SIN[top] + 1;
s    := LALR[i];
while A <> LALR[s] loop
    i := i + 1;
    s := LALR[i]
end;
top := top + 1;
SIN[top] := s;
```

Figure 5: Reduce Action

Analysis of the Method

The space efficiency of the proposed method for compactation of LALR(1) parsing tables depends on the following conditions:

1. The relation *number of transitions/number of states* should be reasonably greater than 1. The bigger this relation is, more space efficient the method is, since it encodes the label and the type of each transition only once.

2. The relation *number of repeated rows/total number of rows* should be less than `(a/m.c)`, where `a` is the space needed to store a pointer, `m` is the average number of entries per row, and `c` is the space occupied by an entry in the Aho and Ullman method. In languages like `PASCAL`, typical values are: `a = 2, m = 10` and `c = 3`. Thus, for these languages, the space occupied by pointers to the lists in the Aho and Ullman method is compensated by the elimination of repeated lists when the above relation is greater than `1/15`.

3. `error` is always the most frequent entry in each row.

In the tests performed, LALR(1) tables for which conditions (1) e (3) holds, were encoded in about 4% of its original space requirements. For instance, the compacted

```
 var parsing : boolean;
     u       : 0..imax;
     v       : value;
     LALR    : array[0..imax] of 0..imax;
     top     : 0..imax;
     SIN     : array[0..max] of 0..imax;
     s, i    : 0..imax;
     A       : 0..imax;
begin
    top := 0; s := initial_state; SIN[top] := s;
    parsing := TRUE;
    Lex(u,v);  LALR[E] := u;  LALR[R] := u;
    while parsing loop
        i := s + 1;    s:= LALR[i];
        while u <> LALR[s] loop
            i := i + 1;  s := LALR[i]
        end;
        if s >= R then                -- REDUCE p
             if  s = R then s := LALR[i - 1] end
             p := LALR[s + 1]; A := PROD[p].LE;
             top := top - PROD[p].SIZE;
             i := SIN[top] + 1;  s := LALR[i];
             while A <> LALR[s] loop
                 i := i + 1;  s := LALR[i]
             end;
             top := top + 1;  SIN[top] := s
        elif s < E then                -- SHIFT s --
             top := top + 1;  SIN[top] := s;
             Lex(u,v); LALR[E] := u;  LALR[R] := u;
        elif s = E then error recovery -- ERROR
        else parsing := FALSE          -- ACCEPT
        end
    end
end
```

Figure 6: The New LALR(1) Parser

parsing table for ADA requires only 8Kbytes of memory space to encode all the 800 LALR(1) parser states.

state 0							state 1				state 2				state 3		
	25	18	7	11	15	53	A	28	54	53	B	34	58	55	C	62	55
0							7				11				15		

state 4							state 5			state 6					
e	25	18	39	11	15	53	a	66	55	b	25	18	43	15	53
18							25			28					

state 7					state 8				state 9				state 10		
d	25	18	47	53	A	28	50	53	B	34	56	55	C	60	55
34					39				43				47		

state 11			E	F	R	r1		r2		r3		r4		r5		r6	
f	64	55		$		b	1	b	2	b	3	b	4	b	5	b	6
50			53	54	55	56		58		60		62		64		66	

Figure 7: An Example of LALR Vector

Conclusion

The compactation method described in this paper is based on two properties of LALR(1) parsers. The first one is the property of the viable prefix, and the second is that all transitions to a given state always have the same label. The efficacy of the method depends on some characteristics of the LALR(1) tables, which, in practice, hold for languages of interest. Its main source of memory economy is the technique of removing labels of transitions and types of actions from the state transitions, and encoding them together with the destination states.

This compactation scheme has been implemented and tested in a compiler generator system, which was developed at the Computer Science Departament of University Federal of Minas Gerais [7].

References

[1] AHO , A. V.,and ULLMAN, J. D., *The Theory of Parsing, Translation, and Compiling*, Vols. 1 e 2, Prentice-Hall, Englewood Cliffs, N.J. 1972.

[2] AHO A. V. and JOHNSON, *Principles of Compiler Design*, Addison-Wesley Publishing Company Co. (1977).

[3] AHO A. V., SETHI, R. and JOHNSON , *Principles of Compiler Design*, Addison-Wesley Publishing Company Co. (1986).

[4] KNUTH, D., *The Art of Computer Programming*, 2nd Edition, Addison-Wesley Publishing Company Co. (1973).

[5] SPECTOR, David, "*Full LR(1) Parser Generation*", ACM Sigplan Notices, Vol. 16, N' 8, August 1981.

[6] PAGER, David, "*A Practical General Method for Constructing LR(k) Parsers*", Acta Informatica 7, 249-268 (1977).

[7] Bigonha, Mariza A. S. and Bigonha, Roberto S., *SIC: A Tool for Implementation Languages*, (In portuguese), Proceeding of the XXI Congresso Nacional de Informática, SUCESU'88, pages. 426-431, 1988.