LR Parsers For Natural Languages[1]

Masaru Tomita
Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

MLR, an extended LR parser, is introduced, and its application to natural language parsing is discussed. An LR parser is a shift-reduce parser which is deterministically guided by a parsing table. A parsing table can be obtained automatically from a context-free phrase structure grammar. LR parsers cannot manage ambiguous grammars such as natural language grammars, because their parsing tables would have multiply-defined entries, which precludes deterministic parsing. MLR, however, can handle multiply-defined entries, using a dynamic programming method. When an input sentence is ambiguous, the MLR parser produces all possible parse trees without parsing any part of the input sentence more than once in the same way, despite the fact that the parser does not maintain a chart as in chart parsing. Our method also provides an elegant solution to the problem of multi-part-of-speech words such as "that". The MLR parser and its parsing table generator have been implemented at Carnegie-Mellon University.

1 Introduction

LR parsers [1, 2] have been developed originally for programming language of compilers. An LR parser is a shift-reduce parser which is deterministically guided by a parsing table indicating what action should be taken next. The parsing table can be obtained automatically from a context-free phrase structure grammar, using an algorithm first developed by DeRemer [5, 6]. We do not describe the algorithm here, reffering the reader to Chapter 6 in Aho and Ullman [4]. The LR parsers have seldom been used for Natural Language Processing probably because:

1. It has been thought that natural languages are not context-free, whereas LR parsers can deal only with context-free languages.

2. Natural languages are ambiguous, while standard LR parsers can not handle ambiguous languages.

The recent literature [8] shows that the belief "natural languages are not context-free" is not necessarily true, and there is no reason for us to give up the context-freedom of natural languages. We do not discuss on this matter further, considering the fact that even if natural languages are not context-free, a fairly comprehensive grammar for a subset of natural language sufficient for practical systems can be written in context-free phrase structure. Thus, our main concern is how to cope with the ambiguity of natural languages, and this concern is addressed in the following section.

2 LR parsers and Ambiguous Grammars

If a given grammar is ambiguous,[2] we cannot have a parsing table in which every entry is uniquely defined; at least one entry of its parsing table is multiply defined. It has been thought that, for LR parsers, multiple entries are fatal because they make deterministic parsing no longer possible.

Aho *et. al.* [3] and Shieber [12] coped with this ambiguity problem by statically[3] selecting one desired action out of multiple actions, and thus converting multiply-defined entries into uniquely-defined ones. With this approach, every input sentence has no more than one parse tree. This fact is desirable for programming languages.

For natural languages, however, it is sometimes necessary for a parser to produce more than one parse tree. For example, consider the following short story.

I saw the man with a telescope.
He should have bought it at the department store.

When the first sentence is read, there is absolutely no way to resolve the ambiguity[4] at that time. The only action the system can take is to produce two parse trees and store them somewhere for later disambiguation.

In contrast with Aho *et. al.* and Shieber, our approach is to extend LR parsers so that they can handle multiple entries and produce more than one parse tree if needed. We call the extended LR parsers MLR parsers.

[1] This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

[2] A grammar is ambiguous, if some input sentence can be parsed in more than one way.

[3] By "statically", we mean the selection is done at parsing table construction time.

[4] "I" have the telescope, or "the man" has the telescope.

3 MLR Parsers

An example grammar and its MLR parsing table produced by the construction algorithm are shown in fig. 1 and 2, respectively. The MLR parsing table construction algorithm is exactly the same as the algorithm for LR parsers. Only the difference is that an MLR parsing table may have multiple entries. Grammar symbols starting with "*" represent pre-terminals. "sh *n*" in the action table (the left part of the table) indicates the action "shift one word from input buffer onto the stack, and go to state *n*". "re *n*" indicates the action "reduce constituents on the stack using rule *n*". "acc" stands for the action "accept", and blank spaces represent "error". Goto table (the right part of the table) decides to what state the parser should go after a reduce action. The exact definition and operation of LR parsers can be found in Aho and Ullman [4].

We can see that there are two multiple entries in the table; on the rows of state 11 and 12 at the column of "*prep". As mentioned above, once a parsing table has multiple entries, deterministic parsing is no longer possible; some kind of non-determinism is necessary. We shall see that our dynamic programming approach, which is described below, is much more efficient than conventional breath-first or depth-first search, and makes MLR parsing feasible.

Our approach is basically pseudo-parallelism (breath-first search). When a process encounters a multiple entry with n different actions, the process is split into n processes, and they are executed individually and parallelly. Each process is continued until either an "error" or an "accept" action is found. The processes are, however, synchronized in the following way: When a process "shifts" a word, it waits until all other processes "shift" the word. Intuitively, all processes always look at the same word. After all processes shift a word, the system may find that two or more processes are in the same state; that is, some processes have a common state number on the top of their stacks. These processes would do the exactly same thing until that common state number is popped from their stacks by some "reduce" action. In our parser, this common part is processed only once. As soon as two or more processes in a common state are found, they are combined into one process. This combining mechanism guarantees that any part of an input sentence is parsed no more than once in the same manner. This makes the parsing much more efficient than simple breath-first or depth-first search. Our method has the same effect in terms of parsing efficiency that posting and recognizing common subconstituents of different parses have in the chart parsing method [10, 11]. The idea should be made clear by the following example.

4 An Example

In this section, we demonstrate, step by step, how our MLR parser processes the sentence:

I SAW A MAN WITH A TELESCOPE

using the grammar and the parsing table shown in fig 1 and 2. This sentence is ambiguous, and the parser should accept the sentence in two ways.

Until the system finds a multiple entry, it behaves in the exact same manner as a conventional LR parser, as shown in fig 3-a below. The number on the top (rightmost) of the stack indicates the current state. Initially, the current state is 0. Since the parser is looking at the word "I", whose category is "*n", the next action "shift and goto state 4" is determined from the parsing table. The parser takes the word "I" away from the input buffer, and pushes the preterminal "*n" onto the stack. The next word the parser is looking at is "SAW", whose category is "*v", and "reduce using rule 3" is determined as the next action. After reducing, the parser determines the current state, 2, by looking at the intersection of the row of state 0 and the column of "NP", and so on.

STACK	NEXT-ACTION	NEXT-WORD
0	sh 4	I
0 *n 4	re 3	SAW
0 NP 2	sh 7	SAW
0 NP 2 *v 7	sh 3	A
0 NP 2 *v 7 *det 3	sh 10	MAN
0 NP 2 *v 7 *det 3 *n 10	re 4	WITH
0 NP 2 *v 7 NP 12	re 7, sh 6	WITH

Fig 3-a

At this point, the system finds a multiple entry with two different actions, "reduce 7" and "shift 6". Both actions are processed in parallel, as shown in fig 3-b.

```
(1)   S --> NP VP
(2)   S --> S PP
(3)  NP --> *n
(4)  NP --> *det *n
(5)  NP --> NP PP
(6)  PP --> *prep NP
(7)  VP --> *v NP
```

Fig 1

State	*det	*n	*v	*prep	$	NP	PP	VP	S
0	sh3	sh4				2			1
1				sh6	acc		5		
2			sh7	sh6			9	8	
3		sh10							
4			re3	re3	re3				
5				re2	re2				
6	sh3	sh4				11			
7	sh3	sh4				12			
8				re1	re1				
9			re5	re5	re5				
10			re4	re4	re4				
11			re6	re6,sh6	re6		9		
12				re7,sh6	re7		9		

Fig 2

```
-----------------------------------------------------------
0 NP 2 VP 8                              re 1        WITH
0 NP 2 *v 7 NP 12 *prep 6                wait           A

0 S 1                                    sh 6        WITH
0 NP 2 *v 7 NP 12 *prep 6                wait           A

0 S 1 *prep 6                            sh 3           A
0 NP 2 *v 7 NP 12 *prep 6                sh 3           A
-----------------------------------------------------------
```

Fig 3-b

Here, the system finds that both processes have the common state number, 6, on the top of their stacks. It combines two processes into one, and operates as if there is only one process, as shown in fig 3-c.

```
-----------------------------------------------------------
0 S 1 ───────────>*prep 6                sh 3           A
0 NP 2 *v 7 NP 12/

0 S 1 ───────────>*prep 6 *det 3         sh 10  TELESCOPE
0 NP 2 *v 7 NP 12/

0 S 1 ───────────>*prep 6 *det 3 *n 10   re 4           $
0 NP 2 *v 7 NP 12/

0 S 1 ───────────>*prep 6 NP 11          re 6           $
0 NP 2 *v 7 NP 12/
-----------------------------------------------------------
```

Fig 3-c

The action "reduce 6" pops the common state number 6, and the system can no longer operate the two processes as one. The two processes are, again, operated in parallel, as shown in fig 3-d.

```
-----------------------------------------------------------
0 S 1 PP 5                               re 2           $
0 NP 2 *v 7 NP 12 PP 9                   re 5           $

0 S 1                                    accept
0 NP 2 *v 7 NP 12                        re 7           $
-----------------------------------------------------------
```

Fig 3-d

Now, one of the two processes is finished by the action "accept". The other process is still continued, as shown in fig 3-e.

```
-----------------------------------------------------------
0 NP 2 VP 8                              re 1           $
0 S 1                                    accept
-----------------------------------------------------------
```

Fig 3-e

This process is also finished by the action "accept". The system has accepted the input sentence in both ways. It is important to note that any part of the input sentence, including the prepositional phrase "WITH A TELESCOPE", is parsed only once in the same way, without maintaining a chart.

5 Another Example

Some English words belong to more than one grammatical category. When such a word is encountered, the MLR parsing table can immediately tell which of its categories are legal and which are not. When more than one of its categories are legal, the parser behaves as if a multiple entry were encountered. The idea should be made clear by the following example.

Consider the word "that" in the sentence:

That information is important is doubtful.

A sample grammar and its parsing table are shown in Fig. 4 and 5, respectively. Initially, the parser is at state 0. The first word "that" can be either "*det" or "*that", and the parsing table tells us that both categories are legal. Thus, the parser processes "sh 5" and "sh 3" in parallel, as shown below.

```
 STACK                          NEXT ACTION  NEXT WORD
-------------------------------------------------------
0                               sh 5, sh 3     That

0                                 sh 5         That
0                                 sh 3         That

0 *det 5                          sh 9       information
0 *that 3                         sh 4       information

0 *det 5 *n 9                     re 2          is
0 *that 3 *n 4                    re 3          is

0 NP 2                            sh 6          is
0 *that 3 NP 2                    sh 6          is
-------------------------------------------------------
```

Fig. 6-a

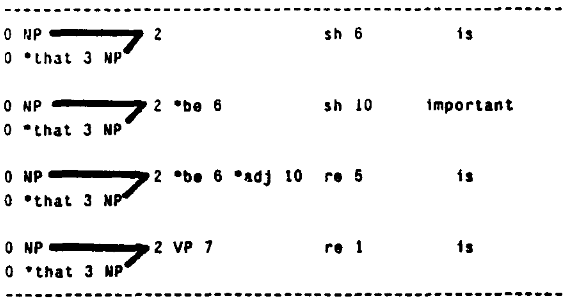
At this point, the parser founds that both processes are in the same state, namely state 2, and they are combined as one process.

```
------------------------
(1)  S  --> NP VP
(2)  NP --> *det *n
(3)  NP --> *n
(4)  NP --> *that S
(5)  VP --> *be *adj
------------------------
```

Fig. 4

State	*adj	*be	*det	*n	*that	$	NP	S	VP
0			sh5	sh4	sh3		2	1	
1						acc			
2		sh6							7
3			sh5	sh4	sh3		2	8	
4		re3							
5				sh9					
6	sh10								
7		re1				re1			
8		re4							
9		re2							
10		re5				re5			

Fig. 5



```
------------------------------------------------------
0 NP ———————→ 2                       sh 6        is
0 *that 3 NP

0 NP ———————→ 2 *be 6                 sh 10     important
0 *that 3 NP

0 NP ———————→ 2 *be 6 *adj 10         re 5        is
0 *that 3 NP

0 NP ———————→ 2 VP 7                  re 1        is
0 *that 3 NP
------------------------------------------------------
```

Fig. 6-b

The process is split into two processes again.

```
------------------------------------------------------
0 NP 2 VP 7                           re 1        is
0 *that 3 NP 2 VP 7                   re 1        is

0 S 1                                #ERROR#      is
0 *that 3 S 8                         re 4        is
------------------------------------------------------
```

Fig. 6-c .

One of two processes detects "error" and halts; only the other process goes on.

```
------------------------------------------------------
0 NP 2                                sh 6        is
0 NP 2 *be 6                          sh 10     doubtful
0 NP 2 *be 6 *adj 10                  re 5        $
0 NP 2 VP 7                           re 1        $
0 S 1                                 acc         $
------------------------------------------------------
```

Fig. 6-d

Finally, the sentence has been parsed in only one way. We emphasize again that, in spite of pseudo-parallelism, each part of the sentence was parsed only once in the same way.

6 Concluding Remarks

The MLR parser and its parsing table generator have been implemented at Computer Science Department, Carnegie-Mellon University. The system is written in MACLISP and running on Tops-20.

One good feature of an MLR parser (and of an LR parser) is that, even if the parser is to run on a small computer, the construction of the parsing table can be done on more powerful, larger computers. Once a parsing table is constructed, the execution time for parsing depends weakly on the number of productions or symbols in a grammar. Also, in spite of pseudo-parallelism, our MLR parsing is theoretically still deterministic. This is because the number of processes in our pseudo-parallelism never exceeds the number of states in the parsing table.

One concern of our parser is whether the size of a parsing table remains tractable as the size of a grammar grows. Fig. 6 shows the relationship between the complexity of a grammar and its LR parsing table (excerpt from Inoue [9]).

	XPL	EULER	FORTRAN	ALGOL60
Terminals	47	74	63	66
Non-terminals	51	45	77	99
Productions	108	121	172	205
States	180	193	322	337
TableSize(byte)	2041	2587	3662	4264

Fig. 6

Although the example grammars above are for programming langauges, it seems that the size of a parsing table grows only in proportion to the size of its grammar and does not grow rapidly. Therefore, there is a hope that our MLR parsers can manage grammars with thousands of phrase structure rules, which would be generated by rule-schema and meta-rules for natural language in systems such as GPSG [7].

Acknowledgements

I would like to thank Takehiro Tokuda, Osamu Watanabe, Jaime Carbonell and Herb Simon for thoughtful comments on an earlier version of this paper.

References

[1] Aho, A. V. and Ullman, J. D.
The Theory of Parsing, Translation and Compiling.
Prentice-Hall, Englewood Cliffs, N. J., 1972.

[2] Aho, A. V. and Johnson, S. C.
LR parsing.
Computing Surveys 6:2:99-124, 1974.

[3] Aho, A. V., Johnson, S. C. and Ullman, J. D.
Deterministic parsing of ambiguous grammars.
Comm. ACM 18:8:441-452, 1975.

[4] Aho, A. V. and Ullman, J. D.
Principles of Compiler Design.
Addison Wesley, 1977.

[5] Deremer, F. L.
Practical Translators for LR(k) Languages.
PhD thesis, MIT, 1969.

[6] DeRemer, F. L.
Simple LR(k) grammars.
Comm. ACM 14:7:453-460, 1971.

[7] Gazdar, G.
Phrase Structure Grammar.
D. Reidel, 1982, pages 131-186.

[8] Gazdar, G.
Phrase Structure Grammars and Natural Language.
Proceedings of the Eighth International Joint Conference on Artificial Intelligence v.1, August, 1983.

[9] Inoue, K. and Fujiwara, F.
On LLC(k) Parsing Method of LR(k) Grammars.
Journal of Information Processing vol.6(no.4):pp.206-217, 1983.

[10] Kaplan, R. M.
A general syntactic processor.
Algorithmics Press, New York, 1973, pages 193-241.

[11] Kay, M.
The MIND system.
Algorithmics Press, New York, 1973, pages 155-188.

[12] Shieber, S. M.
Sentence Disambiguation by a Shift-Reduce Parsing Technique.
Proceedings of the Eighth International Joint Conference on Artificial Intelligence v.2, August, 1983.