On Single Production Elimination in Simple LR(k) Environment

Akifumi Makinouchi*

A method of syntax description for programming languages is presented. It allows to add context sensitive conditions to the conventional BNF so that such parts of programming language syntax as arithmetic expressions can be notated without single productions. This results in speed-up of grammar parsing as well as flexibility in description of grammars. The method of construction of LR parsers for such grammars is also shown.

1. Introduction

Since Knuth [1] proposed LR(k) grammars, much effort has been made in developing LR parsing methods. This is because LR(k) grammars constitute the largest natural class of unambiguous grammars for which we can construct deterministic parsers. Moreover, it is possible to produce LR parsers that are competitive with other types of parsers if adequate optimizations are applied (see [2]).

Optimizations have mainly taken the course of parser table size reduction [3, 4, 5 and 6], possibly with restricting LR(k) grammars. Among them, DeRemer's Simple LR(k) grammars (and their direct extensions LALR(k) grammars) seem to have been accepted widely and practical parser generators based on his method have been implemented both in academic fields and in industry [7 and 8]. While considerable effort has been made to optimize table size, very little attention has ever been paid to speeding up Simple LR parsers, which is another area of optimization [2 and 9].

This paper deals with a simple technique for speeding up Simple LR parsers via single production elimination. This problem is not negligible as the LR parser is applied to programming languages such as FORTRAN, ALGOL, etc.; and programs written in any of these languages contain arithmetic (and/or logical) expressions whose parsing time accounts for a considerable part of the total parsing time [10 and 11].

Our method starts with a basic idea as follows:

A user writes a grammar (a set of production rules) which has no single productions but which is not ambiguous with the aid of (left and/or right) context conditions.

Although left contexts may be specified in a grammar, the parser for the grammar does not have to look back at the current left context. The parser remains a Simple LR parser.

The paper [12] presented a similar idea in the sense that an ambiguity grammar is parsed by an LR parser whose action table conflicts are resolved by means of disambiguity rules such as precedence and associativity. These disambiguity rules, however, are not integrated in the production rules; and it seems hard and awkward, if not impossible, to do so since precedence and associativity are notions of different levels from syntax notation in terms of nonterminals and terminals.

2. Operator Grammar with Context Condition

We base our idea on the operator grammar as defined by Floyd [13]. Hereafter, we suppose that the reader is familiar with the notions and terminology of context-free grammar and will discuss our idea rather informally.

Let G be a context-free grammar. If no production of the grammar G takes the form $A \rightarrow \alpha BC\beta$ where A, B, and C are nonterminals and α and β are strings of grammatical symbols (including empty string), the G is said to be an operator grammar.

Example 1. G_1 which is given below for arithmetic expressions with operators $+$ and $*$ is an operator grammar.

$$G_1: E \rightarrow E+E \quad (1)$$

$$|E * E \quad (2)$$

$$|(E) \quad (3)$$

$$|a \quad (4)$$

Now we introduce a context sensitive feature into operator grammars.

Let C^k be a set of, at most, k-long strings of terminals in G. C^k is suffixed with l or r, indicating left or right, respectively. We call C_l^k (C_r^k) left condition (right condition).

AG is said to be an augmented operator grammars with negative contexts (hereafter, simply called augmented grammars) if productions of AG have context conditions on their left-hand side as follows:

$$\neg C_l^1 A \neg C_r^k \rightarrow \alpha \quad \text{where } A \rightarrow \alpha \text{ is a production of } G.$$

*Department of Computer Science, Fujitsu Laboratories Ltd., 1015 Kamiodanaka Nakahara-ku, Kawasaki, Japan.

Example 2. The following is an augmented grammar for G_1. {and} are used for set notation, and each element of a set is separated by a comma (,)

$$AG_1\colon \neg\{+,*\}E\neg\{*\}\rightarrow E+E \qquad (1)$$

$$\neg\{*\}E\rightarrow E*E \qquad (2)$$

$$E\rightarrow(E) \qquad (3)$$

$$E\rightarrow a \qquad (4)$$

Let $\neg C_l^1 A\neg C_r^k\rightarrow\alpha$ be a production of an augmented grammar AG and the start symbol of AG be S. If βaAy is a right sentential form of AG, where a is a terminal and y is a string of terminals, then the given production rule can be applied to A in the sentential form and we have a sequence of sentential forms

$$S\underset{rm}{\overset{*}{\Rightarrow}}\beta aAy\underset{rm}{\Rightarrow}\beta a\alpha y$$

where *rm* represents rightmost derivation, iff $a\notin C_l^1$ and any first n-long substring of $y(1\leqq n\leqq k)\notin C_r^k$.

Example 3. Using AG_1 given in Example 2, we have a sequence of right sentential forms as below. Note that underlined nonterminals indicate the target of derivation in each step.

$$\underline{E}\underset{rm}{\Rightarrow}E*\underline{E}\underset{rm}{\Rightarrow}\underline{E}*a\underset{rm}{\Rightarrow}E*\underline{E}*a$$

$$\underset{rm}{\Rightarrow}\underline{E}*a*a\underset{rm}{\Rightarrow}a*a*a$$

The following sequence cannot occur.

$$\underline{E}\underset{rm}{\Rightarrow}E*\underline{E}\underset{rm}{\Rightarrow}E*E+E$$

This is because production rule (1) of AG is not applicable to the rightmost E in the second sentential form.

The reader may easily confirm that AG_1 is equivalent to G_2 in Example 4 in terms of usual arithmetic expression parsing. (See Appendix A for proof.)

Example 4.

$$G_2\colon E\rightarrow E+T \qquad (1)$$

$$|T \qquad (2)$$

$$T\rightarrow T*F \qquad (3)$$

$$|F \qquad (4)$$

$$F\rightarrow(E) \qquad (5)$$

$$|a \qquad (6)$$

3. Simple LR(1) Parser Construction for the Augmented Grammar

For the sake of simplicity as well as practicality, we confine ourselves to Simple LR(1) parser construction for augmented grammars although an extension to Simple LR(k) or LALR(k) is straightforward. An LR parser needs a stack in which is stored a sequence of grammar symbols representing a portion of the right sentential form along with special symbols called states and a parsing table which controls the next move of the parser with a state given on top of the stack and a current input symbol. The parsing table consists of two parts; namely, an action table and a goto table. The reader is invited to see Appendix B for better understanding of moves of the parser, roles of the table, etc.

The problem, then, of the LR parser construction for a given grammar is to fill up a parsing table. Definitions used in this section for Simple LR(1) parser table construction follow those in references [12] and [2] with necessary modifications.

Let AG be an augmented grammar for which we want to fabricate a Simple LR(1) parser. It is supposed that the first production rule has the form $S'\rightarrow\vdash S\dashv$ where S' is a new start symbol, S is an old one and $\vdash$and$\dashv$ are left and right end markers of the input string, respectively.

If A is a nonterminal and $\neg C_l^1 A\neg C_r^1\rightarrow\alpha$ is a production of AG, FOLLOW($\neg C_l^1 A\neg C_r^1$) is defined to be a set of terminals which can appear immediately to the right of A in a right sentential form to which the production rule is applicable.

PRECEDE($\neg C_l^1 A\neg C_r^1$) is defined under the same condition as the FOLLOW above to be the set of terminals which can appear immediately to the left of A in a right sentential form to which the production rule is applicable.

PRECEDE($\alpha.B\beta$) where $\neg C_l^1 A\neg C_r^1\rightarrow\alpha B\beta$ is a production of AG is a set of terminals which can precede B in a right sentential form and is defined as follows:
PRECEDE($\alpha.B\beta$)$=$*if* α is empty, *then* PRECEDE ($\neg C_l^1 A\neg C_r^1$) *else* the rightmost symbol of α. (See note 1)

Note that C_l^1 and C_r^1 may be empty sets; so these definitions are appropriate to any nonterminals which are assigned no context conditions.

$\neg C_l^1 A\neg C_r^1\rightarrow\alpha.\beta$ is said to be an item. We define CLOSURE(I) to be the smallest set of items where I is a set of items and the following conditions are satisfied:

(1) I is contained in CLOSURE(I).
(2) If the item $\neg C_l^1 A\neg C_r^1\rightarrow\alpha.B\beta$ is in CLOSURE(I) and any terminal in $C_l'^1$ of the production $\neg C_l'^1 B\neg C_r'^1\rightarrow\gamma$ is not in PRECEDE($\alpha.B\beta$) then the item $\neg C_l'^1 B\neg C_r'^1\rightarrow.\gamma$ is in CLOSURE(I).

GOTO(I, X)$=J$, where I is a set of items, X a grammar symbol and J is a closure of the set of items

$$\{\neg C_l^1 A\neg C_r^1\rightarrow\alpha X.\beta\,|\,\neg C_l^1 A\neg C_r^1\rightarrow\alpha.X\beta \text{ is in } I\}.$$

In order to construct a parsing table for AG, we compute a collection of sets of items STATE beginning with

$$\text{STATE}=\{I_0=\text{CLOSURE }(\{S'\rightarrow\vdash.S\dashv\})\}.$$

Then for each set of items I in STATE and for each grammar symbol X, GOTO(I, X) is computed; and if the resulting set of items is not already in STATE, it is added there. Computation is terminated when no more sets of items can be added.

Note 1: The fact that this symbol is a terminal is guaranteed by the condition that AG is an operator grammar.

Example 5. The collection of sets of items for AG is given below. Although context conditions are omitted, the reader can easily supply them by tracing the computation step by step.

I_0: $E' \to \vdash .E \dashv$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .a$

$I_1 = \mathrm{GOTO}\ (I_0, E)$
: $E' \to \vdash E. \dashv$
$E \to E.+E$
$E \to E.*E$

$I_2 = \mathrm{GOTO}\ (I_0, ()$
: $E \to (.E)$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .a$

$I_3 = \mathrm{GOTO}\ (I_0, a)$
: $E \to a.$

$I_4 = \mathrm{GOTO}\ (I_1, +)$
$E \to E+.E$
$E \to .E*E$
$E \to .(E)$
$E \to .a$

$I_5 = \mathrm{GOTO}\ (I_1, *)$
: $E \to E*.E$
$E \to .(E)$
$E \to .a$

$I_6 = \mathrm{GOTO}\ (I_2, E)$
: $E \to (E.)$
$E \to E.+E$
$E \to E.*E$

$I_7 = \mathrm{GOTO}\ (I_4, E)$
: $E \to E+E.$
$E \to E.*E$

$I_8 = \mathrm{GOTO}(I_5, E)$
: $E \to E*E.$

$I_9 = \mathrm{GOTO}\ (I_6,))$
: $E \to (E).$

Once STATE for AG is established, it is possible to construct action and goto tables of the LR parser for AG following the rules given below:

Let the set of items I_i in STATE be named state i.

1. If I_i contains an item of the form $\neg C_l^1 A \neg C_r^1 \to \alpha.a\beta$ where a is a terminal, then the action of state i on input a is "shift j" where j is the state associated with the set of items $\mathrm{GOTO}(I_i, a)$.
2. If I_i contains the item $S' \to \vdash S.\dashv$, then the action of state i on the right end marker $\dashv$ is "accept".
3. If I_i contains the item $\neg C_l^1 A \neg C_r^1 \to \alpha.$ and a is in $\mathrm{FOLLOW}(\neg C_l^1 A \neg C_r^1)$, then the action of state i on input a is "reduction by production $\neg C_l^1 A \neg C_r^1 \to \alpha$".
4. Otherwise, the action of state i on input a is "error".
5. The goto table entry for state i on nonterminal A is the state j where $I_j = \mathrm{GOTO}(I_i, A)$.

When one, and only one, action is defined for each entry of the parsing table, the parser is said to be a Simple LR(1) parser for the given augmented grammar.

Note that this construction method can be applied to any Simple LR(1) grammar which is not necessarily an operator grammar. This is so because C_l^1 and C_r^1 are allowed to be null.

Example 6. Table 1 in Appendix B is the parsing table for AG_1 defined in Example 2.

4. Specification Capability of Augmented Operator Grammars

Examples given here show how powerful and simple it is to specify a syntax of a language by means of augmented operator grammars. All of the examples are cited from the reference [12]. What we are interested in here is a class of augmented grammars, each of whose parsers is not only a Simple LR(1) parser but also more economical (in terms of parsing table size and parsing time) than the parser for the counterpart Simple LR(1) grammars.

Example 7. The following is another version of G. The expression is evaluated right to left as in APL.

$$
\begin{array}{lllr}
AG_2: & E\neg\{+,*\} & \to E+E & (1)\\
 & E\neg\{+,*\} & \to E*E & (2)\\
 & E & \to (E) & (3)\\
 & E & \to a & (4)
\end{array}
$$

Example 8 may help the reader to conjecture a rule for defining a syntax of more complex expressions which involve several operators.

Example 8. The lowest precedence is given to $<$; and progressively higher precedences are assigned to operators $+$, $*$, $\uparrow$ in this order. $+$ and $-$ have the same precedence and so do $*$ and $/$. $<$ has no associativity, $+$, $-$, $*$ and $/$ are left associative and $\uparrow$ is right associative as usual.

$$
\begin{array}{llr}
AG_3: & \neg\{<,+,-,*,/,\uparrow\}E\neg\{<,+,-,*\ /,\uparrow\} \to E<E & (1)\\
 & \neg\{+,-,*,/,\uparrow\}E\neg\{*,/,\uparrow\} \to E+E & (2)\\
 & \neg\{+,-,*,/,\uparrow\}E\neg\{*,/,\uparrow\} \to E-E & (3)\\
 & \neg\{*,/,\uparrow\}E\neg\{\uparrow\} \to E*E & (4)\\
 & \neg\{*,/,\uparrow\}E\neg\{\uparrow\} \to E/E & (5)\\
 & E\neg\{\uparrow\} \to E\uparrow E & (6)\\
 & E \to (E) & (7)\\
 & E \to id & (8)
\end{array}
$$

The same idea may be applied to PL/I like IF statement.

Example 9. *else* is to be associated with the closest unmatched then.

AG_4: $S\neg\{else\} \to$ *if* boolean-expression *then S*
| *if* boolean-expression *then S else S*
$s \to$ other-statement

5. Implementation Consideration

So far, we have explained our idea on the basis of operator grammars. Although operator grammars seem flexible enough to specify a variety of programming languages' syntaxes, it may be nice if almost all the syntax of a desired language can be written by means of non-restricted context-free grammar notation and if the augmented grammar notation is incorporated in small parts of the syntax where the latter notation benefits. An arithmetic (and/or logical) expression is a good example of such parts. This is possible if a nonterminal with a left context condition specified in the left hand side of some production rule in a given grammar is always preceded by terminals in the right hand side of any production of the grammar. We can say this because left contexts to be checked against specified left context conditions on the rightmost derivation of the grammar need to be fixed only when a collection of sets of its items is calculated.

The right context condition of a production, if any, is merely concerned with selection of one of the actions which, without the context condition, would conflict in a parsing table entry associated with the state involving an item with a dot at the end of the production and indicates that reduction by the production should be excluded.

Therefore, implementation of the context condition feature in the environment of the Simple LR parser is very simple. In fact, we are implementing a parser generator whose features are like the following:

1. Generates an LALR(k) parser for a given grammar, where $k=0$, 1, 2 or 3.
2. Accepts a wide variety of extended BNFs, especially with great freedom in putting semantic actions in a production rule.
3. In addition to 2, the generator accepts the context condition notation.
4. Provides for a given-grammar dependent, but not a specific-language dependent, error recovery routine.

We have completed the implementation and plan to have various experiments.

6. Conclusion

It has been shown so far that with restricted but very simple context conditions a grammar (which, without them, would be ambiguous) can be parsed in an LR parser.

This seems to be a considerable improvement in favor of LR parser application to real programming language parsing.

The works [9, 14] presented the idea of using ambiguous context-free grammars with disambiguity rules involving notions such as precedence and associativity and showed that LR parsers for them can be constructed with the aid of disambiguity rules to resolve LR parsing table conflicts.

The work [15], on the other hand, discussed theoretically the possibility of conversion of a context-free grammar to a context-sensitive grammar whose production rules may be associated with $a(m, n)$ context so as to reduce the number of nonterminals and showed some interesting theoretical results. It must be mentioned, however, that the (m, n) contexts are affirmative; while the context conditions with which an operator grammar is augmented are negative in the sense described earlier in this paper. Moreover, the (m, n) contexts seem to be checked during parsing although the work did not mention any particular parsing method.

We showed in this paper that certain kinds of unambiguous (context-sensitive) grammars with negative $(1, 1)^{\text{note }2}$ context conditions may be parsed by LR parsers (especially by simple LR parsers) and that the context conditions are directly related to the construction of an LR parsing table without conflicts. Hence, our idea may be said to be an amalgam of the previous ones mentioned above.

Appendix A:

To show that AG_1 is equivalent to G_2 (in the sense that any expression generated by AG_1 can be generated by G_2 with the same production tree except for single productions and vice versa), we make a transformation between the two grammars which preserves the production tree structure, except for single productions. The transformation is done step by step; and it is easily seen that each step transformation preserves the production tree structure, except for single productions.

AG_1 is transformed into G_2

Because productions (1) and (2) of AG_1 cannot be applied to E (which is at the right side of $*$ in production (2)), we can replace the E by F if production $E \to F$ is introduced and every left-hand-side E of productions (3) and (4) is replaced by F in order to preserve the applicability of productions (3) and (4) to E of productions (1) and (2). Thus, we have the following grammar omitting $*$ from the context condition in question.

$$\neg\{+\}E\neg\{*\} \to E+E \qquad (1\text{–}1)$$

$$E \to E*F \qquad (1\text{–}2)$$

$$E \to F \qquad (1\text{–}3)$$

$$F \to (E) \qquad (1\text{–}4)$$

$$F \to a \qquad (1\text{–}5)$$

Because production (1–1) is inapplicable to E at the right side of $+$ in (1–1) as well as to E at the left side of $*$ in (1–2), replace these E's and E's in the left hand side of (1–2) and (1–3) by T and introduce production $E \to T$ which will preserve the applicability of (1–2) and (1–3) to the E which has not been replaced in the right-hand side of (1–1). Then, by omitting the conditions which are no longer necessary, we have G_2 as follows:

$$E \to E+T \qquad (2\text{–}1)$$

$$E \to T \qquad (2\text{–}2)$$

Note 2. Note that the extension to (m, n) context is a direct thing to do, although it is not necessary from the practical viewpoint.

$$T \to T * F \qquad (2\text{–}3)$$

$$T \to F \qquad (2\text{–}4)$$

$$F \to (E) \qquad (2\text{–}5)$$

$$F \to a \qquad (2\text{–}6)$$

Transformation of G_2 to AG_1

The transformation is done by reversing the transformation of AG_1 to G_2. Firstly, replace T by F and, secondly, F by E. The applicability of productions being considered, appropriate context conditions are given to the left hand side of the productions.

Appendix B:

LR parser consists of an imput tape, a stack and a parsing table. The parsing table is represented by a two-dimensional matrix where a row represents a state I_i and a column either a lookahead terminal symbol string u for an action part or a grammar symbol X for goto part. An entry is designated by $f(I_i, u)$ for an action part or goto (I_i, X) for a goto part. See Table 1 for grammar AG_1. This is a Simple LR(1) parsing table.

As in [2], an LR parser's move is expressed by a sequence of LR parser configurations which are a triple $(I_0X_1I_1 \cdots X_mI_m, \omega, \pi)$ where I_i is a state, X_i is a grammar symbol, ω is a string of terminals, and π is an output string of production numbers. $I_0X_1I_1 \cdots X_mI_m$ is in the stack whose top contains I_m, and ω is on the input tape.

Let the parser be in configuration $(I_0X_1I_1 \cdots X_mI_m, \omega, \pi)$.

(1) If $f(I_m, u) =$ shift I and $\omega = a\omega'$ then the next configuration becomes $(I_0X_1I_1 \cdots X_mI_maI, \omega', \pi)$.

(2) If $f(I_m, u) =$ reduce i and the i-th production is $A \to \alpha$ and the length of $\alpha = l$, then the next configuration is $(I_0X_1I_1 \cdots X_{m-l}I_{m-l}AI, \omega, \pi)$ where I_{m-l} is the state on the top when a string of length $2l$ is removed from the top of the stack and $I =$ goto (I_{m-l}, A).

(3) Otherwise, there is no next configuration.

The following is the move of the LR parser for AG_1 with input $a+a$.

$$\begin{aligned}(0, a+a\dashv, e) &\vdash (0a3, +a\dashv, e)\\ &\vdash (0E1, +a\dashv, 4)\\ &\vdash (0E1+4, a\dashv, 4)\\ &\vdash (0E1+4a3, \dashv, 4)\\ &\vdash (0E1+4E7, \dashv, 44)\\ &\vdash (0E1, \dashv, 441)\\ &\vdash \text{accept}\end{aligned}$$

This sequence of configurations is shorter by 3 than the corresponding one for G_2.

Table 1 Simple LR(1) table for AG.

si, *ri* and *a* are abbreviations of shift *i*, reduction *i* and accept, respectively. This table is smaller by 2 rows and 2 columns than the corresponding table for G_2.

State	Action						Goto
	+	*	(	)	*a*	⊣	*E*
0			*S*2		*S*3		1
1	*S*4	*S*5				*a*	
2			*S*2		*S*3		6
3	*r*4	*r*4		*r*4		*r*4	
4			*S*2		*S*3		7
5			*S*2		*S*3		8
6	*S*4	*S*5		*S*9			
7	*r*1	*S*5		*r*1		*r*1	
8	*r*2	*r*2		*r*2		*r*2	
9	*r*3	*r*3		*r*3		*r*3	

Acknowledgment

The author thanks Dr. K. Inoue, professor of Tokyo Institute of Technology, who read the first draft of this paper and gave several comments on it; as well as Mr. T. Hayashi for his suggestion on the operator grammar.

Reference

1. Knuth, D. E. On the Translation of Languages from Left to Right. *Information and Control* 8, (1965).
2. Aho, A. V. and Ullman, J. D. The Theory of Parsing, *Translation and Compiling*. 1: Parsing. Prentice-Hall.
3. Korenjak, A. J. A Practical Method for Constructing LR(k) Processors. *CACM*, 12, 11, (1969).
4. DeRemer, F. L. Practical Translation for LR(k) Languages. *Project MAC MIT*, (1969).
5. Hayashi, T. On the Construction of LR(k) Analyzer. *Proc. of ACM 1971 National Conference.*
6. Joliat, M. L. Practical minimization of LR(k) parser tables. *Proc. IFIP Congress 1974.*
7. Lalonde, W. R. An Efficient LALR Parser Generator. *Technical Report CSRG-2*, University of Toronto, (February 1971).
8. Kojima, T. et al. LR(k) Parser Generator and Its Application to FORTRAN Compilers (written in Japanese). *Information Processing Society of Japan*, (1974).
9. Aho, A. V. and Johnson, S. C. LR Parsing. *Computing Surveys* 6, 2, (June 1974).
10. Alexander, W. G. et al. Static and Dynamic Characteristics of XPL Programs. *Computer*, (November 1975).
11. Knuth, D. E. An Empirical Study of Fortran Programs. *Software-Practice and Experience*, Wiley Interscience, (1971).
12. Aho, A. V. and Johnson, S. C. Deterministic Parsing of Ambiguous Grammars. *CACM* 18, 8, (1975).
13. Floyd, R. W. Syntactic Analysis and Operator Precedence. *Journal of ACM*, 10, (1963).
14. Earley, J. Ambiguity and Precedence in Syntax Description. *Acta Informatica* 4, (1975), 183-192.
15. Katayama, T. A Simplification of the Context Free Grammar by Converting it to the Context Sensitive Grammar. *Trans. IECE*, 56, 4, (In Japanese).
16. Brosgol, B. M. Articulated LR(k) grammars, *Proc. of the 1975 Conference on Information Science and Systems.*
17. Makinouchi, A. A Practical Method of Analyzer Construction for Extended Simple LR(k) Crammar. *Proc. of the 1975 Conference on Information Science and Systems.*