Acta Informatica 7, 249—268 (1977)

A Practical General Method for Constructing $LR(k)$ Parsers*

David Pager

Received September 16, 1975

Summary. The paper presents in detail the case for $k=1$ of a practical general method for constructing $LR(k)$ parsers. For $k=1$ this method is of rival efficiency to the previous general algorithm described by the author in [21]. The method involves combining the states of an $LR(k)$ parser as they are generated, reducing to a fraction, in the process, the number of configurations that need actually be evaluated, or for which space must be assigned - compared to such general methods as those of [1, 11, 12, 17]. The criteria of compatibility introduced for this purpose are such that the parser obtained is in practice identical in size to, or negligibly larger than, that obtained by resolving the inadequacies of an $LR(0)$ parser (as is done for various subsets of the $LR(k)$ grammars in [5, 8, 14, 20]).

Introduction

In this paper we present details of the case for $k=1$, of a practical *general* method for constructing $LR(k)$ parsers. The original $LR(k)$ construction method by Knuth [12] required an excessive amount of time and space to produce parsers for practical grammars. A number of variations of this method have been devised by which parsers may be constructed with greater efficiency for various subsets of the $LR(k)$ grammars. Among these are the practical methods for the $SLR(1)$ grammars by DeRemer [8], for the $L(m)R(k)$ grammars by Pager [16, 20], and for the $LALR(1)$ grammars by La Londe [14], and Anderson, Eve and Horning [5].

Most grammars for computer languages fall into the above subsets of the $LR(1)$ grammars, and for these the methods mentioned are sufficient. However it is useful to have a practical general method which will also cover those grammars which do not lie in the subsets concerned. This saves the need to rephrase the grammars involved into what might be a less direct or convenient form. In other applications, such as natural language processing, *most* of the grammars employed lie outside the subsets mentioned, and include grammars which are non-LR and ambiguous. Ways of applying the $LR(k)$ technique to grammars such as these are described, using semantic means, by Aho, Johnson and Ullman [4] and Anderson, Eve and Horning [5]. For such grammars there are serious disadvantages to employing a non-general $LR(k)$ method, since this produces a parser with additional inadequacies to those inherent in the properties of the grammar concerned.

No practical general $LR(k)$ algorithm applicable to all $LR(k)$ grammars appears to have been devised, other than the previous method of this kind by the author

* This paper is a development of one of the ideas proposed in Pager [16]. The work was supported by the National Science Foundation under Grant GJ-43362.

[20, 21]. In [20] it is shown that the method put forward by DeRemer [7] for this purpose is invalid. Heuristic means for constructing parsers for any $LR(1)$ grammar, using various definitions of compatibility, have been described in Pager [17], Aho and Ullman [1], and Jolliat [11][1]. While these heuristic methods produce a parser of acceptable size, one has first to generate the entire Knuth $LR(1)$ parser, which itself can be an impractically large task. A number of authors, such as Korenjak [13] and Anderson, Eve and Horning [5], have, for instance, reported on the impracticality of their attempts to construct such an $LR(1)$ parser for ALGOL. One of the most important attributes of the general method, which is described here, is that it reduces the work and space required to an amount which is of the same order as that for constructing an $LR(0)$ parser (as is the case with the *non*-general methods of [5, 8, 14, 20]).

The approach taken by this method bears a contrast to that of the previous general algorithm described in [21]. The latter employs an $LR(0)$ algorithm initially, and then, if the grammar is non-$LALR$, splits states so as to remove conflicts. The present algorithm, on the other hand, avoids the occurrence of conflicts in the first place by employing an $LR(k)$ algorithm, but, at the same time, prevents the combinatorial explosion in the number of states which this normally entails, by combining states as they are generated.

To test the efficiency of the algorithm, we employed 40 student-written grammars, for languages such as ALGOL, BASIC, and various subsets of English, drawn from classes at the University of Hawaii on Compiler Theory and Artificial Intelligence. The various statements in the paper concerning our findings on the practical efficiency of the algorithm are based on the tests made with these grammars.

We will first of all briefly review the $LR(1)$ parsing and parser construction algorithms. In this paper the symbols of a grammar are denoted by Roman letters while Greek letters are employed to denote strings. ε is the null-string. $\alpha \overset{*}{\Rightarrow} \beta$ means β is derivable from α (which is considered to be true if $\beta = \alpha$). By THEADS(α) we refer to the set $\{b \mid b$ is the head of a terminal string derivable from $\alpha\}$. We assume that every symbol of the grammars considered occurs in the derivation of some sentence.

***LR*(1) Parsing.** Consider the grammar G_1:

Production Number	Production
1	$E \to E + T$
2	$E \to T$
3	$T \to a$

[1] The method of Pager [17] has the drawback of relinquishing some of the advantages which the LR method has over precedence techniques with regard to immediate error detection. Aho and Ullmann [3] and Jolliat [11] have adapted the approach employed there so as to avoid this drawback, but there is some doubt as to the effectiveness of the method suplied by Aho and Ullman. In the sole example quoted in its support (Example 9 [1], Exercise 7.3.1 (a) [3, p. 615 and 641]), a parser with 7 states is produced for the grammar considered. However by simply generating the $LR(0)$ machine, resolving its conflicts, and then eliminating final states, one directly obtains a parser with 6 states. (On the other hand if one uses either of the methods of [11] or [17], one obtains a parser for the grammar involved with only 3 states.) Similar results to these have been obtained with regard to all the grammars tried.

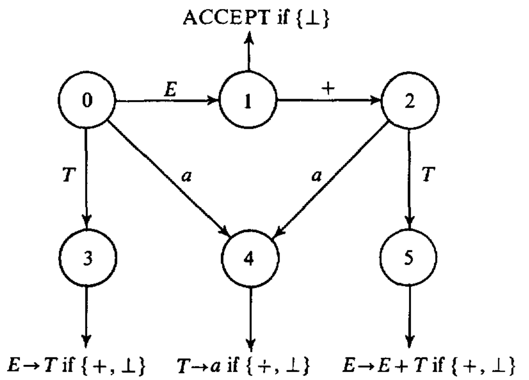

Fig. 1. (a) The parsing machine for G
$E \to E+T \mid T \quad T \to a$

STEP NO.	STACK	REMAINING INPUT
1	0	$a+a\perp$
2	0 4	$+a\perp$
3	0 3	$+a\perp$
4	0 1	$+a\perp$
5	0 1 2	$a\perp$
6	0 1 2 4	$\perp$
7	0 1 2 5	$\perp$
8	0 1	$\perp$
9	ACCEPT	

(b) Using the machine to parse $a+a$

The parser for this grammar can be represented by a parsing machine of the kind shown in Figure 1 (a).

In this figure the circled numbers are called states. The machine is used in conjunction with a stack of state numbers. Initially the stack contains a 0 and the machine is in state 0. An end marker $\perp$ is attached as the rightmost symbol of the string to be parsed. The symbols of this string (called the *input symbols*) are then read in one at a time from left to right, and the stack is manipulated in accordance with instructions associated with the machine's directed lines. For instance, the a labelling the directed line between states 2 and 4 means "If, when in state 2, the next input symbol is an a, make a transition to state 4 by stacking a 4 and putting the machine in state 4; then read the next input symbol to determine what the next applicable instruction is". State 4 in such a case is called the *a-successor* of state 2. The successor relationship is extended by defining the αx-successor of a state S as the x-successor of its α-successor. A different kind of instruction is represented by the legend labelling the arrows leading out of the states. For instance, the legend "$E \to E+T$ if $\{+, \perp\}$" labelling the arrow leading out of state 5 means "If, when in state 5, the head of the remaining input string is $+$ or $\perp$, make a reduction corresponding to $E \to E+T$ by (1) popping as many items as there are in the right-hand side of $E \to E+T$ (i.e., 3) from the stack; (2) stacking the E-successor (E being the left-hand side of $E \to E+T$) of the state now at the top of the stack; and (3) putting the machine in this E-successor state. The head of the remaining input string stays unchanged and is again referred to to determine the next instruction." The set of *contexts* for an x-transition from a state is $\{x\}$. On the other hand, if a state S has an x-successor, then by the *action* for the input symbol x at S we mean the transition to its x-successor state. Similarly, if "p if U" is a conditional reduction at a state, then U is the set of *contexts* for the reduction at the state, and reduction p is the *action* for the members of U. Thus the action for the input symbol a at state 2 is the transition to state 4, whereas the action for $+$ or $\perp$ at state 5 is the reduction $E \to E+T$ The input string is accepted if the machine performs the ACCEPT instruction

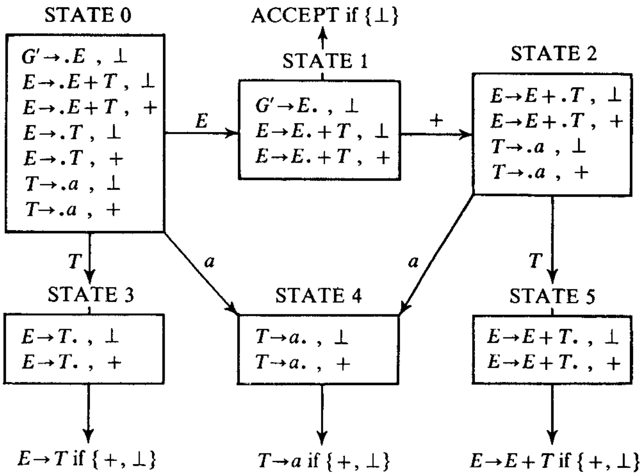

Fig. 2. The detailed $LR(1)$ parsing machine for G_1 showing the configurations associated with each state

(in Fig. 1, if $\perp$ is the next input symbol when the machine is in state 1), and rejected if for the current state no instruction is associated with the head of the remaining input string (e.g., if when in state 2 the next input symbol is $+$). Figure 1 (b) shows the successive contents of the stack in parsing $a+a$.

The Original *LR* (1) Parser Construction Algorithm. The parser construction method described here is that of Knuth [12], and the parsing machines which it constructs are referred to as *Knuth parsing machines*. By a *configuration* $[u \to x_1 \ldots \cdot x_j \ldots x_n, b]$ we mean a production $u \to x_1 \ldots x_n$ with a position marked in its right-hand side (indicated by a marker dot), together with a terminal b, which is referred to as the configuration's *context*. The symbol x_j immediately to the right of the marker dot in the configuration is called the *scanned symbol*. The sets of configurations generated by the algorithm are referred to as *states*. The detailed parsing machine for the grammar G_1, showing the sets of configurations generated with each state, is given in Figure 2.

Here the scanned symbol in, for instance, the first configuration of state 2 is T. By an *immediate successor* of a configuration of the form $[u \to x_1 \ldots \cdot x_j \ldots x_n, b]$ we mean a configuration of the form $[x_j \to \cdot \beta, b']$ for some β, and some $b' \in \mathrm{THEADS}(x_{j+1} \ldots x_n b)$. For instance in Figure 2 the third configuration in state 2 is an immediate successor of the first. The *closure* C of a set S of configurations is the smallest set which contains S and all C's own immediate successors. Thus in Figure 2 the configurations in state 2 are the closure of the set which consists only of the first two configurations listed for that state. The configuration $[u \to x_1 \ldots x_j \cdot x_{j+1} \ldots x_n, b]$ (in which the marker dot has been moved one position to the right) is called a *transition successor*, or more specifically, the *x_j-successor* of $[u \to x_1 \ldots \cdot x_j \ldots x_n, b]$. For instance in Figure 2 the second configuration in state 3 is a T-successor of the fifth configuration in state 0.

In constructing a Knuth parsing machine, an extra production $G' \to G$ is added to the grammar, where G is the original goal symbol, and G' is a symbol not previously in the grammar. G′ now becomes the new goal symbol. State 0 is then taken to be the closure of $\{[G' \to \cdot G, \perp]\}$. If at least one of the configurations in a state S of a parsing machine has an x-successor, for some symbol x, then S has an x-successor state in the machine, which consists of the closure of the x-successors of the configurations in S. For instance, the successor with respect to $+$ of state 1 is state 2, and state 2 is the closure of its first two configurations, which in turn are respectively the successors with respect to $+$ of the last two configurations in state 1.

We can now define the parsing machine for a grammar as the smallest set of states which includes the state 0 involved and all its own successors. Thus to construct the machine in Figure 2, starting with state 0 we evaluate its E, a, and T-successors, then we find all the successor states of the states so formed, and so on. Since there are only a finite number of possible states (sets of configurations), the process must terminate. A state containing configurations of the form $[u \to x_1 \dots x_n \cdot, b_i]$ with the marker dot as shown, for $1 \leqq i \leqq t$, is given the action $u \to x_1 \dots x_n$ if $\{b_1, \dots, b_t\}$, e.g. the reduce action at state 5 arises from the two configurations of that state. The special reduce action $G' \to G$ is taken as the ACCEPT instruction.

If (and only if) a grammar is not $LR(1)$, the application of the $LR(1)$ parser construction algorithm will result in the occurrence of a state at which the context sets associated with different actions are not disjoint, i.e. at which there is more than one action for the same symbol. Such a condition is called a *conflict* (between the actions) at the state concerned, and a parser containing conflicts is invalid.

The New Proposed Parser Construction Algorithm for $k = 1$

We show how to modify the original $LR(1)$ parser construction algorithm so that certain states can be combined as they are generated.

By the *nucleus* of a set of configurations we mean the subset of configurations in which the marker dot is not at the extreme left (e.g. the first two configurations in state 2 constitute its nucleus). Configurations in such a nucleus are called *nucleus configurations*. By the *core* of a set of t configurations $\{[p_i, b_i] \mid 1 \leqq i \leqq t\}$, where the p_i are marked productions and the b_i the associated contexts, we mean the set of marked productions $\{p_i \mid 1 \leqq i \leqq t\}$. For given p and S, the set of configurations $\{[p, b_i] \mid 1 \leqq i \leqq t\}$ involving the same marked production p in state S is referred to as a *config-group* of S, and is represented as $(p, \{b_1, \dots, b_t\})$, where $\{b_1, \dots, b_t\}$ is here referred to as the config-group's *context set*. E.g. $(E \to E + \cdot T, \{+, \perp\})$ is a config-group of state 2 in Figure 2 and its context set is $\{+, \perp\}$. Assuming the config-groups to be ordered in some way, with nucleus configurations preceding non-nucleus ones, by $CG_{i,S}$ we denote the ith config-group of state S, and by $CT_{i,S}$ its context set. Most of the definitions applied to configurations will also be applied to config-groups. For instance a *nucleus config-group* is one containing nucleus configurations, the *nucleus* of a set of config-groups is its subset of nucleus config-groups, and the *core* of a set of config-groups is the core (set of marked productions) of the set of contained configurations involved.

If ζ is a configuration of the form $[A \to \nu \cdot B\beta, x]$ for some A, ν, B, β, x, let us call β its *tail string*. A sequence of configurations $\zeta_1, \ldots, \zeta_t$, where $t \geqq 1$, is called a *lane* if, for $1 \leqq i \leqq t$, ζ_{i+1} is an immediate or transition successor of ζ_i, with ζ_{i+1} being a transition successor of ζ_i for at most one value of i. If no such value of i exists, the lane is called *internal*. Let β_i be the tail string of ζ_i for $1 \leqq i \leqq t$. Then, if $\beta_i \overset{*}{\Rightarrow} \varepsilon$ for each i such that ζ_{i+1} is an immediate successor of ζ_i, the lane is called a *connecting* lane (*connecting* ζ_1 to ζ_t). Let ζ be a configuration in a state S whose tail string is β, and let $b \in \mathrm{THEADS}(\beta)$ for some b, and ζ' be an immediate successor of ζ. Then ζ (or simply S) is said to *generate b* for any config-group containing a configuration ζ'' to which ζ' is connected (which includes ζ'), as well as to any conditional reductions associated with ζ''. If a config-group η contains a configuration connected to one in a config-group η', then η is said to *give its contexts* to η', or to any conditional reductions associated with η'. (Thus contexts are given or generated via connecting lanes to configurations in the same or successor states.) In Figure 2, for example, because of the existence of the connecting lane $[E \to E + \cdot T, +]$ in state 2, $[T \to \cdot a, +]$ in state 2, and $[T \to a \cdot, +]$ in state 4, the config-group $(E \to E + \cdot T, \{+, \perp\})$ in state 2 gives its contexts to the config-group $(T \to a \cdot, \{+, \perp\})$ in state 4, as well as to the conditional reduction at state 4, $T \to a$ if $\{+, \perp\}$. Also, in state 0, the second configuration generates $+$ for the config-group containing the last configuration.

Our algorithm employs a definition of compatibility by means of which the states of a parser are combined. We avoid the (in general) impractical need to first generate the entire $LR(1)$ parser, by combining states as they are generated, in the process reducing to a small fraction the number of configurations and states which are actually evaluated and the amount of space required. For this purpose none of the finite-state machine theory definitions of compatibility employed in [1, 11, 17] are applicable in any way. Besides (i) allowing states to be combined as they are generated for the purpose described above, our definition must have the additional attributes of being, on the one hand (ii) sufficiently restrictive so as to produce a conflict-free parser for all $LR(1)$ grammars, while, on the other, (iii) sufficiently comprehensive so as to produce a parser of similar size to that obtained, for appropriate grammars, by the various nongeneral methods mentioned previously [5, 8, 14, 20] (i.e. a size close to that of an $LR(0)$ parser). The definition that we describe below, which was initially suggested in Pager [16], has these three properties.

Definition 1 of Compatibility. Our first definition of compatibility is motivated by the following intuitive consideration. Conflicts can arise, as a result of combining states S and S' with a common core into a state S'', *only if* we thereby introduce an intersection between the context sets of a pair of config-groups of S'', where no such intersection occurred with regard to the corresponding pair of config-groups in either S or S'.

Let S and S' be two sets of config-groups with a common core C, and let U_r, U'_r denote the sets of contexts associated with the rth member of C in S and S' respectively. Then we say that S and S' are *weakly compatible* if, for all

$1 \leqq i < j \leqq$ number of members of C, at least one of the following is true:

(a) $U_i \cap U_j' = \phi$ and $U_i' \cap U_j = \phi$ or

(b) $U_i \cap U_j \neq \phi$ or

(c) $U_i' \cap U_j' \neq \phi$.

Note that, as is proved in Theorem 4, Appendix II, two states are weakly compatible iff their nuclei are. To test whether two states S and S' whose nuclei have a common core C_n are compatible, check first of all whether for all $1 \leqq i < j \leqq$ number of members of C_n, condition (a) of the definition holds (i.e., $U_i \cap U_j'$ and $U_i' \cap U_j$ are empty, implying compatibility). In practice, almost all pairs of states with a common core are compatible and pass this test.

To avoid introducing more abstract notation at this point, we will postpone presenting the second definition of compatibility (referred to as strong compatibility) until after the description of the new parser construction algorithm. For the moment the word "compatible" in this algorithm can be interpreted as "weakly compatible" (but, as we indicate later, it can also be taken as referring to "strong compatibility"). We define the algorithm in terms of the modifications required to the original $LR(1)$ parser construction algorithm.

The New Parser Construction Algorithm. Instead of the set of config-groups generated by the original algorithm, only the nucleus of this set is stored associated with each state generated. As the nucleus of each state S' is generated, in determining (say) the x-successor of some previously generated state S_b, a check is made to see whether this nucleus is compatible with the sets of config-groups associated with any of the previous states generated[2], and, if so, S' is not introduced as a new state but is instead *merged* into the first such compatible state S in the following way: Let the nucleus config-groups of S' be $\{(p_i, U_i') \mid 1 \leqq i \leqq t\}$ and let those associated with S be $\{(p_i, U_i) \mid 1 \leqq i \leqq t\}$. Then the result of merging S' into S is to associate instead with S the set of config-groups $\{(p_i, U_i \cup U_i') \mid 1 \leqq i \leqq t\}$ and to set S as the x-successor of S_b.

Whenever a state such as S is altered by merging in this way, then we should, in theory (a) re-evaluate its conditional reductions, and (b) regenerate its successors and, if compatible, merge them in turn into the corresponding existing successors. The same effect can be achieved with less work by the context-propagation procedure given below. This reduces (and usually avoids) the need to repetitively regenerate entire portions of the machine, as required in the first given $LALR(1)$ method of Anderson, Eve, and Horning [5]. Let H consist of the nucleus config-groups in S whose context sets were enlarged as a result of merging S' in S, i.e. $H = \{CG_{i,S} \mid U_i' \nsubseteq U_i,\ 1 \leqq i \leqq t\}$. Then (a) can be achieved in the following way: For each conditional reduction p if W in state S, and each

2 An efficient method for doing this is to associate with each symbol a list of states so far generated which are successors with respect to that symbol. A newly generated state S' which is a successor with respect to a symbol x can only be compatible with states in the list associated with x. The search for a previously generated compatible state S can thus be confined to this list. (For ALGOL the average number of comparisons between state nuclei that such a search involves is less than 2.1).

nucleus config-group $CG_{i,S} \in H$ which gives its contexts to that reduction, via an internal connecting lane, replace W by $W \cup (U_i' - U_i)$. We describe the context-propagation procedure for (b) as it applies to a y-successor T of S. If no config-group in H gives its context to any nucleus config-group in T, then T remains unchanged. Otherwise let $\{V_1, \ldots, V_m\}$ be the set of numbers such that, for $1 \leqq r \leqq m$, some member of H gives its context to the nucleus config-group $CG_{V_r,T}$, or else for which context is generated by a configuration in S, and let $NCT_{V_r,T}$ be the set of all the new contexts obtained in this way, i.e. $NCT_{V_r,T}$ is the union of the sets $U_i' - U_i$, for all i such that $CG_{i,S} \in H$ gives its contexts to $CG_{V_r,T}$, plus the set of all contexts generated for $CG_{V_r,T}$ by state S. For $i \notin \{V_1, \ldots, V_m\}$, $NCT_{i,T} = \phi$. If the nucleus config-groups of T with their present contexts are compatible with a set of config-groups with the same marked productions, but with, for each i, the context set $NCT_{i,T}$ associated instead with the ith config-group, then, and only then, the new y-successor of the altered state S is compatible with T, and the effect of merging it with T is obtained by replacing $CT_{V_r,T}$ by $CT_{V_r,T} \cup NCT_{V_r,T}$ for $1 \leqq r \leqq m$. If the new y-successor concerned is not compatible with T, then it must be regenerated as a distinct state.

For ALGOL the average number of nucleus configurations per state is less than 1.3. In our observations, the need to regenerate successors has not occurred with respect to any of the practical grammars tested.

Example. Consider the grammar G_2

$$
\begin{aligned}
X &\to aYd \mid aZc \mid aT \mid bYe \mid bZd \mid bT \\
Y &\to tW \mid uX \\
Z &\to tu \\
T &\to uXa \\
W &\to uV \\
V &\to \varepsilon .
\end{aligned}
$$

The parsing machine obtained for this grammar using the algorithm described above is shown in Figure 3.

Note that states 2 and 8 (and also 3 and 9) are incompatible, and hence have not been merged. An illustration of the context-propagation procedure is provided by the generation of states 1, 4, 7 (which replace, in all, 12 states of the corresponding Knuth machine). Let the initial a and b-successors of state 0 be denoted $1'$, $7'$ respectively, and the result of merging their u-successors $4'$. After the a-successor of $4'$ is evaluated, it is merged into state $1'$ to form the state 1 shown in Figure 3. At this stage, as a result of the merge referred to, a, d, e have been added to the context sets of the three nucleus config-groups of state $1'$, while the context sets of the two nucleus config-groups of state $4'$ are $\{d, e\}$ and $\{\perp\}$. Since state 1 generates d for the first nucleus config-group of state $4'$, $NCT_{1,4'} = \{d\}$; and, since the third nucleus config-group of state 1 gives its contexts to the second nucleus config-group of state $4'$, $NCT_{2,4'} = \{a, d, e\}$. We now verify that a set of config-groups with context sets $\{d, e\}$, $\{\perp\}$ ($CT_{1,4'}$, $CT_{2,4'}$) is compatible with one which has the same core, but with context sets $\{d\}$, $\{a, d, e\}$ respectively ($NCT_{1,4'}$, $NCT_{2,4'}$). As this is the case, we replace $CT_{1,4'}$ and $CT_{2,4'}$ by $\{d, e\}$, $\{a, d, e, \perp\}$ respectively, giving state 4.

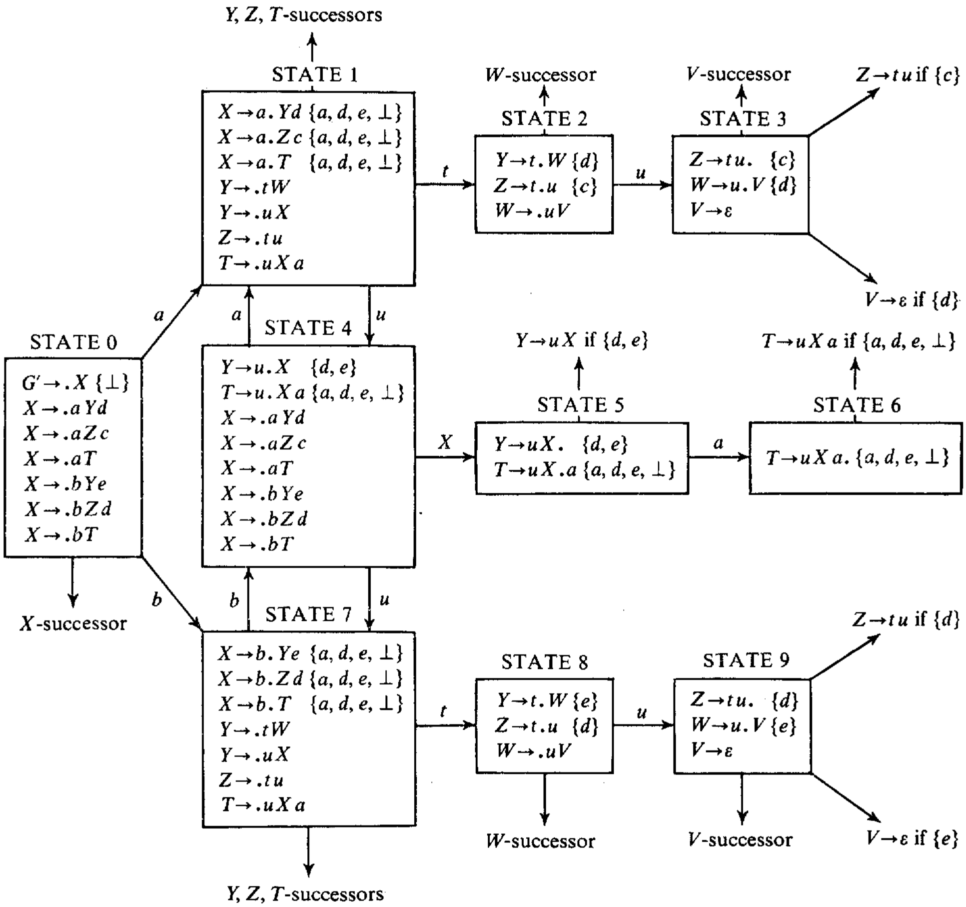

Fig. 3. Part of the parsing machine for G_2 obtained using the paper's algorithm. Only the contexts of the nucleus config-groups are shown (except for the extra production in state 0).

$$X \to aYd \mid aZc \mid aT \mid bYe \mid bZd \mid bT \quad Y \to tW \mid uX \quad Z \to tu \quad T \to uXa \quad W \to uV \quad V \to \varepsilon$$

Intuitive Explanation of Weak Compatibility. Let S_k be an α-successor of state 0 for some string α in a Knuth $LR(1)$ machine M_k and let S be the α-successor of state 0 in the $LR(1)$ machine M obtained for the same grammar using our algorithm. Then we say S *replaces* S_k. Clearly S will have the same core as S_k, and will contain all the configurations of S_k, plus possibly other configurations obtained through merging, and through the context-propagation process. Note that S may be an α-successor of state 0 for many different strings α, and accordingly may replace many different states of M_k. S can have a conflict between a reduction and a transition only if at least one of the states of M_k which it replaces has such a conflict. Thus M has no conflicts of this kind if M_k is conflict-free (Lemma 4, Appendix II).

We say that there is a *potential (i, j) conflict* (with respect to some symbol b) at a state if the context sets associated with its ith and jth config-groups both contain b. Clearly any state T (in M or M_k) with a conflict between reductions must have an associated potential (i, j) conflict for some i, j, while on the other hand any state with the same core as T, which also has a potential (i, j) conflict, must also have a conflict. The key property that M is conflict-free whenever M_k is, now follows immediately from the observation that the algorithm "introduces no new potential conflicts", in the strict sense that a state S of M can have a potential (i, j) conflict for some i, j, only if at least one of the states in M_k which S replaces has a potential (i, j) conflict. This fact can be proved inductively by observing that it is true if S is state 0, and then showing that it remains true for states formed or altered by the operations of constructing successor states, merging states, and carrying out the context-propagation process (Theorem 3, Appendix II).

Definition 2 of Compatibility. For all the practical grammars that we tried, including ones which are not $LALR(1)$, the above algorithm resulted in a parser of precisely the same size as that obtained using the method of [21] or, for grammars within the appropriate subsets, by the methods of [5, 8, 14, 20]. However it is possible to specifically construct grammars (such as G_3, whose parsing machine is shown in Fig. 4) for which the parser produced by the above algorithm turns out to be larger. In practice, the difference in size, if any, should be small, and, in any event, the actual difference between the amounts of space required to store the two parsers will be considerably reduced if one makes use of the methods suggested in [5, 19, 23] for combining the representation of common portions of symbol-action lists. If, nevertheless, one wishes to *ensure* that the size of the parser is reduced to that obtained by the methods of [5, 8, 14, 20, 21], one can make use of the more complex definition of compatibility which is described below. The result then follows from Theorem 6.

If αx is a string and $x \to \eta$ is a production, we write $\alpha x \underset{RR}{\to} \alpha\eta$. A sequence of strings $\alpha_1, \ldots, \alpha_n$ is called a *strong rightmost derivation* (of α_n from α_1) if, for $1 \leqq i \leqq n-1$, $\alpha_i \underset{RR}{\to} \alpha_{i+1}$. If $\omega = \alpha$, or there is a strong rightmost derivation of ω from α, then ω is called a *strong rightmost descendant* of α and we write $\alpha \overset{*}{\underset{RR}{\Rightarrow}} \omega$. Let p_0, p_0' be marked productions, and let the portion of their right-hand sides to the right of the marker dot (the *scanned strings*) be respectively β, β'. Let there further be a string ω with strong rightmost derivations from β, β', which respectively involve applying the (possibly empty) sequence of reductions $p_1, \ldots, p_t$ and $p_1', \ldots, p_s'$, and let the last member of $p_0, p_1, \ldots, p_t$ be different from the last member of $p_0', p_1', \ldots, p_s'$. Under these circumstances, ω is called a *shared descendant* of the scanned strings β, β' of p_0, p_0'.

Let S, S', C, U_r and U_r' be as specified in the first definition of compatibility, and let the rth member of C be denoted by $A_r \to \alpha_r \cdot \beta_r$. Then S and S' are *strongly compatible* if for all $1 \leqq i < j \leqq$ number of members of C,

(a) $U_i \cap U_j' = \phi$ and $U_i' \cap U_j = \phi$ or

(b) β_i and β_j do not share a descendant[3].

3 Consider, in contrast, that (say) $c \in U_i \cap U_j'$ for some i, j, and c, and that β_i and β_j do share a descendant. Let the last productions used in the strong rightmost derivations

A method of determining whether (b) is true or not is given in Appendix I. Note again that, for an $LR(1)$ grammar, two states are strongly compatible iff their nuclei are (Theorem 7, Appendix II).

It can be shown that this definition of compatibility is the widest one possible that can be employed with our algorithm. This is the case since, using *any* criterion of compatibility, conflicts will occur for an $LR(1)$ grammar iff two strongly compatible states are merged (Theorem 6, Appendix II).

Since in practice most states with a common core *are* weakly compatible, it is computationally advantageous to first test for weak compatibility, and only check for strong compatibility if this initial test fails. (From Theorems 3 and 6, Appendix II, weak compatibility implies strong compatibility for all $LR(1)$ grammars.)

There are, moreover, various compromises between the definitions of strong and weak compatibility (which trade exactness for ease of calculation). These are obtained by replacing (b) by a disjunction of conditions which imply it. For instance, let $\beta_i = x_1 \ldots x_n$ and $\beta_j = y_1 \ldots y_m$ and let s and t be the largest numbers such that $x_s \overset{*}{\not\Rightarrow} \varepsilon$, $y_t \overset{*}{\not\Rightarrow} \varepsilon$ respectively; then (b) is true if (say) $s \leqq t$ and $x_1 \ldots x_{s-1} \neq y_1 \ldots y_{s-1}$, or if $s < t$ and x_s is a terminal which is different from y_s, etc. Another such condition, suggested by the referee, is referred to in Appendix I.

Example. Consider grammar G_3

$$\begin{aligned} &X \rightarrow aYd \mid aZc \mid bYe \mid bZd \\ &Y \rightarrow tuv \\ &Z \rightarrow tuw. \end{aligned}$$

The parsing machine obtained for this grammar using the paper's method with respect to weak compatibility is shown in Figure 4. Note that states 2 and 7, while not weakly compatible, are strongly compatible, since uv and uw have no shared descendant. Hence, in the parser obtained using strong compatibility, states 2 and 7 (and also states 3 and 8) would appear combined.

The Algorithm for $k > 1$. This is a straightforward generalization of that given above. The details are given in [24]. By the arguments of Theorems 3 and 5, as is the case with the original algorithm, the application of the $LR(k)$ parser construction algorithm described here will result in a parser which contains conflicts iff the grammar is not $LR(k)$. When this is detected during parser construction, one can (besides rejecting the grammar for reformulation) either (a) increment k by 1 and try the algorithm again; or (b) apply the Lane Tracing Algorithm of [21] to determine the required contexts of length $> k$ and, if necessary, split states so as to remove the conflicts.

As the referee points out, one can drive $LR(k)$ parsers of this kind, where $k > 1$, by variable length patterns, e.g. of the form $**a*b$, where * indicates any terminal symbol. The patterns represent the set of valid contexts for the action involved.

of this descendant from β_i and β_j be $B_i \rightarrow \omega_i$ and $B_j \rightarrow \omega_j$ respectively. Then there will be a state in the resulting machine which contains the configurations $[B_i \rightarrow \omega_i \cdot, c]$ and $[B_j \rightarrow \omega_j \cdot, c]$, and hence possesses a conflict (Lemma 6, Appendix II). (Note that the criteria supplied as to whether two states may be combined are based intuitively on considerations of *in*compatibility, with the more stringent conditions applied before one rules out two states as strongly incompatible.)

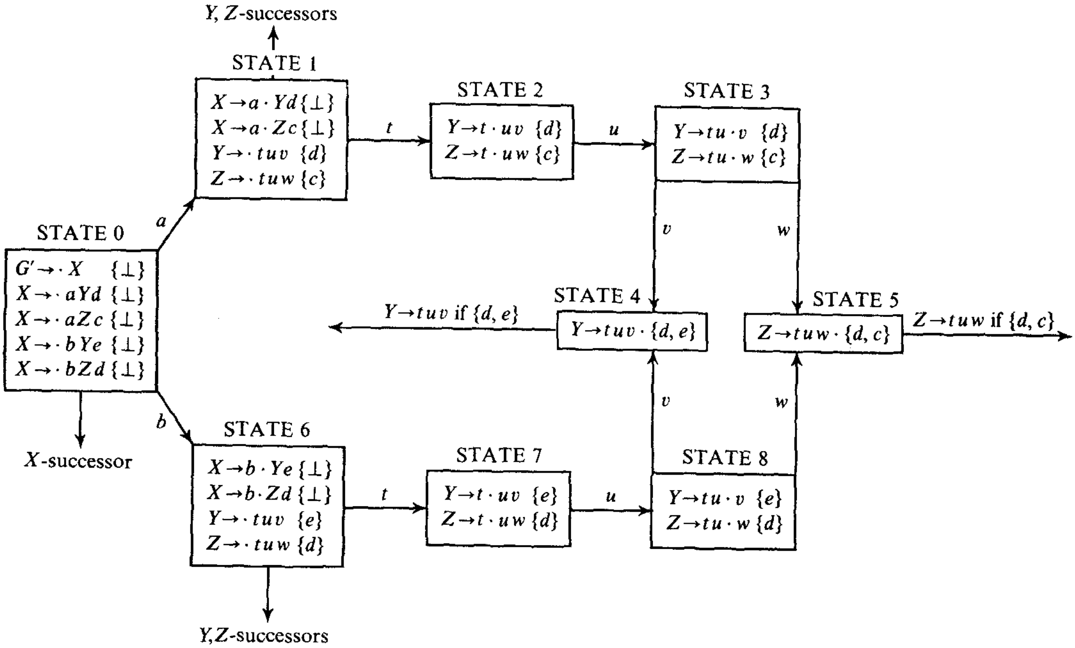

Fig. 4. Part of the parsing machine for G_3 employing weak compatibility

$$X \to aYd \mid aZc \mid bYe \mid bZd \qquad Y \to tuv \qquad Z \to tuw$$

Comparison with Previous General Methods. We have found that the ratio

$$\frac{\text{no. of configurations evaluated by the paper's algorithm}}{\text{no. of configurations evaluated by Knuth's } LR(1) \text{ algorithm}},$$

where the no. of configurations includes repetitions, and configurations obtained by merging and context propagation, is in practice very close to the ratio

$$\frac{\text{no. of states in the Knuth } LR(0) \text{ parsing machine}}{\text{no. of states in the Knuth } LR(1) \text{ parsing machine}}.$$

In all the practical grammars tried, the first ratio was within 5% of the second. (For G_2 both ratios are 0.41.)

The previous general methods [1, 11, 17] compare even less favourably than Knuth's $LR(1)$ algorithm with the paper's method, because they employ the former algorithm in the first instance, and then have to expend the additional effort of attempting to combine states in the resulting machine.

In comparing the paper's method with that of the other general $LR(1)$ algorithm [21], no precise picture emerges from our implementations of the two algorithms. Broadly speaking, the method of [21] is faster when the conflicts which occur are few and computationally simple to resolve, while the present algorithm is faster when there are many 'complex' conflicts, particularly in parsing machines for ill-conditioned non-$LALR(1)$ grammars.

For all the practical grammars that we tried, the algorithm, using weak compatibility, produced a parser of precisely the same size as that obtained by the method of [21] or, where applicable, by the non-general methods of [5, 8, 14, 20].

The present method is simpler conceptually than that of [21], and is easier to program, or to apply by hand. We feel that it deserves attention from both a theoretical and practical point of view.

Appendix I

An algorithm to determine whether two strings α, β share a descendant, where α, β each forms the scanned string of a nucleus configuration for the grammar concerned.

By a *RHS* or *LHS* we mean respectively a right-hand side or left-hand side of a production of the grammar concerned. A string α is said to *vanish* if $\alpha \overset{*}{\Rightarrow} \varepsilon$. By $\mathrm{TAILS}_h(\alpha)$ we mean the set of tails of length h of strings derivable from α. If the RHS of a production p is $z_1 \ldots z_v$, then by $\mathrm{TAIL}(p, r)$ we mean $z_r \ldots z_v$.

To determine whether α, β have a shared descendant, one can first of all, as the referee points out, make such preliminary computationally simple tests as determining whether $\mathrm{TAILS}_h(\alpha) \cap \mathrm{TAILS}_h(\beta) = \phi$ (implying that no such descendant exists) for $h=1$ or 2, or perhaps larger numbers. If, however, non-empty intersections are found for each h tested, and $\alpha = \mathrm{TAIL}(q_1, u_1)$ and $\beta = \mathrm{TAIL}(q_2, u_2)$, then the question can be resolved by employing CALL CHECK(q_1, u_1, q_2, u_2), where CHECK is the recursive procedure described below. Assuming that, as in this case, $(q_1, u_1) \neq (q_2, u_2)$ and neither q_1 nor q_2 has ε as its RHS, the procedure returns *true* iff such a descendant exists.

To prevent the repeated testing for a common descendant of the same pair of strings, the procedure employs a global binary function $f(x, p, r)$ which is set to 1 as soon as the argument (x, p, r) is tested. f is initially zero for all its values and is defined over a domain of arguments (x, p, r) for all x and (p, r) that can appear, in some such triple, as arguments of f in steps 3 and 4 below, e.g. one can restrict x to those nonterminals such that, for some η, $x\eta$ occurs as the tail of a RHS, and η vanishes, while (p, r) must be such that $\mathrm{TAIL}(p, r)$ is defined and non-null[4].

The Recursive Procedure CHECK(p_1, r_1, p_2, r_2)

(1) Let $\mathrm{TAIL}(p_1, r_1) = x_1 \ldots x_n$ and $\mathrm{TAIL}(p_2, r_2) = y_1 \ldots y_m$, and let s, t be the largest numbers such that x_s and y_t do not vanish (if $x_1 \ldots x_n$ or $y_1 \ldots y_m$ themselves vanish, then s or t is zero respectively).

(2) If $s=t=0$, or $x_1 \ldots x_s = y_1 \ldots y_t$, then return *true* and stop.

4 For ALGOL [13] the number of such arguments in the domain of f is 22,168 (68×326). Thus the values of f can in this case be stored as a 2-dimensional bit array (of x versus (p, r)) in an area of $3k$ bytes (moreover, in the CHECK procedure described below, we actually only need to store the arguments (x, p, r) for which $f(x, p, r)$ is set to 1, making hash-code methods possible, if desired). The result of the test implied by f can also be stored in a similar array for use with any subsequent top-level calls of the CHECK procedure.

(3) For each i (if any) such that (a) $\max(1, s) \leqq i \leqq \min(n, t)$ when $s \leqq t$, or $i = s$ when $t < s < m$, and (b) $x_1 \ldots x_{i-1} = y_1 \ldots y_{i-1}$, and (c) $f(x_i, p_2, r_2 + i - 1) = 0$,[5] do the following:

(i) Set $f(x_i, p_2, r_2 + i - 1)$ to 1.

(ii) For each production p_1' of the form $x_i \to \omega$, where ω is non-null, and, in the case where $i = r_2 = 1$, such that $p_1' \neq p_2$, call CHECK $(p_1', 1, p_2, r_2 + i - 1)$.

(4) As in step (3), interchanging the roles of the first and second arguments with those of the third and fourth respectively.

(5) Exit.

A recursive call to the procedure CHECK is made only if $f(x, p, r)$ is 0 for the arguments involved, and, before this call is made, the value of f concerned is set to 1. This places a limit on the total number of recursive calls that could possibly occur. In all the practical grammars that we have tried, we have not found any case in which the total number of such calls has been larger than the number of productions. Thus, in our experience at least, the evaluation of the CHECK procedure has required only a negligible amount of computer time.

Appendix II

Theorems on the Validity of the Paper's Algorithms

In merging a state S' into a state S, we combine the corresponding contexts of their nuclei and make use of the context-propagation procedure. This produces the same parser (with less work) as would be obtained if we instead combined the corresponding contexts of the entire state, and, if this altered any of the contexts of S, re-evaluated its successors. This latter method of merging is referred to as *full* merging. To simplify the proofs of the following lemmas and theorems, we assume throughout that full merging is employed, but the theorems apply equally well to the identical parsers produced by the paper's algorithm. We refer to the machines formed by the algorithm described using our own, *or any other*, definitions of compatibility as *proper machines*.

Theorems 1 and 2 below prove that a conflict-free proper machine has the two important properties cited that we require of a valid parser. We then prove that, using the criteria of compatibility employed by the paper, we do in fact obtain conflict-free proper machines for $LR(1)$ grammars, and, further, that two states of such machines are compatible iff their nuclei are. These last two results are proved first for weak compatibility (Theorems 3 and 4) and then for strong compatibility (Theorems 5 and 7). Theorem 6 shows that strong compatibility is the widest possible valid criterion that can be employed in a proper machine.

---

5 One can evaluate $f(x, p, r)$ in the case where all its values are stored (in column order) as a 2-dimensional bit array, by employing a function POINTER, such that POINTER$(p) + r$ (or a similar expression, e.g. POINTER$(p) + 4r$ for the IBM 360) is an address, whose contents supply the displacement in bits, from the beginning of the array, of the column which provides the values of f when p, r are the second and third arguments. Adding the numeric code which corresponds to x to this displacement, then gives the bit displacement of $f(x, p, r)$.

We describe now some further terminology for use in the proofs. Additional terminology, employed only in Lemma 5, is described immediately before that result. If M is a proper machine, then by M^t we refer to the partially-formed version after t states have been generated, and the symbol t_0 is reserved to denote the final number of states, as occur in M. By M_k we refer to the corresponding Knuth $LR(1)$ machine for the same grammar. The length of a string ω is denoted by $|\omega|$. A *right sentential form* is a string derivable from the goal symbol by a rightmost derivation (in which at each step a production is applied to the rightmost nonterminal). A configuration of the form $[A \to \omega\cdot, x]$ with the marker dot as shown, is called a *final* configuration. By the *scanned string* of a configuration $[A \to \eta\cdot\omega, x]$ we refer to ω. In a parsing machine let $b \in CT_{i,S} \cap CT_{j,S}$, for some b, S, i, j, where $i \neq j$, and let the scanned strings of $CG_{i,S}$, $CG_{j,S}$ have a shared descendant. Then state S is said to contain an *(i, j) conflict generator* with respect to b. Assume that the stack at some stage of a parse contains the state numbers $S_0 S_1 \ldots S_n$, where, for $0 \leqq i \leqq n-1$, S_{i+1} is an x_{i+1}-successor of S_i. Then by the *string in the stack* we refer to $x_1 \ldots x_n$. If after reading r input symbols (i.e. after making r terminal transitions), a parser specifies a reduction p, or specifies error, then it is respectively said to output the *reduce specification* (r, p) or (r, ERROR). If a configuration in a state T generates a symbol b for *a* config-group η in T via an internal connecting lane, then b is said to be *internally generated* in T for η.

Lemma 1. (a) For any α, there exists an α-successor S of state 0 in a proper machine M iff (and in M^t *only if*) there exists an α-successor S_k of state 0 in M_k; and S and S_k here have the same core. (b) For any x, i, S, $x \in CT_{i,S}$ in M iff (and in M^t *only if*) $x \in CT_{i,S_k}$ for some state S_k of M_k which S replaces.

Theorem 1. If the input string α is a sentence of the $LR(1)$ grammar involved, a conflict-free proper machine M will output the same sequence of reduce specifications as M_k.

Proof. Let r_0 be the number of actions performed by M in parsing α. We show by induction that after r actions, where $r \leqq r_0$:

(a) the sequence of reduce specifications outputted by M is identical to that outputted by M_k;

and, for the purpose of proving the inductive step for (a):

(b) the number of symbols read (i.e. terminal transitions performed) by M is the same as that by M_k;

(c) the stack using M consists precisely of the result of replacing states in the stack obtained using M_k by their replacements.

Clearly (a), (b), (c) are true for $r=0$. Assume that they are true for some value of $r < r_0$, and consider the situation after r actions. Let the state at the top of the stack in M be S and that at the top of the stack in M_k be S_k, and let y be the next input symbol. By (c), S is a replacement of S_k, and so, by Lemma 1, if the action at S_k for y is a transition or a reduction π, so is the action at S. (a), (b), (c) of the inductive step now follow immediately, giving the theorem.

Lemma 2. If any state of a Knuth parsing machine is an α-successor of state 0 for some α, then α is the head of a right sentential form.

Proof. This follows directly from Theorem 1, Pager [17][6].

Theorem 2. A conflict-free proper machine M has the property of *immediate error-detection,* in that if $y_1 \ldots y_m$ is the head of a sentence of the $LR(1)$ grammar concerned, but $y_1 \ldots y_m y_{m+1}$ is not, M will report error before reading the $m+1$th symbol of an input string with $y_1 \ldots y_m y_{m+1}$ as its head.

Proof. Let the number of actions M makes in parsing the input string $y_1 \cdots y_m y_{m+1} \cdots y_v$ be r_0. We show, by induction, that after r actions, where $r \leqq r_0$, if k_r is the number of symbols read, then the portion of the input string read so far $y_1 \ldots y_{k_r}$ is derivable from the string in the stack, which will here be the head of a right sentential form. The theorem then follows, since we can deduce that the number of symbols read by M, k_{r_0}, is less than $m+1$, else $y_1 \ldots y_{k_{r_0}}$, and hence in particular $y_1 \ldots y_{m+1}$, would be derivable from the head of a right sentential form; and this in turn would imply that $y_1 \ldots y_{m+1}$ was the head of a sentence.

Note first of all that if α is the string in the stack, then, by the parser-construction algorithm, M is in the α-successor of state 0. By Lemma 1 (a), M_k must then also have an α-successor of state 0, and hence, by Lemma 2, α is the head of a right sentential form. The inductive result to be proved is trivially true for $r=0$. Assume that it is true for some $r<r_0$, and let the contents of the stack after r actions be $S_0 S_1 \ldots S_u$, and let the string in the stack be $x_1 \ldots x_u$. If the $r+1$th action of M is a transition, then the new string in the stack becomes $x_1 \ldots x_u y_{k_{r+1}}$, giving the inductive step since, by the inductive hypothesis, $y_1 \ldots y_{k_r}$ is derivable from $x_1 \ldots x_u$. If, on the other hand, the $r+1$th action of M is a reduction $A \rightarrow x_{u-n+1} \cdots x_u$ (from the parser-construction process, it must be of this form, for some n), then $k_r = k_{r+1}$, and the string in the stack becomes $x_1 \ldots x_{u-n} A$, from which $x_1 \ldots x_u$, and hence turn $y_1 \ldots y_{k_{r+1}}$, is derivable. This completes the inductive step, giving the theorem.

Lemma 3. A proper machine for a non-$LR(1)$ grammar possesses conflicts. (From Lemma 1 (b).)

Lemma 4. A proper machine M for an $LR(1)$ grammar has no conflicts between transitions and reductions. (From Lemma 1, since this would imply a similar conflict in M_k.)

Theorem 3. A proper machine M, in which all the states merged are weakly compatible, has a conflict iff the grammar involved is not $LR(1)$.

Proof. In view of Lemmas 3 and 4, it is sufficient to prove the following Proposition A, which implies that the grammar is not $LR(1)$ if M has conflicts between reductions. Proposition A: For $1 \leqq t \leqq t_0$ and any i, j, M^t has a potential (i,j) conflict at a state S only if M_k also has a potential (i,j) conflict at some state which S replaces. This is clearly true for $t=1$, since the states 0 of M^1 and M_k are identical. Assume that it is true for $t=r$ where $r<t_0$, and let the $r+1$th state T evaluated be an x-successor of a state S. It is sufficient for the inductive step to prove that Proposition A remains true after T is generated, since, from

6 Or, for those familiar with the terminology of [3], from the restatement of the same result in Theorem 5.10.

the definition of weak compatibility, T can be merged into an existing state only if it introduces no new potential conflicts into that state; and a similar comment applies to the propagation of contexts. Assume that T has a potential (i, j) conflict for some i, j. Then there is some symbol b such that $b \in CT_{i,T} \cap CT_{j,T}$. If b is internally generated for $CG_{i,T}$, then it will be internally generated for the ith configuration of every state in M_k which T replaces, including that state T_k, which by Lemma 1 (b) must exist, where $b \in CT_{j,T_k}$. Thus T_k has a potential (i, j) conflict in this case, providing the inductive step. The step concerned follows, similarly, if b is internally generated for $CG_{j,T}$. If, on the other hand, a single configuration of S gives its contexts, including b, to both $CG_{i,T}$ and $CG_{j,T}$, then clearly there will be a potential (i, j) conflict in the x-successor of every state in M_k which S replaces, again giving the inductive step.

The only other possible remaining case is where two different configurations of S, say $CG_{i_0,S}$, $CG_{j_0,S}$, such that $b \in CT_{i_0,S}$, and $b \in CT_{j_0,S}$, give their respective contexts to $CG_{i,T}$, $CG_{j,T}$. But in this case, S has a potential (i_0, j_0) conflict, and hence, by the inductive assumption, so has some state S_k of M_k which S replaces. It then follows that the x-successor of S_k must have a potential (i, j) conflict. This completes the inductive proof of Proposition A.

Theorem 4. Two states of a partially-formed proper machine are weakly compatible iff their nuclei are.

Proof. The theorem is trivially true if the states S and S' do not have a common core. Assume that they do. Clearly, from the definition of weak compatibility, if the nuclei are weakly incompatible, then so are S, S'. We show conversely that, if S, S' are weakly incompatible, then so are their nuclei. If S, S' are weakly incompatible, there exist b, i, j, where $i \neq j$, such that

$$b \in CT_{i,S} \cap CT_{j,S'} \text{ but } CT_{i,S} \cap CT_{j,S} = CT_{i,S'} \cap CT_{j,S'} = \phi. \tag{1}$$

Since b occurs in $CT_{i,S}$, either (a) some nucleus configuration $CG_{i_n,S}$ such that $b \in CT_{i_n,S}$ gives its contexts to $CG_{i,S}$, or (b) b is internally generated for $CG_{i,S}$. But, since S and S' have a common core, case (b) would imply that $b \in CT_{i,S'}$ in contradiction to (1). We conclude that (a) is true, and that, similarly, some nucleus configuration $CG_{j_n,S'}$ such that $b \in CT_{j_n,S'}$ gives its contexts to $CG_{j,S'}$. But then $CG_{j_n,S}$ must in turn give its contexts to $CG_{j,S}$. It follows that, for any c, $c \in CT_{i_n,S} \cap CT_{j_n,S}$ implies that $c \in CT_{i,S} \cap CT_{j,S}$, which is in contradiction to (1). Hence $CT_{i_n,S} \cap CT_{j_n,S} = \phi$, and similarly $CT_{i_n,S'} \cap CT_{j_n,S'} = \phi$. Since $b \in CT_{i_n,S} \cap CT_{j_n,S'}$, the nucleus of S and S' have hereby been shown to be weakly incompatible, proving the theorem.

Let L be a sequence of configurations $\zeta_1, \ldots, \zeta_n$. Then L is called a $\overset{*}{\textit{connecting}}$ lane ($\overset{*}{\textit{connecting}}$ ζ_1 to ζ_n) if there is a sequence of numbers $n_1 \leqq n_2 \ldots \leqq n_t$ such that $n_1 = 1$ and $n_t = n$ and, for $1 \leqq i \leqq t-1$, each of the segments of L $\zeta_{n_i} \zeta_{n_i+1} \cdots \zeta_{n_{i+1}}$ is a connecting lane (so that L consists of a concatenation of connecting lanes). L is further said to *spell out* $x_1, \ldots, x_m$ if $x_1, \ldots, x_{m+1}$ are, in order, the scanned symbols in the nucleus configurations among $\zeta_1, \ldots, \zeta_n$ (implying that the states through and to which L passes are, in order, successors with respect to $x_1, \ldots, x_m$).

Note that in a proper machine if a configuration of $CG_{i,S}$ is $\overset{*}{\text{connected}}$ to one of $CG_{j,T}$ for some i, j, S, T, then $CT_{i,S} \subseteq CT_{j,T}$. (Lemma 6 below makes use of this observation.)

If $\alpha_1, \ldots, \alpha_m$ is a strong rightmost derivation D, where $\alpha_1 = x_1 \ldots x_n$, and if r is the largest number, if any, such that x_r, as a result of the productions applied, *including* one to itself, has a non-null descendent in α_m, then we call r the *critical number* of α_1 (with respect to the derivation D concerned). For $t > 1$, the critical number of α_t is simply its critical number (if any) in the derivation $\alpha_t, \ldots, \alpha_m$. Note that a strong rightmost derivation D of this kind has the following properties with respect to the critical number r of α_1 involved: (a) productions must be applied so as to obtain $x_{r+1} \ldots x_n \overset{*}{\Rightarrow} \varepsilon$ (b) for $1 \leqq i \leqq r-1$, no productions are applied to x_i in α_1 or in its corresponding occurrence in $\alpha_2, \ldots, \alpha_m$.

Lemma 5[7]. Let a configuration ζ, whose scanned string is β, occur in a proper machine. The following propositions hold in this case. (a) For any ω, if ω has a strong rightmost derivation from β in which the last production applied is p, then there is a $\overset{*}{\text{connecting}}$ lane, which spells out ω, from ζ to a final configuration whose production is p. (b) On the other hand, if there is a $\overset{*}{\text{connecting}}$ lane, which spells out ω, from ζ to a configuration whose scanned string is β', then $\beta \underset{RR}{\overset{*}{\Rightarrow}} \omega\beta'$.

Proof of (a). The result is clearly true for $|\omega| = 0$. Assume that it is true for all $|\omega| \leqq t$. Let ζ be a configuration in a state S whose scanned string is β, and let $\alpha_1 (=\beta), \ldots, \alpha_m (=\omega)$ be a strong rightmost derivation from β of a string ω such that $|\omega| = t+1$. If no member of the derivation has a critical number > 1, then ζ is connected to a configuration in S with a marked production of the form $y \to \cdot\, \omega\eta$ for some η such that $\eta \overset{*}{\Rightarrow} \varepsilon$, where $y \to \omega\eta$ is the production applied to the head of the last member of the derivation which has a critical number. The inductive step clearly holds in this case. Otherwise let t be the smallest number such that the critical number of α_t is > 1. Let $\alpha_t = x_1 \ldots x_n$, let its critical number be r, and let the first production applied to the occurrence of x_r in the derivation be $x_r \to \mu$. From the properties (a) and (b) cited for strong rightmost derivations with respect to critical numbers, it follows in this case that ζ is $\overset{*}{\text{connected}}$ to a configuration with marked production $x_r \to \cdot \mu$ in the $x_1 \ldots x_{r-1}$-successor of S, and that ω must be expressible as $x_1 \ldots x_{r-1}\omega'$ for some ω' such that ω' is a strong rightmost descendant of μ. The inductive step now follows from the inductive hypothesis since $|\omega'| \leqq t$.

Proof of (b). This follows easily by a simple induction on the length of the $\overset{*}{\text{connecting}}$ lane.

Lemma 6. A proper machine has a conflict if it contains a state with a conflict generator. (From Lemma 5 (a).)

Theorem 5. A proper machine M in which all the states merged are strongly compatible has conflicts iff the grammar is not $LR(1)$.

---

7 Note, incidentally, that (a) and (b) can be strengthened to form converses of each other.

Proof. In view of Lemmas 3 and 4, it is sufficient to show that if the grammar is $LR(1)$, M contains no conflicts between reductions. We do this by proving by induction that no state of M^t has a conflict generator in this case. This is clearly true for $t=1$, by Lemma 6 (since states 0 of M^1 and M_k have the same set of configurations). Assume that it is true for some value of $t<t_0$. Let the $t+1$th state T be an x-successor of a state S and assume that T *does* have an (i,j) conflict generator with respect to b for some i, j, b. If b were internally generated for $CG_{i,T}$ or $CG_{j,T}$, or b were in the contexts given by a single configuration of S to both $CG_{i,T}$ and $CG_{j,T}$, then, as argued in Theorem 3, M_k would possess a potential (i,j) conflict in some state which T replaces, and hence by Lemma 6, M_k would possess a conflict. Thus there must be distinct config-groups $CG_{i_0,S}$, $CG_{j_0,S}$ such that $b\in CT_{i_0,S}$ and $b\in CT_{j_0,S}$, which respectively give their contexts to $CG_{i,T}$, $CG_{j,T}$. From Lemma 5(b), we conclude that, if β_{i_0}, β_{j_0}, β_i, β_j are respectively the scanned strings in $CG_{i_0,S}$, $CG_{j_0,S}$, $CG_{i,T}$, $CG_{j,T}$, then $\beta_{i_0}\overset{*}{\underset{RR}{\Rightarrow}}x\beta_i$ and $\beta_{j_0}\overset{*}{\underset{RR}{\Rightarrow}}x\beta_j$. It follows here that S has a conflict generator, which constitutes a contradiction to the inductive hypothesis. We conclude that T does *not* in fact have an conflict generator. From the definition of strong compatibility, if T is merged into a previously generated state, then since neither state has a conflict generator, the result will not have one either. Further, no conflict generators are introduced as a result of context propagation. This completes the inductive proof.

Theorem 6. A proper machine for an $LR(1)$ grammar has conflicts iff two strongly incompatible states are merged during its formation. (From Lemma 6 and Theorem 5.)

Theorem 7. Two states of a partially-formed proper machine for an $LR(1)$ grammar are strongly compatible iff their nuclei are.

Proof. We show that if two states S, S' with the same core are strongly incompatible, then so are their nuclei. If S, S' are strongly incompatible, then there exists b, i, j, where $i\neq j$, such that $b\in CT_{i,S}\cap CT_{j,S'}$, while, if β,β' are respectively the scanned strings of $CG_{i,S}$, $CG_{j,S'}$, then β, β' have a shared descendant. *If* b were internally generated for $CG_{i,S}$ or $CG_{j,S'}$ then, as argued in Theorem 5, M_k would possess a conflict. Since this is impossible, we conclude that some nucleus configurations of S, S' $CG_{i_n,S}$, $CG_{j_n,S'}$, such that $b\in CT_{i_n,S}$ and $b\in CT_{j_n,S'}$, give their respective contexts to $CG_{i,S}$, $CG_{j,S'}$. Here again, by the argument in Theorem 5, we can conclude that $i_n\neq j_n$. But, if the scanned strings of $CG_{i_n,S}$ and $CG_{j_n,S'}$ are respectively β_n, β'_n, then it follows from Lemma 5(b) (with $\omega=\varepsilon$) that $\beta_n\overset{*}{\underset{RR}{\Rightarrow}}\beta$ and $\beta'_n\overset{*}{\underset{RR}{\Rightarrow}}\beta'$. β_n, β'_n accordingly also have a shared descendant. Thus the nuclei of S, S' are strongly incompatible, giving the theorem.

References

1. Aho, A. V., Ullman, J. D.: Optimization of $LR(k)$ parsers. J. Computer and System Sciences **6**, 573–602 (1972)
2. Aho, A. V., Ullman, J. D.: A technique for speeding up $LR(k)$ parsers. SIAM J. Computing **2**, 106–127 (1973)
3. Aho, A. V., Ullman, J. D.: The theory of parsing, translation, and compiling. Englewood Cliffs (N.J.): Prentice-Hall 1973

4. Aho, A. V., Johnson, S. C., Ullman, J. D.: Deterministic parsing of ambiguous grammars. Proc. ACM Symposium on Principles of Programming Languages, Boston, Mass., Oct. 1973
5. Anderson, T., Eve, J., Horning, J. J.: Efficient $LR(1)$ parsers. Acta Informatica **2**, 12–39 (1973)
6. Booth, T. L.: Sequential machines and automata theory. New York: Wiley 1967
7. DeRemer, F. L.: Practical translators for $LR(k)$ languages. Mass. Inst. of Tech., Cambridge (Mass.), Project MAC, Tech. Rep. MAC TR-65, Oct. 1969
8. DeRemer, F. L.: Simple $LR(k)$ grammars. Comm. ACM **14**, 453–460 (1971)
9. Ginsburg, S.: An introduction to mathematical machine theory. Reading (Mass.): Addison-Wesley 1962
10. Hayashi, K.: On the construction of $LR(k)$ parsers. Proc. ACM 1971 Annual Conf., ACM, New York, pp. 538–553
11. Joliat, M. L.: On the reduced matrix representation of $LR(k)$ parser tables. Computer Systems Research Group, University of Toronto, Tech. Report CSRG-28, Oct. 1973
12. Knuth, D. E.: On the translation of languages from left to right. Information and Control **8**, 607–639 (1965)
13. Korenjak, A. J.: A practical method for constructing $LR(k)$ processors. Comm. ACM **12**, 613–623 (1969)
14. LaLonde, W. R.: An efficient LALR parser generator. Computer Systems Research Group, University of Toronto, Tech. Report CSRG-2, 1971
15. McKeeman, W. M., Horning, J. J., Wortman, D. B.: A compiler generator. Englewood Cliffs (N.J.): Prentice-Hall 1970
16. Pager, D.: Some ideas for left-to-right parsing. University of Hawaii, Information Sciences Program, Tech. Report No. PE 84, Oct. 1970
17. Pager, D.: A solution to an open problem by Knuth. Information and Control **17**, 462–473 (1970)
18. Pager, D.: Conditions for the existence of minimal closed covers composed of maximal compatibles. IEEE Trans. Computers C-**20**, 450–452 (1971)
19. Pager, D.: Efficient programming techniques for $LR(k)$ parsing. Proc. Seventh Annual Princeton Conf. on Information Sciences and Systems, Princeton University, Princeton, 1973; University of Hawaii, Information Sciences Program, Tech. Report No. PE 236, Jan. 1972
20. Pager, D.: On the incremental approach to left-to-right parsing. University of Hawaii, Information Sciences Program, Tech. Report No. PE 238, Jan. 1972
21. Pager, D.: The lane tracing algorithm for constructing $LR(k)$ parsers and ways of enhancing its efficiency. Information Sciences (to appear)
22. Pager, D.: On eliminating unit productions from $LR(k)$ parsers. In: Loeckx, J. (ed.), Automata, languages, and programming, 2nd Colloquium. Lecture Notes in Computer Science **14**. Berlin-Heidelberg-New York: Springer 1974
23. Pager, D.: A compaction algorithm for combining the symbol-action lists of an $LR(k)$ parser. University of Hawaii, Information Sciences Program, Tech. Report No. PE 259, July 1972
24. Pager, D.: On the decremental approach to left-to-right parsing. University of Hawaii, Information Sciences Program, Tech. Report No. PE 252, July 1972

Professor David Pager
Dept. of Information and Computer Sciences
University of Hawaii
2565 The Mall
Honolulu, Hawaii 96822, USA