Generating LR(1) Parsers of Small Size

Fortes Gálvez, José *

Laboratoire I3S, CNRS – Université de Nice-Sophia Antipolis
Bât. 4, 250 Avenue Albert Einstein, F-06560 Valbonne, France
e-mail: fortes@mimosa.unice.fr

Abstract. Classic LR(1) parsing methods have the problem of producing too large parsing tables for programming language grammars. An alternative is to build an automaton that combines the lookahead symbol with reading the parsing stack from its top, to determine the next parsing action.
The building procedure for such a parser and its use for parsing are presented through an example grammar. Results from an experimental implementation of the parser generator show important reductions in automaton size in comparison with standard LR methods. Some discussion is presented, suggesting that the theoretical drawbacks of the method are of relatively little practical importance.

1 Introduction

LR(1)[10] is a very important class of context-free grammars, from both the theoretical and practical viewpoints. It is the largest class of grammars in which (bottom-up) parsing can progress as the input text is read in a left-to-right scan, with only one symbol lookahead. It is thus of practical interest to be able to automatically build efficient parsers for such a class of grammars. But classic LR(1) parsing methods[1, 14] have the disadvantage of producing unacceptable large automata in relation to grammar size. This is specially relevant when building parsers for programming language grammars.

We take here another approach for LR(1) parsing, which is originally based on the family of precedence parsing methods[1, 11]. These methods were very popular for their efficiency until the advent of LR methods, that greatly surpassed them in parsing power. Since then, precedence has been almost forgotten, and much research effort has concentrated into reducing the storage needs of parsers while preserving their LR(1) parsing power, without much success. Thus, the currently recognized method of choice is LALR(1)[3, 14], where a practical compromise is found by reducing automata size at the cost of some parsing power. We propose here to further investigate whithin the previous theoretical framework, to obtain LR(1) parsing power while preserving the precedence methods efficiency.

* Supported by a grant from the Government and the *Caja Insular de Ahorros* of the Canary Islands. Future correspondence should be addressed to: Departamento de Informática, Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain. e-mail: fortes@fi.upcan.es.

1.1 Our approach for LR(1) parsing

Precedence parsing basic idea is that the most important information for parsing is very frequently near the top of the stack—always considering bottom-up parsing. Thus simple and weak precedence[12, 15, 8], for instance, use the (top-of-stack, lookahead) symbol pair to build precedence relations, and then to use them to know whether to shift or reduce. These methods fail when the needed information is deeper in the stack. A partial solution is given by bounded-context methods[4, 6, 11], in which a deeper—although limited in advance—search in the stack is made. These methods also have the problems of deciding what rule to reduce, and most precedence methods impose strong restrictions on the right-part of grammar rules. In most cases empty rules are forbidden.

In essence, what we need in a bottom-up parser is some procedure to tell us what parsing action to perform next: shift, reduce according to the i-th rule, or error. In classic LR methods the relevant stack information is "condensed" in a state, and a (state, lookahead) table gives the answer. The variety of such stack classes leads to a big number of states in an LR(1) table.

After all, the basic idea of precedence parsing remains true. Let us examine what happens if we bring this idea to its end. In an LR(1) grammar we should be always able to decide the next parsing action upon the current stack-plus-lookahead contents. Experience with precedence methods tells us that very frequently we only need to read a few symbols in the stack, plus the lookahead, to make a decision. For some unfrequent cases we would need to look deeper in the stack. Thus, in principle, it is possible to build an automaton that ***begins reading the stack from its top*** to decide, in combination with the lookahead symbol, which parsing action to perform next. For the above reason, this automaton should be quite small in its depth, and we should frequently use only a few of its states. In this paper it is shown how to build such an automaton. For a more detailed description of the construction procedure, see [5].

2 A discriminating automaton

Our method for building an action-decision automaton is to begin building a (reverse) recognizing automaton for the stack, where each state has the information of what parsing actions are compatible with the portion of the stack read so far from its top. As soon as the next (lower) stack symbol or the lookahead symbol is enough to decide the parsing action to perform, no further states are built, since the parsing action will be unique from this point on. There is an exception, necessary to assure valid reductions: when the only possible action is to reduce, and the rule right-part (i.e. the handle at the top of the stack) has not been completely read yet, state construction should continue until the left end of this right-part, just to check its correctness.

The following explanation of the automaton construction procedure will be developed through the use of the following LR(1) example grammar, where S is the augmented start symbol, and "$\vdash$" and "$\dashv$" represent the begin and end markers for the input text, respectively. Each rule is given a reference number, shown between

parentheses.

$$
\begin{array}{ll}
S \rightarrow \vdash E & (1) \\
E \rightarrow E + T & (2) \\
E \rightarrow T & (3) \\
T \rightarrow FR & (4) \\
R \rightarrow \varepsilon & (5) \\
R \rightarrow * FR & (6) \\
F \rightarrow (E) & (7) \\
F \rightarrow a & (8)
\end{array}
$$

For our automaton initial state we should consider all the possible legal parsing actions: shift and all the rules for reduction. Let us consider some cases. For instance, it should consider the possibility of being in a stack-plus-lookahead configuration where "(" is the current lookahead symbol: according to the grammar, the only possible action is to shift, i.e. to push "(" on top of the stack and read the next symbol. In this case, this decision can be made just by looking at the "(". For another possibility, consider having "a" as the top-of-stack symbol: the only possible parsing action is to reduce "a" to "F" according to rule 8. On the other hand, when the top-of-stack symbol is "R", we should check that the following (looking backwards) symbol in the stack is "F"—the only legal possibility—and decide the reducing rule according to the next stack symbol: rule 6 for "$*$", or else rule 4. Let us see how to automatically develop such a decision process.

2.1 Situations

To build the automaton we will make use of *situations*. A situation conveys all the needed information for all its compatible stack-plus-lookahead configurations. A situation like

$$F \rightarrow (\cdot E)\ (3)\ +$$

means that, according to the current automaton state, stack exploration may be—amongst other possibilities indicated by the other situations at the same state—between "(" and "E)" in rule "$F \rightarrow (E)$" and that, if current lookahead is "+", we should reduce according to rule 3. For instance, the following rightmost sentential form is compatible with that situation:

$$\vdash F * (T \underline{+} a) + a \dashv$$

where the underlining indicates that "+" is the current lookahead symbol, and thus the corresponding stack-plus-lookahead configuration is

$$\vdash F * (T \underline{+}$$

We can see that, effectively, according to rule 3, "T" should be reduced to "E". After some more reductions, we will find the sentential form

$$\vdash F * (E) \underline{+} a \dashv$$

where "(E)" will be reduced to "F": this is why rule "$F \rightarrow (E)$" appears in the situation, since we may need to read its right-part "(" in order to decide amongst

other competing actions. Finally, we should note that the situation lookahead "+" is located, according to the original sentential form, between the dot and the right end of the rule.

Let us now consider this last configuration. After having read—in reverse order—")", "E", and "(", we should be in a situation containing "$F\rightarrow \cdot\,(E)$". Obviously, there should exist a situation indicating reduction with rule 7 when lookahead is "+". But it is important to note that, in this case, the situation lookahead lies just to the right of the situation right-part. We will indicate this particular circumstance by marking the lookahead with an apostrophe: "+′". An additional consideration in this case—although not necessarily for all the situations with the dot on the left of the right-part— is that the handle to reduce is the situation right-part itself. It is important to note this circumstance in order to check the presence of the full right-part in top of the stack before performing its reduction. We mark the situation rule number for this purpose. With these two considerations, the referred situation should be written

$$F\rightarrow \cdot\,(E)\ (7')\ +'$$

2.2 A partial stack-recognizing automaton

Before explaining how to build the full decision automaton, let us see how to build a recognizing automaton for simpler cases. For instance, let us try to build a recognizing reverse automaton for all the legal stack contents where the stack-plus-lookahead configuration indicates reduction with rule 5, i.e. "$R\rightarrow\varepsilon$". The resulting automaton will not be a minimum-state automaton, for we wish to distinguish two states when their sets of compatible lookaheads are different, in order to use this information later, when building the full automaton.

Initially, we have to consider a set of situations where the dot is at the end of the right-part, with all the possible *Follow*(R) as lookahead. Thus we initially have the following set of situations:

$$\begin{array}{l} R\rightarrow\cdot\ (5')\ \dashv' \\ R\rightarrow\cdot\ (5')\ +' \\ R\rightarrow\cdot\ (5')\ )' \end{array}$$

When several situations like the previous ones differ on just the lookahead symbols, we use the convenient shorthand:

$$R\rightarrow\cdot\ (5')\ \dashv'\ +'\ )'$$

When the rule to reduce is not empty, the first automaton states should obviously be devoted to recognize its right-part from right to left, and then for recognizing all its possible left contexts. In our example, with an empty right-part, the automaton immediately begins recognizing this rule left context. For this purpose, in general, we have to consider, for each situation where the dot is on the left of the right-part, all the rules containing in their right-parts the situation left-part nonterminal, that are compatible with the situation lookahead. For instance, for a situation like

$$R\rightarrow\cdot\ (5')\ +'$$

we have to consider "$T \rightarrow FR$" and "$R \rightarrow * FR$", from which the following *inferred* situations are obtained:

$$\begin{array}{l} T \rightarrow F \cdot R \quad (5) +' \\ R \rightarrow * F \cdot R \, (5) +' \end{array}$$

This mechanism permits to know what symbols can follow, in order to be able to progress in the automaton construction. Performing this *closure* process we get the following set of situations associated with the initial state:

$$\begin{array}{ll} R \rightarrow \cdot & (5') \dashv' \; +')' \\ T \rightarrow F \cdot R & (5) \; \dashv' \; +')' \\ R \rightarrow * F \cdot R & (5) \; \dashv' \; +')' \end{array}$$

Clearly, in our example the only possibility is to have "F" as the next (topmost, in this case) stack symbol, since "F" is on the left of the dot in all the situations—situations with the dot at the left end are now useless for transitions. Thus we have a transition from initial state through "F" to a second state, with the following situations:

$$\begin{array}{l} T \rightarrow \cdot FR \quad (5) \dashv' \; +')' \\ R \rightarrow * \cdot FR \, (5) \dashv' \; +')' \end{array}$$

The second set of situations clearly indicates that the next stack symbol can legally be "$*$". The first set needs a closure computation, producing:

$$\begin{array}{ll} E \rightarrow E + \cdot T & (5) \dashv' \; +')' \\ E \rightarrow \cdot T & (5) \dashv' \; +')' \\ S \rightarrow \vdash \cdot E & (5) \dashv' \; + \\ E \rightarrow \cdot E + T & (5) + \\ F \rightarrow (\cdot E) & (5) +) \end{array}$$

We finally conclude that from this second state there are transitions with "$*$", "$+$", "$\vdash$", and "$($" to other states, whose sets of situations are obtained from these ones after a dot displacement one symbol to the left.

Proceeding in this way, the reader can verify that the recognizing automaton can be completely built. When two states have equal sets of situations, we consider them to be the same state—although this criterion might be refined. Final states are those containing "$S \rightarrow \cdot \vdash E$" situations, as the one obtained through the previous "$\vdash$" transition.

In an analogous way, we could build the recognizing automaton for every rule in the grammar. Let us now consider another case: how to build a recognizing automaton for all the legal stack contents where the stack-plus-lookahead configurations indicate that the next parsing action is to shift. For this purpose, we just have to consider every rule in the grammar, and every position of the dot before a terminal symbol in its right-part—i.e. the current lookahead symbol. Here is the exhaustive

listing for all these possibilities, where we code a shift action with a "(0)":

$$\begin{array}{ll} S\rightarrow\cdot\vdash E & (0)\vdash \\ E\rightarrow E\cdot+T & (0)+ \\ R\rightarrow\cdot*FR & (0)* \\ F\rightarrow(E\cdot) & (0)) \\ F\rightarrow\cdot(E) & (0)(\\ F\rightarrow\cdot a & (0)\,a \end{array}$$

Since some situations have the dot at the beginning of their right-parts, a closure computation should be performed, resulting in the following additional set of situations:

$$\begin{array}{ll} T\rightarrow F\cdot R & (0)* \\ R\rightarrow *\,F\cdot R & (0)* \\ T\rightarrow\cdot FR & (0)(a \\ R\rightarrow *\cdot FR & (0)(a \\ E\rightarrow E+\cdot T & (0)(a \\ E\rightarrow\cdot T & (0)(a \\ S\rightarrow\vdash\cdot E & (0)(a \\ F\rightarrow(\cdot E) & (0)(a \end{array}$$

From now on, and following the same procedure, the recognizing automaton can be built.

2.3 A simple discriminating automaton

Let us now consider the following problem. We are given all the possible stack-plus-lookahead configurations where the possible parsing actions are to shift, or to reduce according to precisely rule 5. Our task is to classify each configuration as belonging to one or another class, i.e. we have a two-option discrimination problem.

If the grammar is LR(1), a trivial solution is to build both stack-recognizing automata, one for reducing using rule 5, another for shifting. We can read each stack-plus-lookahead configuration from top to bottom of the stack, using both automata. In most cases, one of the automata will be unable to recognize the current stack, thus indicating the other automaton's action as the correct one. Anyway, it is possible that in both automata a final state is reached. But since the grammar is LR(1), the situation lookaheads in both final states should be disjoint, for it is not possible to have the same (stack, lookahead) pair for both actions, and thus we are also able to discriminate, in this case upon the lookahead.

But it is clear that this solution is unnecessarily costly. We could combine both automata into only one, in principle able to recognize both types of stacks. In this automaton, each state knows—from its sets of situations— whether the two competing actions, or just only one of them, are still possible, according to the stack portion read so far. Furthermore, since our objective is to discriminate, we do not need to completely build a recognizing automaton, but only those states where the actions are not unique.

In our example, according to the initial sets of situations for each automaton, reducing rule 5 is only possible when lookahead is "$\dashv$, +", or ")", and shift is only

possible for lookahead values "$\vdash$", "$+$", "$*$", "$)$", "$($", or "a". Thus, we can decide that, if lookahead is "$\dashv$" we have to reduce using rule 5, and if it is "$\vdash$", "$*$", "$($", or "a" we have to shift. Only from lookaheads "$+$" and "$)$" we cannot decide what action to perform. But if we extract the situations with precisely these lookahead values from both automata, we obtain the following:

- Automaton for reducing rule 5:

$$\begin{array}{lll} R \rightarrow \cdot & (5') & +')' \\ T \rightarrow F \cdot R & (5) & +')' \\ R \rightarrow * F \cdot R & (5) & +')' \end{array}$$

- Automaton for shifting:

$$\begin{array}{lll} E \rightarrow E \cdot + T & (0) & + \\ F \rightarrow (E \cdot) & (0) &) \end{array}$$

We see that, for those cases in consideration, just by looking now at the top-of-stack symbol, we can decide: if it is "F" we have to reduce rule 5, and if "E" we have to shift.

If we were left with some situations for which we could not yet decide, we should continue state construction—considering only the remaining competing situations—until finally obtaining a decision. This is the case in our example with the two competing reductions for rules 4 and 6, where we only have a transition from the initial state with "R", and then only a transition with "F" to the following state:

$$\begin{array}{lll} T \rightarrow \cdot FR & (4') & \dashv' +')' \\ R \rightarrow * \cdot FR & (6') & \dashv' +')' \\ E \rightarrow E + \cdot T & (4) & \dashv' +')' \\ E \rightarrow \cdot T & (4) & \dashv' +')' \\ S \rightarrow \vdash \cdot E & (4) & \dashv' + \\ E \rightarrow \cdot E + T & (4) & + \\ F \rightarrow (\cdot E) & (4) &)+ \end{array}$$

We can now derive the known result that, when the next stack symbol is "$*$" we have to reduce according to rule 6, or else for "$+$", "$\vdash$", or "$($"—the only remaining legal possibilities—we have to reduce according to rule 4.

2.4 A full discriminating automaton

From the previous sections it should be clear now how to build the full discriminating automaton for all the possible parsing actions for any legal stack-plus-lookahead configuration. We just have to join into a single initial state all the situations for all the parsing actions, and perform their closure computation. At any state, we discriminate first according to the lookahead symbol, and for the remaining actions we try to discriminate according to the next stack symbol. For those remaining situations for which we are at this state unable to discriminate their actions—because each legal lookahead plus the stack read so far are still compatible with several actions—, we build the transitions to other states through the next stack symbol, and repeat the procedure for the new states.

If the grammar is not LR(1), the construction procedure will find some final (bottom-of-stack) state, where no discrimination is possible upon the lookahead symbol.

If we apply this building procedure to our example grammar, we obtain the actions and transitions of Fig. 1, where "LA" indicates actions upon the current lookahead symbol, and "ST" indicates actions or transitions upon the next stack symbol. For each state, the competing actions are indicated between brackets. Those states with only one possible action are needed to check the presence of the full right-part on top of the stack.

2.5 Parsing with the automaton

For parsing according to an LR(1) grammar, when the parser has to decide the next parsing action to perform, it begins every time using the automaton from its initial state, first using the lookahead to possibly decide the action, or else reading next symbol down in the stack to decide or to make a state transition.

To illustrate parsing with the automaton, let us consider again the rightmost sentential form

$$\vdash F * (T \underline{+} a) + a \dashv$$

Since the current lookahead "+" is not useful for deciding at the initial state, we take a transition upon the top-of-stack symbol "T" to state 3, where the next stack symbol "(" indicates a reduction with rule 3. Now we have the configuration

$$\vdash F * (E \underline{+}$$

and begin again from state 1, where we take a transition upon "E" to state 2, where a shift is indicated by the current lookahead "+". The next lookahead is "a", which indicates in initial state a shift. Now the initial state indicates a reduction with rule 8 upon the top-of-stack "a". We now have the configuration

$$\vdash F * (E + F \underline{)}$$

and we take a transition from initial state upon "F" to state 6, where the current lookahead ")" indicates reduction of "ε" to "R" according to rule 5. The reader can check that parsing continues in this manner until a reduction with rule 1, which always indicates a successful end of parsing, if the current lookahead is "$\dashv$".

2.6 Parsing erroneous inputs

Let us consider an erroneous input similar to the previous one, where the programmer has forgotten to include the right parenthesis. With an erroneous form like

$$\vdash F * (T \underline{+} a + a \dashv$$

the parser would continue its reduction process until the following configuration

$$\vdash F * (E \underline{\dashv}$$

State 1 [0 1 2 3 4 5 6 7 8]
$LA(\vdash)$ = Shift
$LA(()$ = Shift
$LA(a)$ = Shift
$ST(a)$ = Reduce 8
$ST(E)$ = Go To 2
$ST(T)$ = Go To 3
$ST(R)$ = Go To 4
$ST())$ = Go To 5
$ST(F)$ = Go To 6

State 2 [0 1]
$LA(+)$ = Shift
$LA())$ = Shift
$ST(\vdash)$ = Reduce 1

State 3 [2 3]
$ST(\vdash)$ = Reduce 3
$ST(()$ = Reduce 3
$ST(+)$ = Go To 7

State 4 [4 6]
$ST(F)$ = Go To 8

State 5 [7]
$ST(E)$ = Go To 9

State 6 [0 5]
$LA(\dashv)$ = Reduce 5
$LA(+)$ = Reduce 5
$LA())$ = Reduce 5
$LA(*)$ = Shift

State 7 [2]
$ST(E)$ = Reduce 2

State 8 [4 6]
$ST(*)$ = Reduce 6
$ST(+)$ = Reduce 4
$ST(\vdash)$ = Reduce 4
$ST(()$ = Reduce 4

State 9 [7]
$ST(()$ = Reduce 7

Fig. 1. Actions and transitions of the automaton

At this point, the parser goes to state 2 upon "E", and then signals an error, since the lookahead is not "+" nor ")", and the only remaining legal possibility for next stack symbol is to be "⊢", but it actually is "(".

In general, since our parser is not intended to check the correctness of the full stack, errors might be detected later that in the best possible case, i.e. the correct prefix property can not always be assured. More reductions might be done, and several symbols might be shifted beyond the first possible point of error detection. Anyway, reductions are always legal in the sense that the right-parts are always verified, and for this reason errors are always detected.

But, what is possibly more important, since in a sense our automaton performs somehow like precedence methods, the well known error recovery procedures for these methods can be easily adapted[7]. It can also be of interest to investigate how new powerful methods for error processing specially adapted to bounded context grammars[13, 2] could be applied.

3 Validity of the method

Some considerations concerning the validity of the proposed method may be in order, specially a justification of its LR(1) parsing power.

During a bottom-up parsing for an LR(1) grammar, the next parsing action is unique for each legal stack-plus-lookahead configuration:

- To shift. This should correspond to rules such as $A \rightarrow \alpha a \beta$, where a correponds to the current lookahead. This is represented by the initial state's situations $A \rightarrow \alpha \cdot a\beta(0)a$.
- To reduce a rule such as $A \rightarrow \alpha(i)$. This can only happen when current lookahead $a \in \mathit{Follow}(A)$. This is represented by the initial state's situation $A \rightarrow \alpha \cdot (i')a'$.

As we have seen for the example grammar, for each such possible parsing action we can build a top-to-bottom recognizing automaton for all its compatible legal stacks, i.e. those stacks permitting some rightmost sentential form in accordance with the action. It is important to note that situations contains all the basic information to build the rest of the automaton that recognize the rest (from the current point of exploration to the bottom) of compatible legal stacks, for as we have seen situations allow to infer the new situations needed to build the next transitions and states.

We have also seen that a top-to-bottom recognizing automaton for precisely the set of all legal stacks can be built, by merging all these initial-situation derived automata. It is cleat that this automaton can always indicate the next parsing action after having read the whole stack, since then the lookahead symbol will always suffice to discriminate amongst the would-be different actions—LR(1) property.

But the key point is that in almost all practical LR(1) grammars a small section of the full recognizing automaton is enough to decide the parsing action. In fact, the construction procedure for the discriminating automaton *begins* building such a recognizing automaton, but only generates its useful discriminating section—preserving its full right-part section. So, in a state, when there is a situation whose lookahead uniquely determines the parsing action, this situation's subsequent situations are not generated. The same happens for the rest of state situations, when their following

stack symbol uniquely determines the parsing action. This mechamism dramatically prunes the recognizing automaton, while preserving a parsing power of LR(1), since the same discriminating results are obtained.

Finally, if the grammar is not LR(1), there is a stack-plus-lookahead configuration with at least two "legal" different parsing actions. In this case, automaton states might be built until a (bottom of stack) "final" state, where those different actions are present but lookahead is useless for discrimination. This type of state is easily detected.

4 Discussion from some experimental results

An experimental parser generator has been built according to the indicated method. Practical results are in accordance with the foreseen parsing power and small automata sizes. Since no classic LR(1) parser generator is yet available to us, comparisons have been done with Yacc[9], the most widely used parser generator. Reverse automata number of states for different size grammars range from −30% to −40% that of the corresponding Yacc-generated automata. Some results are given in the following table, including those for the biggest grammar yet tested, an independently-obtained[1] full C language grammar for Yacc, and its modified grammar to make it exactly LALR(1). Although current comparisons are restricted to LALR(1) grammars, future comparisons with classic LR(1) generators—the same parsing power—are clearly anticipated to be very favourable.

		Automaton no. of states	
Grammar	*No. of rules*	*reverse LR(1)*	*Yacc*
Minilanguage	94	118	173
LALR(1) C language	235	284	439
Yacc C language	214	n/a	367

A reason for the improvement in size may be that classic LR(0) combines parsing action decision with full checking of the correctness of the whole stack, with LR(1) considering the lookahead symbol, which results in a great increase in size. LALR(1) storage requirements are a little bigger than LR(0). On the contrary, the reverse automaton construction stops as soon as discrimination is guaranteed, and further simplifications, well suited to large programming language grammars, are allowed. An additional reason may be that it begins the discrimination process from the point where the information is usually more useful, as the existence of several precedence methods demonstrates, and thus quickly separates the rest of discriminations into simpler tasks.

In theory, parsing efficiency compares disfavourably with classic parsing methods in the sense that they do not search in the stack; in some methods because the necessary information is already stored in a "state". However, the reverse automaton deeply searches in the stack—relatively short for programming language programs—only in the worst theoretical cases. Even for those worst cases, that should not happen frequently: in the case of a hard left context configuration, the resulting

[1] From the `comp.compilers` Usenet newsgroup database.

nonterminal will normally be different for the alternative actions, being enough for subsequently not to search so deeply. Although statistical comparisons with other methods are not yet available, analysis of generated automata reveals the following:

- Shift actions—the most frequent ones—are discriminated after very few states (typically between 1 and 3). Thus, for these most frequent actions the parser reads very few symbols from the stack.
- The number of states not in the full right-parts section (i.e. those actually needed to solve conflicting reductions) only represent about one third of the automaton. Thus conflicting cases—those needing extra stack exploration—represent a relatively small part of cases, most of them amongst competing reductions. In terms of size and parsing time, this can be considered the extra cost—very small in relation with the improvement in parsing power—in comparison with precedence methods, for these also have to recognize the presence of the rule right-part on top of the stack.
- Loops are very scarce. This again suggests that very rarely the automaton needs to search relatively deep in the stack.

We can conclude that all the data available suggest that parsing efficiency should not be significatively damaged in practice. By contrast, other powerful methods, like classic LR(1), build much bigger automata, whose big tables are usually compacted, resulting in a diminished access efficiency. This should be compared with the cost of analyzing a normally small part of the stack near its top. Finally, the order in which the possible parsing actions are checked may be improved.

Errors are always detected, but sometimes later than in the best possible case—although compaction of tables in other methods may also negatively affect this feature. Nevertheless, error-free inputs are always well parsed, and reductions are always correct. Errors may also introduce a deeper search in the stack to be detected. Overall, the additional overhead for erroneous inputs does not seem to be very significant, considering that modern parsers should automatically treat input errors to continue parsing, and thus analysis and modification of the area in error must be performed. The relatively little importance of this extra cost, combined with an acceptable performance and adequacy for known error recovery methods, results in a good compromise.

Construction efficiency should be much better than classic LR(1), from simple size comparisons. Anyway, more extensive tests and developments are under way in this and in other aspects to obtain more significant results.

Finally, a great increase in size appears when trying to improve classic LR(1). On the contrary, the parsing power of the reverse automaton seems to be possible to be improved, e.g. to LR(2), at low cost.

5 Conclusions

A method for generating parsers for LR(1) grammars, that builds a deterministic finite-state automaton that is able to determine, by reading the parsing stack from its top in combination with the lookahead symbol, the next parsing action, has been presented through the use of an example grammar.

Important reductions in automata size are obtained from experimental results in comparison with classic LR methods, due mainly to the facts that the most useful information is normally near the top of the stack, and that the building procedure stops as early as the next parsing action can be uniquely determined. Its theoretical drawbacks, later error detection and diminished parsing efficiency, are found to be of little practical importance. Further experimentation and improvements should indicate whether the method has to be considered of practical interest to compete with current LR methods. Anyway, its actual applicability for big LR(1) grammars should be pointed out.

From the theoretical point of view, we note that a powerful parsing method has been obtained by extrapolating the ideas from the old family of precedence methods, resulting in a new approach for parsing that deserves further research.

Acknowledgments. This work owes a lot to discussions with Olivier Lecarme and Jacques Farré, and to the friendly environment at the laboratory. It resulted from an initial proposal for extending a mixed strategy precedence parser generator to empty rules and better context exploration. The idea that the lookahead symbol should not be used first is due to Olivier.

I wish to especially acknowledge my beloved wife, María del Mar.

References

1. A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling.* Prentice-Hall, 1973.
2. G. V. Cormack. An LR substring parser for noncorrecting syntax error recovery. *ACM SIGPLAN Notices*, 24(7):161–169, 7 1989.
3. F. DeRemer. *Practical Translators for LR(k) Languages.* PhD thesis, M.I.T., Cambridge, Massachusetts, U.S.A., 1969.
4. R. W. Floyd. Bounded context syntactic analysis. *Comm. ACM*, 7(2):62–67, 1964.
5. J. Fortes Gálvez. A discriminating reverse automaton for LR(1) parsing. Research Report 91-23, Laboratoire I3S, Bât. 4, 250 Avenue Albert Einstein, F-06560 Valbonne, France, October 1991. A revised version has been submitted for publication in *Information Processing Letters.*
6. R. M. Graham. Bounded context translation. In *AFIPS Spring Joint Computer Conference*, pages 184–205, 1964.
7. S. L. Graham and S. P. Rhodes. Practical syntactic error recovery. *Comm. ACM*, 18(11):639–650, 1975.
8. J. D. Ichbiah and S. P. Morse. A technique for generating almost optimal Floyd-Evans productions for precedence grammars. *Comm. ACM*, 13(8):501–508, 1970.
9. S. C. Johnson. Yacc—yet another compiler compiler. Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, New Jersey, U.S.A., 1975.
10. D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.
11. W. M. McKeeman, J. J. Horning, and D. B. Wortman. *A Compiler Generator.* Prentice-Hall, 1970.
12. C. Pair. Trees, pushdown stores and compilation. *RFTI—Chiffres*, 7(3):199–216, 1964.
13. H. Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, 7 1985.

14. S. Sippu and E. Soisalon-Soininen. *Parsing Theory.* Springer-Verlag, 1990.
15. N. Wirth and H. Weber. EULER: A generalization of ALGOL and its formal definition. Parts I and II. *Comm. ACM*, 9(1):13–23 and 89–99, 1966.