A Bounded Graph-Connect Construction for LR-regular Parsers

Jacques Farré[1] and José Fortes Gálvez[2,1]

[1] Laboratoire I3S, CNRS and Université de Nice - Sophia Antipolis
[2] Depart. de Informática y Sistemas, Universidad de Las Palmas de Gran Canaria

Abstract. Parser generation tools currently used for computer language analysis rely on user wisdom in order to resolve grammar conflicts. Here practical LR(0)-based parser generation is introduced, with automatic conflict resolution by potentially-unbounded lookahead exploration. The underlying LR(0)-automaton item dependence graph is used for lookahead DFA construction. A bounded graph-connect technique overcomes the difficulties of previous approaches with empty rules, and compact coding allows to precisely resume right-hand contexts. Resulting parsers are deterministic and linear, and accept a large class of LR-regular grammars including LALR(k). Their construction is formally introduced, shown to be decidable, and illustrated by a detailed example.

1 Introduction

Grammars for many computer languages (programming languages, description languages ...) are neither LL(1) nor LALR(1). Unfortunately, most available parser generators, without user help, are restricted to these grammar classes. And the user is confronted with adapting the grammar, if not the language, to the restrictions imposed by the tool. Unnatural design, which may not adapt well to the application, results from these restrictions, e.g., the discussion on a Java LALR grammar in [8]. Moreover, this adaptation often requires user expertise and detailed analysis. The need for more powerful parsers exists, as shown by the popularity of tools proposing different means to overcome conflicts. A first choice is to use generators that propose to resolve conflicts by means of predicates [9, 14]. These are apparently-simple semantic checks or lookahead explorations specified "ad hoc" by the user. But resorting to predicates is intrinsically insecure, because parser generators cannot detect design errors, and may produce incorrect parsers without user notice. Moreover, ill-designed syntactic predicates may involve heavy backtracking, resulting in inefficient parsers.

A second choice is to use parsers for unrestricted context-free grammars [6, 12,11]. In particular *Generalized LR* [18], which has efficient implementations [1] and has been successfully used outside the scope of natural language processing (e.g., [19]). Unfortunately, these generators cannot always warn about grammar ambiguity. As a result, the user may be confronted with an unexpected forest at parsing time. Although GLR has been practically found near-linear in many

R. Wilhelm (Ed.): CC 2001, LNCS 2027, pp. 244–258, 2001.

cases, nondeterminism occurs for each conflict, and linearity is not guaranteed, even for unambiguous grammars. Finally, nondeterminism compels to undo or to defer semantic actions.

We think that, between the above choices, a tool offering automatic conflict resolution through potentially-unbounded lookahead exploration should be useful. It would allow to produce deterministic and reliable parsers for a large class of unambiguous grammars, without relying on user help or expertise.

In LR-regular (LRR) parsing [5], an LR conflict can be resolved if there exist disjoint regular languages covering the right-hand contexts of every action in conflict. The LRR class is theoretically interesting, since it allows to build efficient parsers using deterministic finite-state automata (DFA) for lookahead exploration. In particular, it has been established that LRR parsing is linear [10], what is not ensured by GLR or parsers with predicates.

Unfortunately, full LRR parser generators cannot be constructed because it is undecidable whether a grammar is LRR [17]. Nevertheless, several attempts have been done to develop techniques for some subset of LRR, but have failed to produce sufficiently practical parser generators. At most, they guarantee LALR(k) only for grammars without ε-rules, what nowadays is considered unacceptable.

In this paper, we propose a bounded graph-connect technique for building LRR parsers. This technique is inspired by noncanonical discriminating-reverse (NDR) parsers [7], which actually accept a wider class of grammars not restricted within LRR. However, the method shown here naturally produces canonical and correct-prefix parsers (what is not ensured by NDR), which are usually considered desirable properties.

1.1 Related Work

The first attempt to develop parsers for a subset of LRR grammars is due to Baker [2], with the XLR method. In XLR, the LR automaton is used as the basis for the lookahead automata construction, with the addition of ε-arcs that allow to resume exploration after reduction states. However, the lack of precise right-hand context recovery makes XLR(k) a proper subclass of LALR(k).

The methods R(s)LR by Boullier [4], and LAR(s) by Bermudez and Schimpf [3], were independently developed with similar results. Their approach can be described as using a bounded memory of s LR(0) automaton states in order to improve right-hand context recovery. Thus, both methods are more powerful than XLR, but they both have problems with ε productions. As they show, R(s)LR(0) accepts all ε-free LALR(s) grammars, but for any s there exist LALR(1) grammars that are not LAR(s).

Seité studied automatic LALR(1) conflict resolution [15]. His method, while not more powerful than Boullier's in the lookahead exploration phase, introduces an additional phase which scans some stack prefix when the lookahead exploration has not been able to resolve the conflict. This extension, which increases the accepted class with some non-LR, LRR grammars, is quite independent of the lookahead exploration technique used. Although it can be applied to our method, we shall not consider it here.

1.2 Introduction to Our Approach

In our approach, right-hand context computation is made much more precise than previous methods, thanks to several innovations:

- Instead of automaton state transitions, we follow the underlying kernel-item transition graph of the LR(0) automaton. Using kernel items instead of full state item-set allows a simpler and more efficient construction.
- Instead of keeping track of s states for the right-hand context to recover, we keep track of at most h *path subgraphs* of the underlying item graph. Each such subgraph is coded by its extreme items, and the underlying graph is in turn used to follow precisely each possible path.
- An ε-skip move, combined with the above bounded graph-connect technique, allows to skip and precisely resume exploration on the right of ε-deriving nonterminals.

Our construction follows precisely right-hand contexts as far as no more than h path subgraphs are needed. Since at most a new subgraph is added for each explored terminal, all LALR(h) grammars are accepted. Moreover, for most practical grammars, an effective lookahead length well beyond h (in fact, usually unbounded, and thus allowing grammars not in LR) should be actually obtained.

In previous methods, since transitions on ε-deriving nonterminals add states to the (bounded) state memory, there exist LALR(1) grammars whose conflicts cannot be resolved.

2 Context Recovery and Subgraph Connections

In this section, we shall first describe the kernel-item underlying graph, and then the different aspects of our approach for computing right-hand contexts.

Usual notational conventions, e.g., [16], will be generally followed. The original grammar $\mathcal{G}$ is augmented with $P' = \{S' \to S\dashv\} \cup P$. Productions in P' are numbered from 1 to $|P'|$, what is sometimes noted as $A \xrightarrow{i} \alpha$.

2.1 LR(0) Construction

Let us first recall the LR(0) automaton construction where each state q corresponds to a set K_q of *kernel* items. For initial state q_0 we have $K_{q_0} = \{[S' \to \cdot S\dashv]\}$. The transition function Δ can be defined as follows:

$$\Delta(K_q, X) = \{[A \to \alpha X \cdot \beta] \,|\, [A \to \alpha \cdot X\beta] \in \mathcal{C}(K_q)\},$$

where $\mathcal{C}$ is the *closure* operation, which can be defined as

$$\mathcal{C}(K) = K \cup \{[B \to \cdot\gamma] \,|\, [A \to \alpha \cdot \beta] \in K, \beta \underset{\text{rm}}{\Rightarrow}^* Bx \Rightarrow \gamma x\}.$$

A kernel item ι indicates the parsing actions "shift" if there is some $[A \to \alpha \cdot a\beta]$ in $\mathcal{C}(\{\iota\})$ and "reduce j" for each $[B \xrightarrow{j} \gamma \cdot]$ in $\mathcal{C}(\{\iota\})$. A *conflict*-state item set indicates a set of two or more different actions. For these states, our method builds a *lookahead* automaton, by using the kernel-item graph as its basic reference.

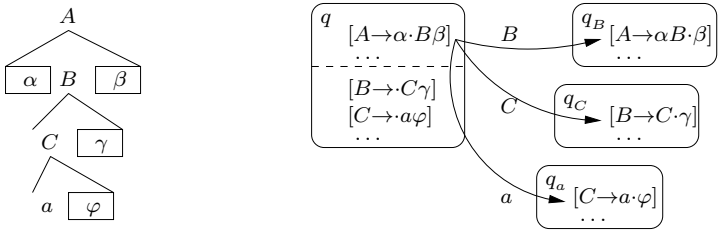

Fig. 1. Illustration of δ function

2.2 The Underlying LR(0) Kernel-Item Graph

In the LR(0) automaton underlying graph, each node $\nu = [A{\to}\alpha{\cdot}\beta]_q$ corresponds to some kernel item $[A{\to}\alpha{\cdot}\beta]$ in some set K_q, i.e., the same item in different state sets will correspond to different nodes. (Single) transitions are defined on V, as illustrated by Fig. 1:

$$\delta(\nu, X) = \{[A{\to}\alpha X{\cdot}\beta]_{q'} \mid \Delta(K_q, X) = K_{q'}, \nu = [A{\to}\alpha{\cdot}X\beta]_q\}$$
$$\cup\, \{[C{\to}X{\cdot}\gamma]_{q'} \mid \Delta(K_q, X) = K_{q'}, \nu = [A{\to}\alpha{\cdot}B\beta]_q, B \underset{\text{rm}}{\Rightarrow}^* Cx \Rightarrow X\gamma x\}.$$

In the first subset, the dot moves along the right-hand side of a rule until its right-hand end, where no more transitions are possible. The second subset may be understood as "transitions through" nonkernel items resulting by closure from some kernel item in the originating state q to some kernel item in the destination state q'.

We extend δ for transitions on strings in V^*:

$$\delta^*(\nu, \varepsilon) = \{\nu\}, \qquad \delta^*(\nu, X\alpha) = \bigcup_{\nu' \in\, \delta(\nu, X)} \delta^*(\nu', \alpha).$$

We shall usually write $\nu \xrightarrow{\alpha} \nu'$ instead of $\nu' \in \delta^*(\nu, \alpha)$.

2.3 Simple Context Recovery

Item graph transitions on terminals determine the lookahead DFA transitions. The starting points, for each LR(0) conflict, are kernel items in the conflict state. Suppose that, in Fig. 2, q is a state with a shift-reduce conflict. For the shift action in conflict, the transition on a will be followed initially from node $[A{\to}\alpha{\cdot}B\beta]_q$ to node $[C{\to}a{\cdot}w]_{q_a}$, and then, if necessary, successive transitions on symbols in w.

A first aspect to consider is how to precisely recover the right-hand contexts of a left-hand side nonterminal when its corresponding rule's right-hand side has been completely traversed, e.g., in Fig. 2, when $\nu = [C{\to}aw{\cdot}]_{q_{aw}}$ is reached, graph traversal must resume with $\nu' = [D{\to}C{\cdot}\gamma]_{q_C}$, but not with nodes for which there exist transitions on C from some ν'' in $q' \neq q$. For this purpose,

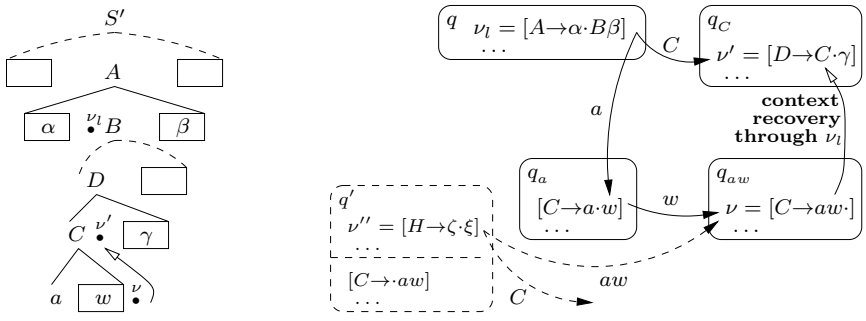

Fig. 2. Illustration of simple context-recovery

we use *node pairs* (ν_l, ν), where ν represents the *current position* in the item graph, and ν_l its *left-hand reference* node, i.e., the kernel-item node from which right-hand side traversals start. While graph traversal proceeds along some rule's right-hand side, the corresponding ν_l remains unchanged. Figure 2 shows nodes corresponding to pairs (ν_l, ν) and (ν_l, ν') for traversal of aw, i.e., $\nu_l \xrightarrow{aw} \nu$ and $\nu_l \xrightarrow{C} \nu'$.

2.4 Subgraph Connections

Another aspect arises from lookahead transitions corresponding to further descents into possible derivation subtrees after partial right-hand side exploration. Figure 3 illustrates this case: on a transition on b, the current position moves down the subtree with root C from $\nu = [B \to \gamma \cdot C\varphi]_{q'}$ to $\nu' = [G \to b \cdot \zeta]_{q'_b}$. The current $\nu_l = [A \to \alpha \cdot B\beta]_q$ cannot be used as left-hand reference for the nodes corresponding to positions in this subtree. Thus, ν becomes the reference node ν'_l for the subtree walk. In order to recover right-hand context after subtree of root C has been explored, a so-called *subgraph connection* is needed. It will permit, after switching to the new subgraph, to retrieve the previous one if needed. For this purpose, we keep sequences of pairs

$$\kappa = (\nu_c^{(1)}, \nu_t^{(1)}) \cdots (\nu_c^{(m)}, \nu_t^{(m)}), m \geq 1.$$

Last pair $(\nu_c^{(m)}, \nu_t^{(m)})$ will also be noted (ν_l, ν), since $\nu_t^{(m)}$ represents the current position and $\nu_c^{(m)}$ its left-hand reference node. In Fig. 3, $\kappa = (\nu_l, \nu)(\nu, \nu')$ after the transition on b. We shall see in next section that in some cases $\nu_t^{(i)} \neq \nu_c^{(i+1)}$.

Unfortunately, the length of such sequences κ is usually unbounded, since (indirectly) recursive rules produce loops in the item graph. We ensure a finite construction by bounding $|\kappa|$ within some constant h[1], in such a way that, when

[1] Its value could be set by the user, for instance, although $h = 1$ should be enough in most practical cases.

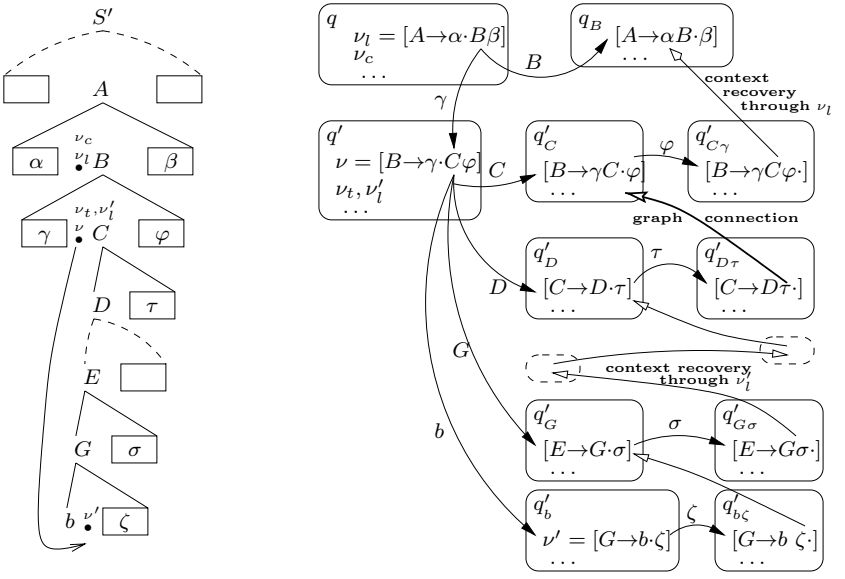

Fig. 3. Illustration of subgraph connection

appending another pair to a sequence of length h, the sequence's first element is lost. Thus, exact right-hand context cannot be computed if the lookahead automaton construction requires to resume graph transitions involving lost reference nodes.

2.5 Skipping ε-Deriving Nonterminals

Let us finally consider ε-productions. Differently from previous methods, the item graph allows to follow transitions on ε-deriving nonterminals without lengthening the graph sequence κ. If current position is $[B{\to}\gamma{\cdot}\rho C\gamma']_{q_\gamma}$, as in Fig. 4 —dashed boxes represent sequences of ε-deriving nonterminals—, and next exploration symbol can be $a \in \mathrm{First}(C)$, a graph connection is performed after skipping the sequence η of ε-deriving nonterminals, and the pair (ν'_l, ν') is added to κ, with $\nu_t \xrightarrow{\eta} \nu'_l \xrightarrow{a} \nu'$. Thus, in general, $\nu_t^{(i)} \xrightarrow{\eta} \nu_c^{(i+1)}, \eta \Rightarrow^* \varepsilon$.

Context is recovered in principle as described in previous sections. However, we also need in general to skip sequences of ε-deriving nonterminals for transitions corresponding to up-right moves in the possible derivation trees, in order to reach a position on the left of some X such that $X \underset{\mathrm{rm}}{\Rightarrow}{}^* bx$, as shown by the upward ε-skip path in Fig. 4. For this purpose, amongst all paths allowed by the LR(0)-automaton graph, we should only consider those followed by the downward ε-skip from ν_t, i.e., those leading from ν_t to the current left-hand reference (ν'_l in Fig. 4.)

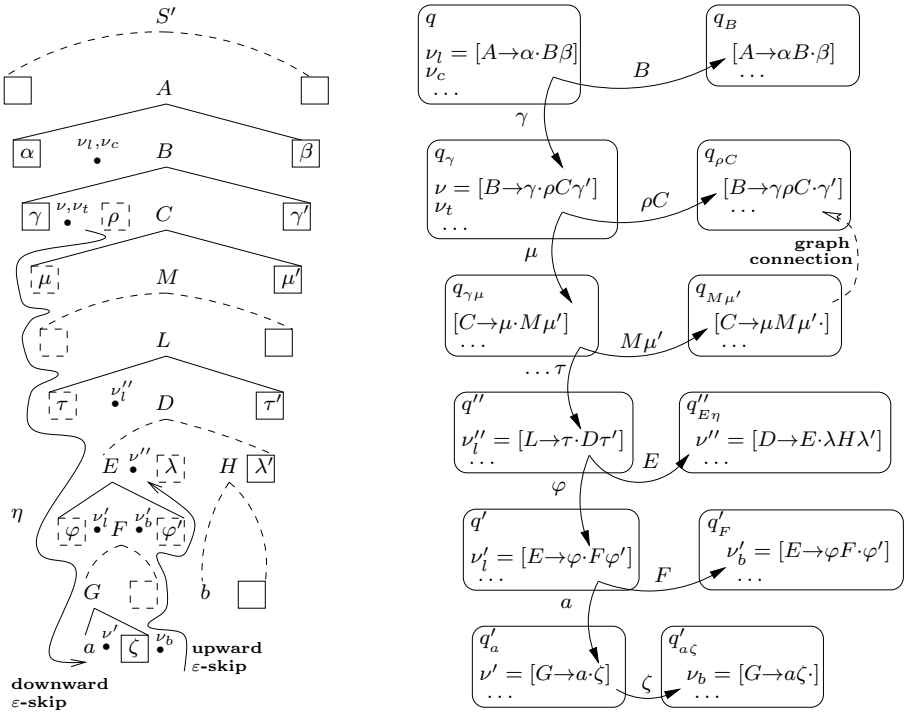

Fig. 4. Illustration of skipping ε-deriving nonterminals

2.6 General Context Recovery

Let us now summarize the above considerations in order to present the general case for single (upward) context-recovery steps from current sequence $\kappa(\nu_l, \nu)$.

In general, we have $\nu_l \xrightarrow{\beta} \nu = [A \to \alpha\beta \cdot \gamma]_q$. If $\gamma \Rightarrow^* \varepsilon$, we have to consider an upward step to resume exploration for the (precise, if possible) right-hand context of A, i.e., we have to compute those sequences $\kappa'(\nu'_l, \nu')$ such that $\nu'_l \xrightarrow{\varphi A} \nu' = [B \to \varphi A \cdot \psi]_{q'}$ and which are compatible with current sequence. We distinguish the following cases:

1. ν' is lower than ν_l in the derivation tree[2] (leftmost tree in Fig. 5). Only ν changes, i.e., $\nu'_l = \nu_l$.
2. Otherwise, there are two subcases:
   a) κ is empty, i.e., current sequence contains only one pair (ν_l, ν) (middle tree[3] in Fig. 5 with no ν_t). If no graph connection has been performed yet,

[2] In this case, $\alpha = \varepsilon$.

[3] The possible positions of the different nodes are shown. For instance, in this tree, lower ν_l passes to higher ν'_l when $\varphi = \varepsilon$, and higher ν_l —which is only possible when $\varphi \neq \varepsilon$— does not move.

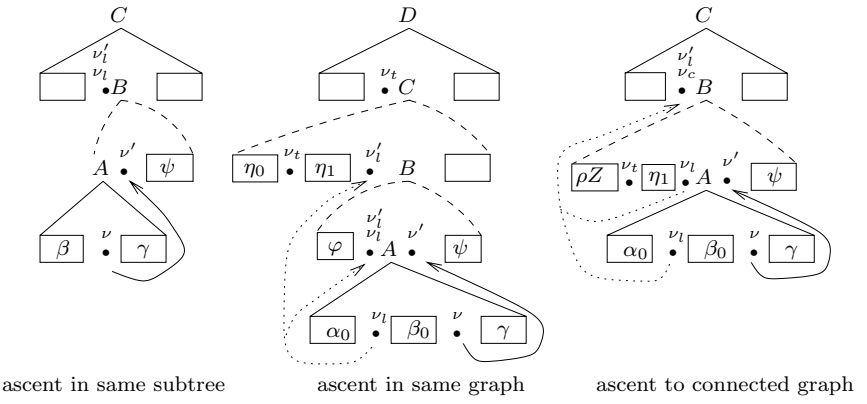

Fig. 5. Illustration of single context-recovery steps

or, if performed and subsequently resumed, no truncation (to h pairs) has taken place, then all (ν_l', ν') such that $\nu_l' \xrightarrow{\alpha} \nu_l$ and $\nu_l' \xrightarrow{A} \nu'$ are exact. However, in case of truncation some of such (ν_l', ν') may be in excess, resulting in precision loss.

b) Otherwise current subgraph is connected[4], i.e., $\kappa = \kappa_1(\nu_c, \nu_t)$. There are two possibilities:
   i. ν' is lower than ν_t (middle tree in Fig. 5). According to the discussion in Sect. 2.5, only those ν_l' on a downward ε-skip path η from ν_t are precise.
   ii. Otherwise ν' can only be at the same level as ν_t (rightmost tree in Fig. 5). The connected subgraph is resumed, i.e., last pair is removed from current sequence and ν_t is replaced by ν'.

The following function performs such single context-recovery steps:

$$\hat{\theta}(\kappa(\nu_l, \nu)) = \begin{cases} \{\kappa_1(\nu_c, \nu') \mid \nu_t \xrightarrow{\eta} \nu_l', \nu' = [B \to \rho Z \eta A \cdot \psi]_q, \eta \Rightarrow^* \varepsilon, (\nu_l, \nu) \upuparrows (\nu_l', \nu')\} \\ \quad \cup \{\kappa(\nu_l', \nu') \mid \nu_t \xrightarrow{\eta\varphi} \nu_l', \nu' = [B \to \varphi A \cdot \psi]_q, \eta\varphi \Rightarrow^* \varepsilon, (\nu_l, \nu) \upuparrows (\nu_l', \nu')\} & \text{if } \kappa = \kappa_1(\nu_c, \nu_t) \\ \{(\nu_l', \nu') \mid (\nu_l, \nu) \upuparrows (\nu_l', \nu')\} & \text{otherwise,} \end{cases}$$

such that $(\nu_l, \nu) \upuparrows (\nu_l', \nu')$ iff $\nu_l' \xrightarrow{\alpha} \nu_l \xrightarrow{\beta} \nu = [A \to \alpha\beta \cdot \gamma]_q$, $\nu_l' \xrightarrow{A} \nu'$, and $\gamma \Rightarrow^* \varepsilon$.

Its closure, which allows any number of context recovering steps, is defined as the minimal set such that $\hat{\theta}^*(\kappa) = \{\kappa\} \cup \{\kappa'' \mid \kappa'' \in \hat{\theta}(\kappa'), \kappa' \in \hat{\theta}^*(\kappa)\}$.

3 Construction of Lookahead Automata

States r in the lookahead DFA correspond to sets J_r of *lookahead items*. A lookahead item $\mu = [j, \kappa]$ will describe a current sequence (i.e., current position

[4] In this case, $\alpha \Rightarrow^* \varepsilon$, since we always have $\nu_t \xrightarrow{\eta} \nu_l, \eta \Rightarrow^* \varepsilon$.

and its recovery context) κ for some action j in conflict. By convention, $j = 0$ for a shift action, and $j = i$ for a reduction according to $A \xrightarrow{i} \alpha$.

3.1 Transitions from Lookahead States

The following function computes the next set of lookahead items after some terminal-symbol transition:

$$\begin{aligned}&\Theta_h(J_r, a) = \\ &\quad \hat{\Theta}(\{[j, \kappa(\nu_l, \nu')] \mid [j, \kappa(\nu_l, \nu)] \in J_r, \nu \xrightarrow{\beta Y} \nu' = [A \to \alpha X \beta Y \cdot \gamma]_q, \beta \Rightarrow^* \varepsilon, Y \Rightarrow^* a\} \\ &\qquad \cup \{[j, \kappa(\nu_c, \nu_t)(\nu_l, \nu) : h] \mid [j, \kappa(\nu_c, \nu_t)] \in J_r, \nu_t \xrightarrow{\eta} \nu_l \xrightarrow{\beta Y} \nu = [A \to \beta Y \cdot \gamma]_q, \\ &\qquad\quad \eta\beta \Rightarrow^* \varepsilon, Y \Rightarrow^* a, \gamma \Rightarrow^* bx\}).\end{aligned}$$

The first subset corresponds to the case in which we are simply moving along the right-hand side of some rule, or we descend in a subtree to immediately return to the current level. To avoid a graph connection in the latter situation is of practical interest[5].

The second subset corresponds to a true descent in the derivation tree, i.e., exploration will continue in it. Thus, a graph connection is performed, in order to correctly resume after the subtree exploration.

Finally, function $\hat{\Theta}$ performs all necessary ε-skip upward context-recovery including removal of useless items[6]:

$$\hat{\Theta}(J_r) = \{[j, \kappa'(\nu_l, \nu)] \mid \kappa'(\nu_l, \nu) \in \hat{\theta}^*(\kappa), [j, \kappa] \in J_r, \nu = [A \to \alpha \cdot \beta]_q, \beta \Rightarrow^* ax\}.$$

3.2 Initial LR(0)-Conflict Lookahead Item Sets

Let r_0^q be the initial lookahead state associated with some conflict-state q. Its associated set of lookahead items can be computed as follows.

$$J_{r_0^q} = \{[0, (\nu, \nu)] \mid K_q \ni \nu \xrightarrow{a} \nu'\} \cup \hat{\Theta}(\{[i, (\nu_l, \nu')] \mid A \xrightarrow{i} \alpha, \nu_l \xrightarrow{A} \nu', \nu_l \xrightarrow{\alpha} \nu \in K_q\}).$$

As for the definition of the transition function, an upward ε-skip is applied.

3.3 Inadequacy Condition

A grammar $\mathcal{G}$ is inadequate iff, for some lookahead state r,

$$\exists\, [j, \kappa], [j', \kappa] \in J_r,\ j \neq j'.$$

[5] In order to make grammars more readable, productions such as *type-name* → IDENT and *var-name* → IDENT are frequently used. Rejecting such grammars because the graph connection looses precise context would be unfortunate.

[6] They are now useless for subsequent transitions if $\text{First}(\beta) = \{\varepsilon\}$.

Since two lookahead items with the same κ will follow exactly the same continuations and thus accept the same language, we can predict that discrimination amongst their actions will be impossible. Such grammars are rejected. Otherwise, a correct parser is produced.

Since those κ exactly code up to at least the first h terminals of right-hand context, inadequacy condition implies that the grammar is not LALR(h).

3.4 Construction Algorithm

In each state r of the lookahead DFA, action j can be decided on symbol a if all lookahead items $[j_i, \kappa_i(\nu_i, [A_i \to \alpha_i \cdot \beta_i]_{q_i})]$ in J_r with $\text{First}(\beta_i) \ni a$ share the same action j. Otherwise, a transition is taken on a to continue lookahead exploration. Accordingly, the following construction algorithm computes the lookahead DFA's action-transition table *At*. Obviously, only new item sets (corresponding to new states) need to be incorporated to *Set-list* for subsequent processing.

DFA(h) GENERATOR:
for each conflict-state q **do**
initialize *Set-list* with $J_{r_0^q}$
repeat
let J_r be next item-set from *Set-list*
$Ps := \{(j, a) \mid [j, \kappa(\nu_l, [A \to \alpha \cdot \beta]_q)] \in J_r, a \in \text{First}(\beta)\}$
for $a \in T \mid \exists (j, a) \in Ps$ **do**
if $\nexists (j', a) \in Ps \mid j' \neq j$ **then** $At(r, a) := j$
else $J_{r'} := \Theta_h(J_r, a)$; add $J_{r'}$ to *Set-list*; $At(r, a) :=$ goto r'
until end of *Set-list* or inadequacy (rejection)

4 Example

We will illustrate our method with a simplified grammar for HTML forms, as the one discussed in [13]. Forms are sequences of tagged values:

```
Company=BigCo
address=1000 Washington Ave
NYC NY 94123
```

Since a value can span several lines, the end of line is not only a delimiter between tag-value pairs. And since letters can belong to a value, a sequence of letters represents a tag only when immediatly followed by an `=` sign. Thus, in order to know if a delimiter ends a tag-value pair, or if it belongs to the value being processed, an unbounded number of characters must be read. We shall use the following example grammar $\mathcal{G}_f$ for HTML forms, in which l stands for any letter, and c stands for any character other than a letter or the `=` sign (separately including space and end-of-line does not basically modify the problem but complicates the example).

$$S' \xrightarrow{1} S \dashv \qquad S \xrightarrow{2} F \qquad S \xrightarrow{3} S\,c\,F \qquad F \xrightarrow{4} T = V$$
$$T \xrightarrow{5} l \qquad T \xrightarrow{6} T\,l \qquad V \xrightarrow{7} \varepsilon \qquad V \xrightarrow{8} V\,l \qquad V \xrightarrow{9} V\,c$$

4.1 Solutions with Commonly Used Parser Generators

It is easy to see that G_f is neither LR(k) nor LL(k) for any k [7].

The ANTLR solution, as given in [13], can be roughly described as follows (terminals are in capitalized letters):

```
// parser
form      : ( TAG string )+                              ;
string    : ( CHAR )+                                    ;
// scanner
TAG       : ( 'a' .. 'z' | 'A' .. 'Z' )+                 ;
FORMTOKEN : (TAG '=' ) ==> TAG '='  {$setType(TAG);}   // predicate
          | .                       {$setType(CHAR);}  ;
```

This solution is rather complex and tricky: it needs to define a scanner that uses a syntactic predicate (and thus reads characters until an `=` or a non-letter is found) to set the token kind (`TAG`, whose value is a letter sequence, or `CHAR`, whose value is a single character). In the proper "grammar" (defined by the parser), no delimiter ends a tag-value pair, what makes the grammar ambiguous, e.g., `t=xyz=v` can be interpreted in two possible ways, as `t=x yz=v` or as `t=xy z=v`, since tags are defined as letter sequences followed by `=`. Parsing is unambiguous only because the scanner reads the whole tag when encountering its first letter.

Furthermore, this solution is inefficient: for each letter of a letter sequence in a value, the predicate is checked, and, in the worst case, parsing time is $O(n^2)$ on the input length.

No better solution can be easily found with *Yacc*; `TAG` and `CHAR` are tokens, and the trick is to let *Lex* do the work in a way analogous to ANTLR:

```
[a-zA-Z]+/= return TAG; /* TAG only if followed by an = */
.          return CHAR;
```

Again, scanning can be $O(n^2)$. Means offered by *Yacc* to resolve conflicts (token priority) are of no help. Letters can be read only once by adding

```
[a-zA-Z]+/[^=] return CHAR; /* CHAR only if not followed by an = */
```

to the Lex code. This simply shows that making letter sequences a token is not enough, if done independently of the right-hand context.

4.2 The Bounded Graph-Connect LRR Solution

G_f is LR-regular because there exist discriminant right-hand regular languages that allow to decide if next character is a form separator, or if it belongs to the value being processed. These languages can be respectively described by the regular expressions $\dashv|cl^+=$ and $l|cl^*(c|\dashv)$. A user of ANTLR (or *Yacc*+*Lex*) should give these regular expressions as syntactic predicates for a clean solution.

[7] One can argue that, on this example, sequences of letters can be a token. This simply would result in an LALR(3) or LL(3) grammar, that neither *Yacc* nor ANTLR can handle without help from the user, as shown here.

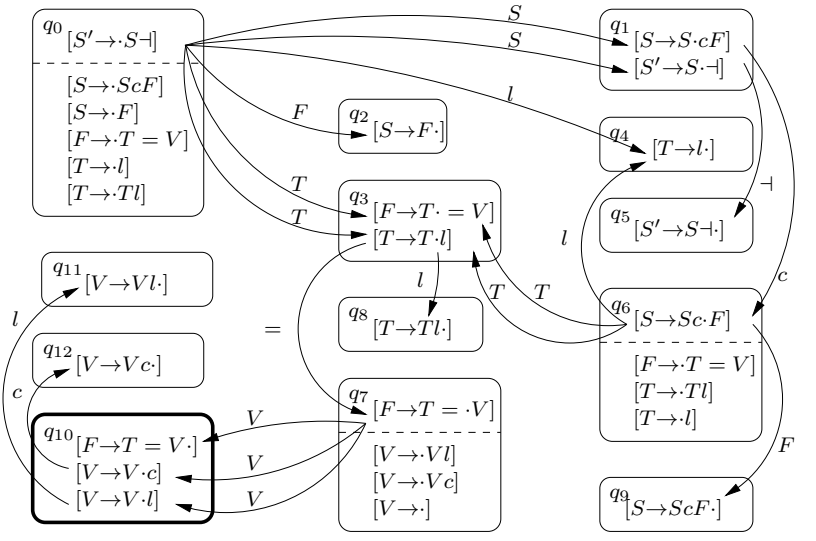

Fig. 6. LR(0) states and kernel-item graph for grammar $\mathcal{G}_f$

Finding these regular expressions is not that easy, especially for someone not trained in compiler/translator writing.

Let us see how our method correctly solves the problem, without user help. Figure 6 shows the corresponding LR(0) states and their underlying graph of kernel items. This automaton has one conflict state, q_{10}. Its (kernel) items are used for the initial state of a lookahead DFA, whose construction (for $h = 1$) is summarized in Fig. 7. Here is how $\hat{\theta}^*$ works for $(\nu, \nu), \nu = [F\to T = V\cdot]_{q_{10}}$ in r_0:

- On one hand, $[S'\to\cdot S\dashv]_{q_0} \xrightarrow{T=V} \nu$, $[S'\to\cdot S\dashv]_{q_0} \xrightarrow{F} [S\to F\cdot]_{q_2}$, followed by a new context-recovery step: $[S'\to\cdot S\dashv]_{q_0} \xrightarrow{S} [S'\to S\cdot\dashv]_{q_1}$ and $[S'\to\cdot S\dashv]_{q_0} \xrightarrow{S} [S\to S\cdot cF]_{q_1}$.
- On the other hand, $[S\to Sc\cdot F]_{q_6} \xrightarrow{T=V} \nu$, and $[S\to Sc\cdot F]_{q_6} \xrightarrow{F} [S\to ScF\cdot]_{q_9}$; again, another context-recovery step takes place and, since $[S'\to\cdot S\dashv]_{q_0} \xrightarrow{ScF} [S\to ScF\cdot]_{q_9}$, we have the same transitions on S as above.

According to current-position nodes in the resulting lookahead item set J_{r_0}, lookahead symbol $\dashv$ is only associated to reduction 4, and l to action 0 (shift), so they can be decided. On the other hand, since c does not allow to decide, a transition is needed to r_1.

Decisions and transitions are likewise computed for the remaining states, resulting in the automaton shown in Fig. 8. In r_1 a transition on l to r_2 is needed because $l \in \mathrm{First}(F)$, while for r_2 some graph transitions are actually

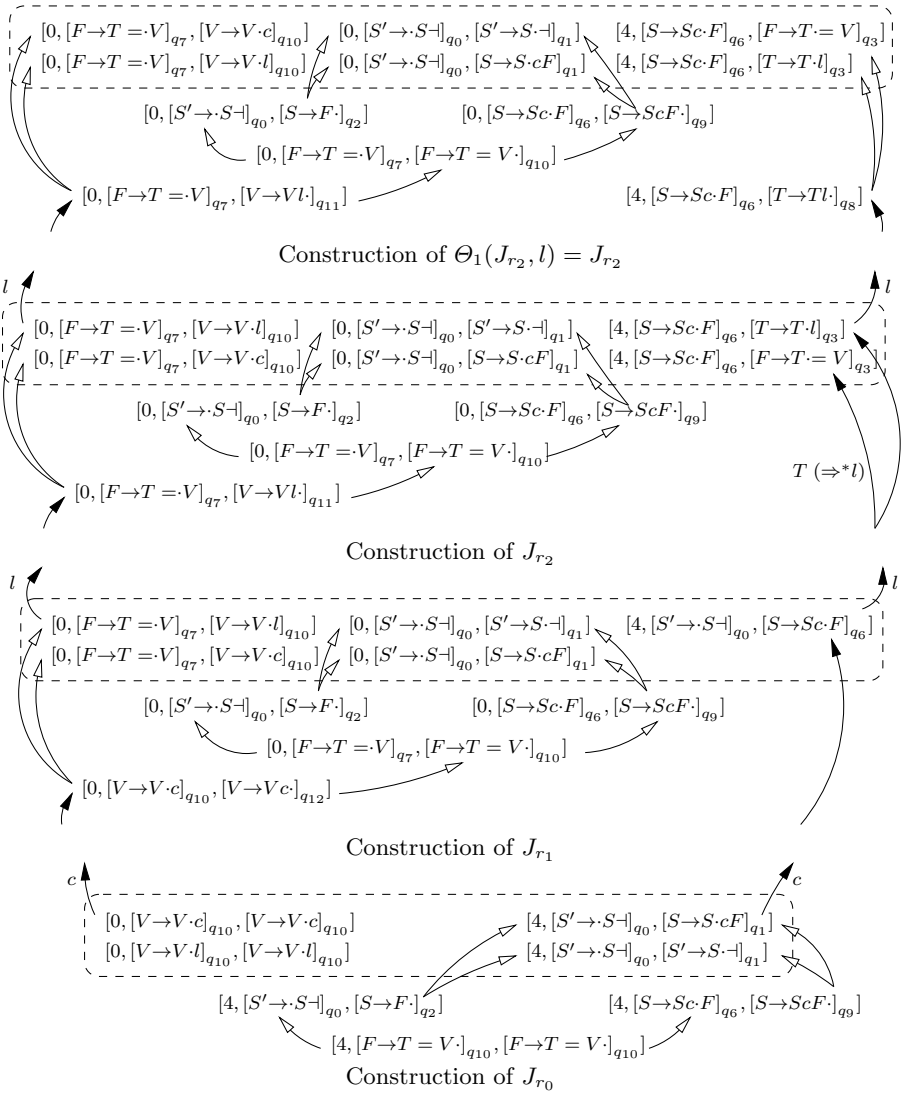

Fig. 7. DFA construction for grammar $\mathcal{G}_f$

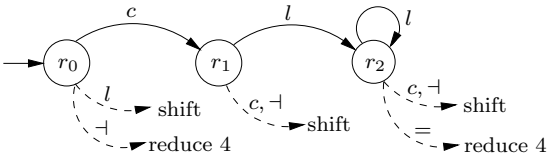

Fig. 8. Lookahead DFA for G_f

taken on T, since $T \Rightarrow^* l$. In this latter state, these downward transitions imply a connect followed by truncation to $h = 1$, although the precision loss will not affect the ability to resolve the conflict. Finally, $\Theta_1(J_{r_2}, l) = J_{r_2}$, what produces a loop in the lookahead DFA, and terminates the construction.

The reader might like to verify that time complexity for parsing is linear on input length. Following the automaton, it is easy to verify that each input character is checked at most two times. For a sequence $c_0 l_1 \cdots l_n c_1 \cdots c_m l_{n+1} \cdots$, $c_1 \neq$ =, $c_0 l_1 \cdots l_n c_1$ is checked a first time, and c_0 is shifted. Next, the l_i are checked, and then shifted, one at a time. Finally, $c_i c_{i+1}, i = 1, \ldots, m-1$ are checked and c_i is shifted, until input is $c_m l_{n+1} \cdots$, which brings back to the first step.

Note that GLR, while producing a nondeterministic parser, gives the same complexity: it maintains two stacks as long as it cannot decide whether a letter belongs to a value or to a tag.

5 Conclusions

Available parser generation tools are not sufficiently powerful to automatically produce parsers for computer language grammars, while previous attempts to develop LR-regular parsers have failed to guarantee sufficient parsing power, since their context recovery is based on LR(0) states only.

Recent research on NDR parsing has provided with an item graph-connecting technique that can be applied to the construction of lookahead DFA to resolve LR conflicts. The paper shows how to build such DFA from an LR(0) automaton by using the dependency graph amongst kernel-items. By combining ε-skip with compact graph-connect coding, we overcome the difficulties of previous approaches. As a result, largely improved context-recovery is obtained while staying relatively simple.

While less powerful than NDR, these LRR parsers naturally preserve the properties of canonical derivation, correct prefix, and linearity. They accept a wide superset of LALR(h) grammars, including grammars not in LR, and thus permit to build practical, automatic generation tools for efficient and reliable parsers.

References

1. J. Aycock and R. N. Horspool. Faster generalized LR parsing. In S. Jähnichen, editor, *Compiler Construction. 8th International Conference, CC'99*, Lecture Notes in Computer Science #1575, pages 32–46. Springer, 1999.
2. T. P. Baker. Extending look-ahead for LR parsers. *J. Comput. Syst. Sci.*, 22(2):243–259, 1981.
3. M. E. Bermudez and K. M. Schimpf. Practical arbitrary lookahead LR parsing. *Journal of Computer and System Sciences*, 41:230–250, 1990.
4. P. Boullier. *Contribution à la construction automatique d'analyseurs lexicographiques et syntaxiques.* PhD thesis, Université d'Orléans, France, 1984. In French.
5. K. Čulik II and R. Cohen. LR-regular grammars — an extension of LR(k) grammars. *J. Comput. Syst. Sci.*, 7:66–96, 1973.
6. J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, Feb. 1970.
7. J. Farré and J. Fortes Gálvez. A basis for looping extensions to discriminating-reverse parsing. In S. Yu, editor, *Fifth International Conference on Implementation and Application of Automata, CIAA 2000.* To appear in LNCS. Springer.
8. J. Gosling, B. Joy, and G. Steele. *The Java^{TM} Language Specification.* Addison-Wesley, 1996.
9. D. Grune and C. J. H. Jacobs. A programmer-friendly LL(1) parser generator. *Software—Practice and Experience*, 18(1):29–38, Jan. 1988.
10. S. Heilbrunner. A parsing automata approach to LR theory. *Theoretical Computer Science*, 15:117–157, 1981.
11. A. Johnstone and E. Scott. Generalised recursive descent parsing and follow-determinism. In K. Koskimies, editor, *Compiler Construction. 7th International Conference, CC'98*, Lecture Notes in Computer Science #1383, pages 16–30. Springer, 1998.
12. B. Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Automata, Languages and Programming*, Lecture Notes in Computer Science #14, pages 255–269. Springer, 1974.
13. T. J. Parr. We are talking really big lexical lookahead here. *www.antlr.org/articles.html.*
14. T. J. Parr and R. W. Quong. ANTLR: A predicated-$LL(k)$ parser generator. *Software—Practice and Experience*, 25(7):789–810, July 1995.
15. B. Seité. A Yacc extension for LRR grammar parsing. *Theoretical Computer Science*, 52:91–143, 1987.
16. S. Sippu and E. Soisalon-Soininen. *Parsing Theory.* Springer, 1988–1990.
17. T. G. Szymanski and J. H. Williams. Non-canonical extensions of bottom-up parsing techniques. *SIAM J. Computing*, 5(2):231–250, June 1976.
18. M. Tomita. *Efficient Parsing for Natural Language.* Kluwer, 1986.
19. T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. *ACM SIGPLAN Notices*, 32(5):31–43, 1997.