# A Proposal on Error Repair

M. Vilares V.M. Darriba F.J. Ribadas

**Abstract**

We describe an algorithm to deal with automatic error repair over unrestricted context-free languages. The method relies on a regional least-cost repair strategy with validation, gathering all relevant information in the context of the error location to provide a good diagnosis. The system guarantees the asymptotic equivalence with global repair strategies. The work is organized around a parsing frame for unrestricted context-free grammars. Our approach is based on a dynamic programming strategy, which guarantees an optimal computation and syntactic sharing.

*Keywords: Context-free parsing, dynamic programming, regional least-cost error repair, push-down automaton, shared forest.*

## 1 Introduction

One critical aspect of a parser that determines its effectiveness is the error repair strategy applied, that is, the capacity to allow the parsing of an imperfect text to continue past a point of error so as to discover as many as possible of the additional errors that it may contain. Until recently, errors were simply recovered by the consideration of fiducial symbols, typically reserved key words of a language, to provide mile-posts for error recovery [10]. This approach allows a reduction in time and space bounds, although it is not always easy to determine if all relevant information to the error recovery process has been seen [8].

At present, the significant reduction in cost processing has propitiated a renewable interest in error repair methods that take into account the constraints on context. We can here differentiate [8] two families of algorithms: one class, called *local repair*, make modifications to the input so that at least one more original input symbol can be accepted by the parser [2, 4, 11]. Although these methods give good results in most cases, their simplicity sometimes causes them to choose a poor repair [9, 3].

In contrast to local techniques, the *global repair* algorithms examine the entire program and make a minimum of changes to repair all the syntax errors [1, 6, 10]. Global methods give the best repairs possible, but they are not efficient. Since they

M. Vilares, V.M. Darriba and F.J. Ribadas are with Department of Computer Science, Faculty of Informatics, University of A Coruña. Campus de Elviña s/n, 15071 A Coruña, Spain. E-mail: {vilares|darriba|ribadas}@dc.fi.udc.es

This work has been partially supported by projects PGIDT 99XI10502B of the Autonomous Government of Galicia, and project 1FD97-0047-C04-02 by the European Community.

expend equal effort on all parts of the program, including areas that contain no errors, much of that effort is wasted.

In between the local and global methods, Levi [5], and Mauney and Fischer [7] suggested *regional repair* algorithms that fix a portion of the program including the error and as many additional symbols as needed to assure a good repair. The global and local algorithms are not efficient enough for practical use, and the regional algorithms must answer the additional question of determining just how large a region to repair.

In addition, whatever the approach applied, when several repairs are available for a same error, the system must provide some method of choosing among them, and a common strategy is to assign individual costs to edit operations. A repair algorithm that guarantees finding the lowest-cost repair possible, is called a *least-cost* repair algorithm.

The algorithm we have developed is a regional least-cost strategy which applies a dynamic validation in order to avoid cascaded errors, that is, errors caused by incorrect repair assumption. Our aim is to completely resume the parse at the point of each error, so as not to miss detecting any subsequent errors, providing a high confidence level for the process.

## 2 A dynamic frame for parsing

We introduce our parsing frame in dynamic programming, as implemented in Ice [13]. Our aim is to parse sentences in the language $\mathcal{L}(\mathcal{G})$ generated by a context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$, where $N$ is the set of non-terminals, $\Sigma$ the set of terminal symbols, $P$ the rules and $S$ the start symbol. The empty string will be represented by $\varepsilon$.

### 2.1 The operational model

We assume that, using a standard technique, we produce a *push-down automaton* (PDA), based on a shift-reduce strategy, from the grammar $\mathcal{G}$. In practice, we chose an LALR(1) device, which is possibly non-deterministic, for the language $\mathcal{L}(\mathcal{G})$, which will allow us to improve sharing of computations and representations [13].

Formally, a PDA can be represented as a 7-tuple $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, \delta, q_0, Z_0, \mathcal{Q}_f)$ where: $\mathcal{Q}$ is the set of states, $\Sigma$ the set of input symbols, $\Delta$ the set of stack symbols, $q_0$ the initial state, $Z_0$ the initial stack symbol, $\mathcal{Q}_f$ the set of final states, and $\delta$ a finite set of transitions of the form $p\ X\ a \mapsto q\ Y$ with $p, q \in \mathcal{Q}$, $a \in \Sigma \cup \{\varepsilon\}$ and $X, Y \in \Delta \cup \{\varepsilon\}$.

Let the PDA be in a configuration $(p, X\alpha, ax)$, where $p$ is the current state, $X\alpha$ is the stack contents with $X$ on the top, $ax$ is the remaining input where the symbol $a$ is the next to be shifted, $x \in \Sigma^*$. The application of a transition $p\ X\ a \mapsto q\ Y$ results in a new configuration $(q, Y\alpha, x)$ where the terminal symbol $a$ has been scanned, $X$ has been popped, and $Y$ has been pushed. If the terminal symbol $a$ is $\varepsilon$ in the transition, no input symbol is scanned. If $X$ is $\varepsilon$ then no stack symbol is popped from the stack. In a similar manner, if $Y$ is $\varepsilon$ then no stack symbol is pushed on the stack. In the case of ambiguous recognition, several such transitions can be applied and the recognizer must manage a set of stacks in an efficient manner.

The algorithm proceeds by building a collection of *items*, essentially compact representations of the recognizer stacks in order to guarantee a good level of sharing of the computational process. New items are produced by applying transitions to existing ones, until no new application is possible. The algorithm associates a set of items $S^w_i$, usually called *itemset*, for each input symbol $w_i$ at the position $i$ in the input string of length $n$, $w_{1..n}$.

An item is of the form $[p, X, S^w_j, S^w_i]$, where $p$ is a PDA state, $X$ is a stack symbol, $S^w_j$ is the *back pointer* to the itemset associated to the input symbol $w_i$ at which we began to look for that configuration of the automaton, and $S^w_i$ is the current itemset.

## 2.2 The recognizer

Formally, given a transition $\tau = \delta(p, X, a) \ni (q, Y)$, we translate it to items of the following form:

$$\begin{array}{lllll}
1. & \tilde{\delta}([p, X, S^w_j, S^w_i], a) & \ni & [q, \varepsilon, S^w_i, S^w_i], & \textbf{if } Y = X \\
2. & \tilde{\delta}([p, X, S^w_j, S^w_i], a) & \ni & [p, Y, S^w_i, S^w_{i+1}], & \textbf{if } Y = a \\
3. & \tilde{\delta}([p, X, S^w_j, S^w_i], a) & \ni & [p, Y, S^w_i, S^w_i], & \textbf{if } Y \in N \\
4. & \tilde{\delta}([p, \varepsilon, S^w_j, S^w_i], a) & \ni & \tilde{\delta}_d([q, \varepsilon, S^w_l, S^w_j], a) \ni [q, \varepsilon, S^w_l, S^w_i], & \textbf{if } Y = \varepsilon \\
 & & & \forall q \in \mathcal{Q} \text{ such that } \exists\, \delta(q, X, \varepsilon) \ni (p, X) &
\end{array}$$

with:

$$\tilde{\delta} : \mathrm{It} \times \Sigma \cup \{\varepsilon\} \longrightarrow \{\mathrm{It} \cup \tilde{\delta}_d\} \qquad\qquad \tilde{\delta}_d : \mathrm{It} \times \Sigma \cup \{\varepsilon\} \longrightarrow \mathrm{It}$$

where *It* is the set of all items developed in the parsing process and $\tilde{\delta}_d$ is called the set of *dynamic transitions*. Succinctly, we can describe the preceding cases as follows:

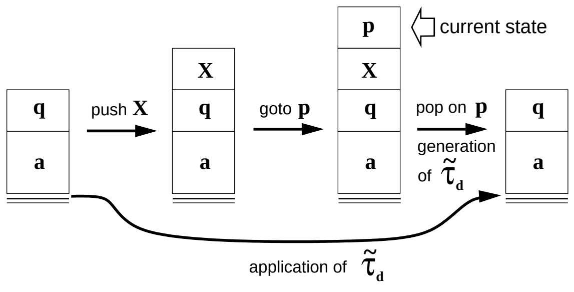
1. A goto action from the state $p$ to state $q$ under transition $X$ in the LALR(1) automaton.

2. A push of terminal $a$ from state $p$. The new item belongs to the next itemset $S^w_{i+1}$.

3. A push of non-terminal $Y$ from state $p$.

4. A pop action from state $p$, where $q$ is an ancestor of state $p$ under transition $X$ in the LALR(1) automaton. In this case, we do not generate a new item, but a *dynamic transition* $\tilde{\tau}_d$ to treat the absence of information about the rest of the stack. This transition is applicable not only to the configuration resulting from the first one, but also on those to be generated and sharing the same syntactic structure, as shown in Fig. 1.

Fairness and completeness of the dynamic programming construction are guaranteed by an equitable selection order in the treatment of items. Transitions may add more items to the current itemset and may also put items in the itemset corresponding to the following token to be analyzed from the input string. Authors prove in [13] that time and space bounds are, in the worst case, $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$ respectivley, when the length of the input string is $n$. These bounds are achieved without the language grammar having to be in Chomsky Normal Form, as was usual in previous works [12].



p
current state
X
q
a
push X
goto p
pop on p
generation of $\tilde{\tau}_d$
application of $\tilde{\tau}_d$

Figure 1: Dynamic transitions in ICE

## 3 Regional least-cost error repair

Following Mauney and Fischer in [8], we talk about the *error* in a portion of the input to mean the difference between what was intended and what actually appears in the input. The repair algorithm cannot actually know what the error is, but the concept is useful for purposes of discussion. So, we can talk about the *point of error* as the point at which the difference occurs. The *point of detection* is the point at which the parser detects that there is an error in the input and calls the repair algorithm. We now introduce the formal concepts.

**Definition 1.** *Let $w_{1..n}$ be an input string, we say that $w_i$ is a* point of error *iff:* $\nexists\ [p, \varepsilon, S_l^w, S_i^w] \in S_i^w / \delta(p, X, w_i) = (q, w_i)$

The point of error is easily fixed by the parser itself and, in order to locate the origin of the error at minimal cost, we should try to limit the impact on the parse, focusing on the context of subtrees close to the point of error.

**Definition 2.** *Let $w_i$ be a point of error for the input string $w_{1..n}$, we define the set of* points of detection associated to *$w_i$, as follows:*

$$detection(w_i) = \{w_{i'} / \exists A \in N,\ A \overset{+}{\Rightarrow} w_{i'} \alpha w_i\}$$

*and we say that $A \overset{+}{\Rightarrow} w_{i'} \alpha w_i$ is a* derivation defining the point of detection *$w_{i'} \in detection(w_i)$.*

Intuitively, the error is located in the immediate left parse context, represented by the closest viable node [1], or in the immediate right context, represented by the lookahead. However, sometimes it can be useful to isolate the parse branch [2] in

[1]that is, the closest node including the current token, for which a repair is possible

[2]that is, the deterministic sequence of items we are dealing with at that moment

which the error appears. A natural manner to do that is to consider the concepts previously introduced in terms of items.

**Definition 3.** *Let $w_i$ be a point of error for $w_{1..n}$, we say that $[p, X, S^w_l, S^w_i] \in S^w_i$ is an* error item *iff:*

$$\exists\, a \in \Sigma,\ \delta(p, \varepsilon, a) \neq \emptyset$$

*and we say that $[p, \varepsilon, S^w_{i'}, S^w_{i'}] \in S^w_{i'}$ is a* detection item *associated to $w_i$ iff:*

$$\begin{array}{ll}
\exists\, a \in \Sigma, \delta(p, A, a) \neq \emptyset, & A \in N \textit{ defining } w_i \\
 & \\
\delta(p, \varepsilon, w_{i'}) \ni (p, B_1), & \delta(p, B_1, w_{i'}) \ni (q_1, \varepsilon) \\
\delta(q_1, \varepsilon, w_{i'}) \ni (q_1, B_2), & \delta(q_1, B_2, w_{i'}) \ni (q_2, \varepsilon) \\
\vdots & \vdots \\
\delta(q_{n-1}, \varepsilon, w_{i'}) \ni (q_{n-1}, B_n), & \delta(q_{n-1}, B_n, w_{i'}) \ni (q_n, \varepsilon) \\
\delta(q_n, \varepsilon, w_{i'}) \ni (q_n, w_{i'}), & B_i \overset{+}{\Rightarrow} \varepsilon,\ \forall i \in [1, n]
\end{array}$$

Intuitively, we talk about *error item* and *detection item*, when they represent nodes including the recognition of tokens classified, respectively, as point of error and point of detection. The condition for error items implies that no scan action is possible for token $w_i$. In the detection case, conditions look for items recognizing a point of detection on a parse branch including an error item in $w_i$, disregarding empty reductions which are not relevant for this purpose. At this point, we can regard the concept at the basis of error repair, as an operative mechanism to modify an erroneous input string.

**Definition 4.** *A* modification *$M$ to a string of length $n$, $w_{1..n} = w_1 \dots w_n$, is a series of edit operations, $E_1 \dots E_n E_{n+1}$, in which each $E_i$ is applied to $w_i$ and possibly consists of a series of insertions before $w_i$, replacements or deletion of $w_i$. The string resulting from the application of the modification $M$ to the string $w$ is written $M(w)$.*

We now restrict the notion of modification to focus on a given zone of the input string, introducing the concept of error repair in this space. Intuitively, we look for conditions that guarantee the ability to recover the parse from the error, at the same time as it allows us to isolate repair branches by using the concept of reduction. We are also interested in minimizing the structural impact in the parse tree, and finally in introducing the notion of scope as the lowest reduction summarizing the process at a point of detection.

**Definition 5.** *Let $x$ be a valid prefix in $\mathcal{L}(\mathcal{G})$, and $w \in \Sigma^*$, such that $xw$ is not a valid prefix in $\mathcal{L}(\mathcal{G})$. We define a* repair of $w$ following $x$ *as a modification $M(w)$, so that:*

$$\exists A \in N \Bigg/ \begin{array}{l}
S \overset{+}{\Rightarrow} x_{1..i-1} A \overset{+}{\Rightarrow} x_{1..i-1} x_{i..m} M(w),\ i \leq m \\
B \overset{*}{\Rightarrow} \alpha A \beta,\ \forall B \overset{+}{\Rightarrow} x_{j..m} M(w),\ j < i \\
A \overset{*}{\Rightarrow} \gamma C \rho,\ \forall C \overset{+}{\Rightarrow} x_{i..m} M(w)
\end{array}$$

*We denote the set of repairs of $w$ following $x$ by* repair$(x, w)$*, and $A$ by* scope$(M)$*.*

However, the notion of *repair*$(x, w)$ is not sufficient for our purposes, since our aim is to extend the error repair process to consider all possible points of detection proposed by the algorithm for a given point of error, which implies simultaneously considering different valid prefixes and repair zones.

**Definition 6.** *Let $e \in \Sigma$ be a point of error, we define the set of* repairs for $e$, *as follows:*

$$\textit{repair}(e) = \{xM(w) \in \textit{repair}(x, w) / w_1 \in \textit{detection}(e)\}$$

*where* detection$(e)$ *denotes the set of points of detection associated to $e$.*

We now need a mechanism to filter out undesirable repair processes, in order to reduce the computational charges. To do that, we should introduce comparison criteria to only select those repairs with minimal cost.

**Definition 7.** *For each $a \in \Sigma$ we assume the existence of positive insert, $I(a)$; delete, $D(a)$, and replace $R(a)$ costs*[3]. *The* cost of a modification[4] $M(w_{1..n})$ *is given by* $\mathrm{cost}(M(w_{1..n})) = \Sigma_{i=1}^{n}(\Sigma_{j \in J} I(w_i^j) + D(w_i) + R(w_i))$. *In particular, $\Sigma_{j \in J} I(w_i^j)$ means that several insertion hypotheses are possible before the token $w_i$ is real.*

When several repairs are available on different points of detection, we need a condition to ensure that only those with the same minimal cost are considered, looking for the best repair quality.

**Definition 8.** *Let $e \in \Sigma$ be a point of error, we define the set of* regional repairs for $e$, *as follows:*

$$\textit{regional}(e) = \{xM(w) \in \textit{repair}(e) \Big/ \begin{array}{l} \textit{cost}(M) \leq \textit{cost}(M'),\ \forall M' \in \textit{repair}(x, w) \\ \textit{cost}(M) = \min_{L \in \textit{repair}(e)}\{\textit{cost}(L)\} \end{array} \}$$

It is also necessary to take into account the possibility of dealing with errors precipitated by previous unsuccessful repair processes. Previous to dealing with the problem, we need to establish the existing relationship between the regional repairs for a given point of error, and future points of error.

**Definition 9.** *Let $w_i, w_j$ be points of error in an input string $w_{1..n}$, such that $j > i$. We define the set of* viable repairs for $w_i$ in $w_j$, *as follows:*

$$\textit{viable}(w_i, w_j) = \{xM(y) \in \textit{regional}(w_i) / xM(y) \ldots w_j \textit{ valid prefix for } \mathcal{L}(\mathcal{G})\}$$

Intuitively, the repairs in *viable*$(w_i, w_j)$ are the only ones capable of ensuring the continuity of the parse in $w_{i..j}$ and, therefore, the only possible repairs at the origin of the phenomenon of cascaded errors, that is, errors precipitated by a previous erroneous repair diagnostics.

[3] if any edit operation is not applied, we assume its cost to be zero.

[4] we assume that delete and replace operations are exclusive for the same token.

**Definition 10.** *Let $w_i$ be an point of error for the input string $w_{1..n}$, we say that a point of error $w_j$, $j > i$ is a* point of error precipitated by $w_i$ *iff*

$$\forall x M(y) \in \textit{viable}(w_i, w_j),\ \exists A \in N \textit{ defining } w_{j'} \in \textit{detection}(w_j)$$

*such that*

$$A \overset{+}{\Rightarrow} \beta \textit{scope}(M) \ldots w_j$$

Intuitively, a point of error $w_j$ is precipitated by the result of previous repairs on a point of error $w_i$, when all reductions defining points of detection for $w_j$ summarize some viable repair for $w_i$ in $w_j$. Otherwise, some of these points of detection are outside the scope of syntactic structures extracted from viable repairs on a previous point of error, and the incorrectness of the parse cannot be attributed to a bad recovery from $w_i$, since it is possible to apply a repair process independently on previous repair assumptions.

## 4 The algorithm

Let us begin by summarizing the main features of our proposal. Essentially, we propose that the repair be obtained by searching the PDA itself to find a suitable configuration to allow the parse to continue. At this point, our approach agrees with McKenzie *et al.* in [9], although this method is not asymptotically equivalent to a global repair strategy, and introduces an unsafe technique to speed up the repair algorithm [3]. In relation to this last, McKenzie *et al.* propose a pruning mechanism in order to reduce the number of configurations to be dealt with during the repair process. This mechanism may lead to suboptimal regional repairs or may cause failure to produce any repair even if an error exists.

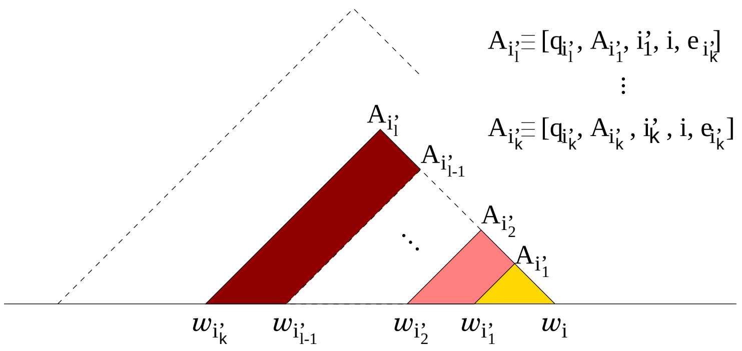

$A_{i'_l} \equiv [q_{i'_l}, A_{i'_1}, i'_1, i, e_{i'_k}]$
⋮
$A_{i'_k} \equiv [q_{i'_k}, A_{i'_k}, i'_k, i, e_{i'_k}]$
$A_{i'_l}$
$A_{i'_{l-1}}$
$A_{i'_2}$
$A_{i'_1}$
$w_{i'_k}$
$w_{i'_{l-1}}$
$w_{i'_2}$
$w_{i'_1}$
$w_i$

Figure 2: Error detection

The problem due to pruning is based on a simple condition that ignores a stack configuration if an earlier one had the same stack top. The motivation is that this newer configuration would not lead to any cheaper repairs than the older one. Our dynamic programming construction eliminates this problem by considering all possible repair paths.

## 4.1 A simple case

In order to simplify the description, we first assume that we deal with the first error detected in the input string. The major features of the algorithm involve beginning with a list of error items, with an error counter zero. In order to compute the error counter, we extend the item structure: $[p, X, S_i^w, S_j^w, \mathrm{e}]$, where now $e$ is the error counter accumulated in the recognition of $X \in N \cup \Sigma$.

For each error item, we successively investigate the corresponding list of detection items, one for each parse branch including the error item. One a point of error $w_i$ has been fixed, we can associate to it different points of detection $w_{i'_1}, \ldots w_{i'_k}$, as is shown in Fig. 2. Detection items are located by using the back pointer, that indicates the itemset where we have applied the last PDA action. So, we recursively go back into its ancestors until we find the first descendant of the last node that would have to be reduced if the lookahead was correct[5].

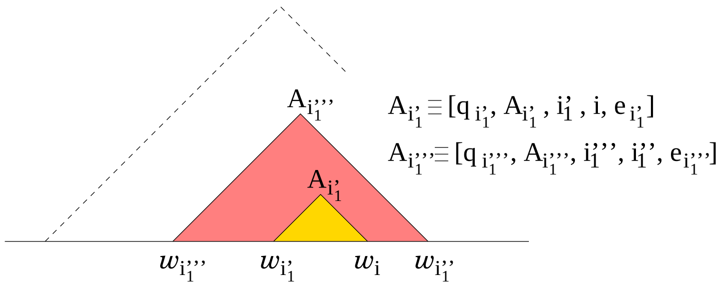

$A_{i'''_1}$
$A_{i'_1} \equiv [q_{i'_1}, A_{i'_1}, i'_1, i, e_{i'_1}]$
$A_{i'''_1} \equiv [q_{i'''_1}, A_{i'''_1}, i'''_1, i''_1, e_{i'''_1}]$
$A_{i'_1}$
$w_{i'''_1}$
$w_{i'_1}$
$w_i$
$w_{i''_1}$

Figure 3: Repair scope

Once the detection items have been fixed for the corresponding error item, on each of the parse branches relying on them we apply all possible transitions beginning at the point of detection. These transitions correspond to four error hypotheses, from a given item:

- For scan transitions the item obtained is the same as for standard parsing.

$$[p, \varepsilon, S_j^w, S_i^w, 0] \overset{\text{scan}\ w_i}{\longrightarrow} [p, w_i, S_i^w, S_{i+1}^w, 0]$$

- In the case of insertion hypothesis, we initialize the error counter by taking into account the cost of the inserted token, which is included as stack symbol, and we add the new item to the same itemset, preserving the back pointer.

$$[p, \varepsilon, S_j^w, S_i^w, 0] \overset{\text{insert}\ a}{\longrightarrow} [p, a, S_i^w, S_i^w, I(a)],\ \delta(p, \varepsilon, a) \neq \emptyset$$

- For deletion hypothesis, we initialize the error counter by taking into account the cost of the deleted token and we add the new item to the next itemset, using the same stack symbol. The back pointer is initialized to the current itemset.

$$[p, \varepsilon, S_j^w, S_i^w, 0] \overset{\text{delete}\ w_i}{\longrightarrow} [p, \varepsilon, S_i^w, S_{i+1}^w, D(w_i)]$$

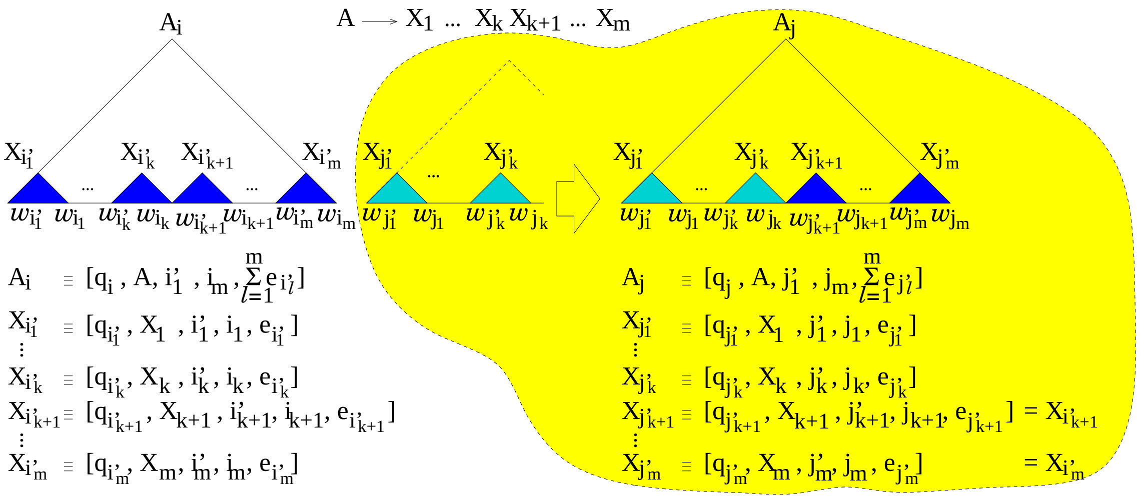
[5]this information is directly obtained from the PDA.

- Finally, for mutation hypothesis, we initialize the error counter by taking into account the cost of the replaced token and we add the new item to the next itemset. The back pointer is initialized to the current itemset, and the new token resulting from the mutation is included as stack symbol.

$$[p, \varepsilon, S^w_j, S^w_i, 0] \overset{\text{replace } w_i \text{ by } a}{\longrightarrow} [p, a, S^w_i, S^w_{i+1}, R(a)], \; \delta(p, \varepsilon, a) \neq \emptyset$$

We do that until a reduction verifying definition 5 covers both error and detection items accepting a token in the remaining input string, as is shown in Fig. 3, where $[w_{i'''_1}, w_{i''_1}]$ delimits the scope of a repair detected at the point $w_{i'_1} \in \mathit{detection}(w_i)$.

Once we have applied the previous methodology to each detection item considered, we take only those repairs with regional lowest cost, applying definition 8. At this moment the parse goes back to standard mode on all parse branches until a new error is detected.



$A_i$
$A \longrightarrow X_1 \ldots X_k X_{k+1} \ldots X_m$
$A_j$
$X_{i'_1}$ $X_{i'_k}$ $X_{i'_{k+1}}$ $X_{i'_m}$
$X_{j'_1}$ $X_{j'_k}$
$X_{j'_1}$ $X_{j'_k}$ $X_{j'_{k+1}}$ $X_{j'_m}$
$w_{i'_1}$ $w_{i_1}$ $w_{i'_k}$ $w_{i_k}$ $w_{i'_{k+1}}$ $w_{i_{k+1}}$ $w_{i'_m}$ $w_{i_m}$
$w_{j'_1}$ $w_{j_1}$ $w_{j'_k}$ $w_{j_k}$
$w_{j'_1}$ $w_{j_1}$ $w_{j'_k}$ $w_{j_k}$ $w_{j'_{k+1}}$ $w_{j_{k+1}}$ $w_{j'_m}$ $w_{j_m}$
$A_i \equiv [q_i, A, i'_1, i_m, \sum_{l=1}^{m} e_{i'_l}]$
$X_{i'_1} \equiv [q_{i'_1}, X_1, i'_1, i_1, e_{i'_1}]$
$X_{i'_k} \equiv [q_{i'_k}, X_k, i'_k, i_k, e_{i'_k}]$
$X_{i'_{k+1}} \equiv [q_{i'_{k+1}}, X_{k+1}, i'_{k+1}, i_{k+1}, e_{i'_{k+1}}]$
$X_{i'_m} \equiv [q_{i'_m}, X_m, i'_m, i_m, e_{i'_m}]$
$A_j \equiv [q_j, A, j'_1, j_m, \sum_{l=1}^{m} e_{j'_l}]$
$X_{j'_1} \equiv [q_{j'_1}, X_1, j'_1, j_1, e_{j'_1}]$
$X_{j'_k} \equiv [q_{j'_k}, X_k, j'_k, j_k, e_{j'_k}]$
$X_{j'_{k+1}} \equiv [q_{j'_{k+1}}, X_{k+1}, j'_{k+1}, j_{k+1}, e_{j'_{k+1}}] = X_{i'_{k+1}}$
$X_{j'_m} \equiv [q_{j'_m}, X_m, j'_m, j_m, e_{j'_m}] = X_{i'_m}$

Figure 4: Dynamic transitions in repair mode

It is important here to underline the role of dynamic transitions in error mode. We use a bottom-up strategy not only to parse the input, but also to compute error counters. This establishes a difference with McKenzie *et al.* [9], that uses a bottom-up parsing architecture with a top-down computation of the error counters. So, in the last case, the error counter associated to a stack configuration is computed from the error counter inherited from the previous stack configuration. In a deterministic context or in a non-deterministic one without sharing of computations[6] this characteristic has no consequences, but this is not the case in our proposal.

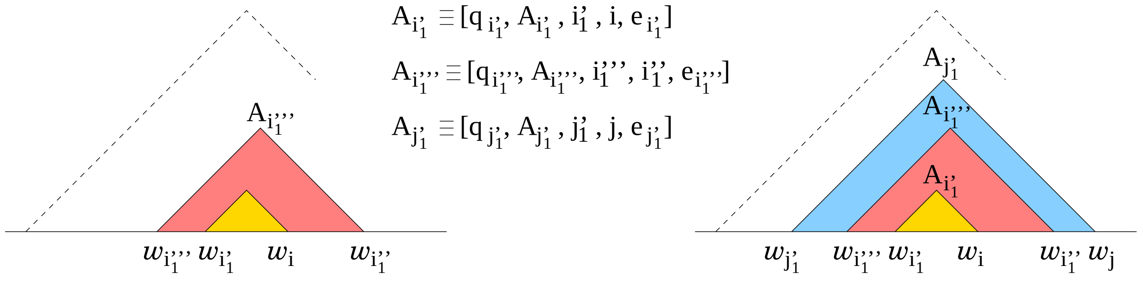
An inheritance strategy to compute the error counters, make the task of propagating these counters on shared parse branches a complex one. In our case, error counters are initialized at each error hypothesis and summarized only at reduce actions time. This implies that, in order to trap reductions in repair mode over stack configurations not yet available, dynamic transitions must include information about the accumulated error counter in the part of the reduce action to be shared. The

[6]typically based on parsers with backtracking.

process is illustrated in Fig. 4 for two reductions, $A_i$ and $A_j$, over a same rule $A \to X_1 \ldots X_m$ sharing the last $X_{k+1} \ldots X_m$ syntactic categories. We re-take the part of the error counter accumulated during the first reduction, $e_{i'_{k+1}} + \ldots + e_{i'_m}$, for these common categories.

## 4.2 The general case

We now assume that the current repair process is not the first one and, therefore, can modify a previously repaired string. This arises when we realize that we come back to a detection item for which any parse branch includes a previous repair process. This process is illustrated in Fig. 5 for a point of error $w_j$ precipitated by $w_i$, showing how the variable $A_{j'_1}$ defining $w_j$ summarizes $A_{i'''_1}$, the scope of a previous repair defined by $A_{i'_1}$.



$A_{i'_1} \equiv [q_{i'_1}, A_{i'_1}, i'_1, i, e_{i'_1}]$
$A_{i'''_1} \equiv [q_{i'''_1}, A_{i'''_1}, i'''_1, i''_1, e_{i'''_1}]$
$A_{j'_1} \equiv [q_{j'_1}, A_{j'_1}, j'_1, j, e_{j'_1}]$
$A_{i'''_1}$
$w_{i'''_1}$ $w_{i'_1}$ $w_i$ $w_{i''_1}$
$A_{j'_1}$
$A_{i'''_1}$
$A_{i'_1}$
$w_{j'_1}$ $w_{i'''_1}$ $w_{i'_1}$ $w_i$ $w_{i''_1}$ $w_j$

Figure 5: Dealing with precipitated errors

To deal with precipitated errors, the algorithm re-takes the previous error counters, adding the cost of the new error repair hypothesis to profit from the experience gained from previous repair processes. This implies that the error repair scheme has the capacity to revoke earlier repairs introducing spurious errors into an otherwise syntactically correct text fragment.

At this point, the definition of regional repair has two important properties which encourage its adoption as a means of effecting error repair. First, it is independent of the shift-reduce parsing algorithm used. The second property is a consequence of the lemma below.

**Lemma 1.** *(The Expansion Lemma) Let $w_i$, $w_j$ be points of error in $w_{1..n} \in \Sigma^*$, such that $w_j$ is precipitated by $w_i$, then*

$$\mathit{min}\{j'/w_{j'} \in \mathit{detection}(w_j)\} < \mathit{min}\{i'/w_{i'} = y_1,\ xM(y) \in \mathit{viable}(w_i, w_j)\}$$

*Proof.* Let $w_{i'} \in \Sigma$, such that $w_{i'} = y_1$, $xM(y) \in \mathit{viable}(w_i, w_j)$ be a point of detection for $w_i$, for which some parsing branch derived from a repair in $\mathit{regional}(w_i)$ has successfully arrived at $w_j$.

Let $w_j$ be a point of error precipitated by $xM(y) \in \mathit{viable}(w_i, w_j)$. By definition, we can assure that

$$\exists B \in N / B \overset{+}{\Rightarrow} w_{j'} \alpha w_j \overset{+}{\Rightarrow} \beta \mathrm{scope}(M) \ldots w_j \overset{+}{\Rightarrow} \beta x_{l..m} M(y) \ldots w_j,\ w_{i'} = y_1$$

Given that *scope*$(M)$ is the lowest variable summarizing $w_{i'}$, it immediately follows that $j' < i'$, and we conclude the proof by extending the proof to all repairs in *viable*$(w_i, w_j)$.

□

**Lemma 1.** *Let $w_i$, $w_j$ be points of error in $w_{1..n} \in \Sigma^*$, such that $w_j$ is precipitated by $w_i$, then*

$$max\{\text{scope}(M),\ M \in viable(w_i, w_j)\} \subset max\{\text{scope}(\tilde{M}),\ \tilde{M} \in regional(w_j)\}$$

*Proof.* It immediately follows from lemma 1.

□

So, regional repairs allow us to get an asymptotic behavior close to global repair methods [1, 6, 10]. This occurs, typically, when successive precipitated errors lead to an increase in the error repair zone until it includes the total input string. This property has profound implications for the efficiency, measured by time and space taken, the simplicity and the power of computing regional repairs.

**Lemma 2.** *Let $w_{1..n}$ be an input string with a point of error in $w_i$, $i \in [1, n]$, then the time and space bounds for the regional repair algorithm are $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$, in the worst case, respectively.*

*Proof.* It immediately follows from the previous corollary 1, taking into account that the time and space bounds for the parser in standard mode are, in the worst case, $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$ respectively.

□

## 5 Conclusions

The purpose of regional algorithms is to increase available information. To improve the quality of repairs we should gather information to the right and to the left of the point of detection as long as this information could possibly be relevant. Our aim is to find a way of deciding whether additional relevant information can be gathered.

A criterion that meets our requirements is to expand the repair mode until it is guaranteed to accept the next input symbol, but maintains the chance of reconsidering the process once the system has detected that an incorrect repair assumption has been made. So, we depart from strictly recovery techniques [2] for which no backtracking is allowed when constructing the parse, without revoking any earlier computations. Our approach also differs from regional approaches, as described by Mauney and Fischer [8], that focus on static regional lookahead in order to guarantee the equivalence between all minimal repairs, which are essentially processes that do not anticipate errors not yet detected.

The consideration of a dynamic programming framework allows us to deal efficiently with computation and syntactic sharing, in a non-deterministic context with a high rate of growth of the number of computations. The consideration of non-determinism is not dispensable in our proposal. In fact, it is crucial for dealing with both the problem of cascaded errors and asymptotic equivalence with global repair approaches.

## References

[1] A.V. Aho and T.G. Peterson. A minimum distance error-correcting parser for context-free languages. *Siam J. Comput*, 1(4), 1972.

[2] S.O. Anderson and R.C. Backhouse. Locally least-cost error recovery in Earley's algorithm. *ACM Transactions on Programming Languages and Systems*, 3(3):318–347, 1981.

[3] Eberhard Bertsch and Mark-Jan Nederhof. On failure of the pruning technique in "Error Repair in Shift-Reduce Parsers". *ACM Transactions on Programming Languages and Systems*, 21(1):1–10, January 1999.

[4] C.N. Fischer, D.R. Milton, and S.B. Quiring. Efficient LL(1) error correction and recovery using only insertions. *Acta Informatica*, 13:151–154, 1980.

[5] J.P. Levy. Automatic correction of syntax-errors in programming languages. *Acta Informatica*, 4:271–292, 1975.

[6] G. Lyon. Syntax-directed least-errors analysis for context-free languages: A practical approach. *Communications of the ACM*, 17(1):3–14, 1974.

[7] J. Mauney and C. N. Fischer. A forward move algorithm for LL and LR parsers. *ACM SIGPLAN Notices*, 17(6):79–87, June 1982.

[8] J. Mauney and C.N. Fischer. Determining the extend of lookahead in syntactic error repair. *ACM Transactions on Programming Languages and Systems*, 10(3):456–469, 1988.

[9] Bruce J. McKenzie, Corey Yeatman, and Lorraine De Vere. Error repair in shift-reduce parsers. *ACM Transactions on Programming Languages and Systems*, 17(4):672–689, July 1995.

[10] A.B. Pai and R.B. Kieburtz. Global context recovery: A new strategy for syntactic error recovery by table-driven parsers. *ACM Transactions on Programming Languages and Systems*, 2(1):18–41, 1980.

[11] T.J. Pennello and F.L. DeRemer. A forward move algorithm for LR error recovery. In *Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 241–254, 1978.

[12] B.A. Sheil. Observations on context-free grammars. In *Statistical Methods in Linguistics*, pages 71–109. Stockholm, Sweden, 1976.

[13] M. Vilares and B.A. Dion. Efficient incremental parsing for context-free languages. In *Proc. of the $5^{th}$ IEEE International Conference on Computer Languages*, pages 241–252, Toulouse, France, 1994.