A formal frame for robust parsing ⋆

M. Vilares [a], V.M. Darriba [a], J. Vilares [b], F.J. Ribadas [a]

[a] *Department of Computer Science, University of Vigo*
Campus As Lagoas s/n, 32004 Ourense, Spain

[b] *Department of Computer Science, University of A Coruña*
Campus de Elviña s/n, 15071 A Coruña, Spain

Abstract

A robust parser for context-free grammars, based on a dynamic programming architecture, is described. We integrate a regional error repair algorithm and a strategy to deal with incomplete sentences including unknown parts of unknown length. Experimental tests prove the validity of the approach, illustrating the perspectives for its application in real systems over a variety of different situations, as well as the causes underlying the computational behavior observed.

Key words: Robust parsing, dynamic programming, push-down automaton

1 Introduction

An ongoing question in the design of parsers is how to gain efficiency in dealing with unexpected input, and to do so without either over-generating or under-generating. This supposes the capacity to deal with gaps, incorrectness and noise contained in the input, which are often the consequence of external deterioration due to transcription errors and human performance deviations, typically in natural language processing, speech recognition or even traditional programming tasks. At this point, robustness should be conceived as the ability to handle non-standard input, and to interpret it in order to generate a plausible interpretation.

⋆ Research partially supported by the Spanish Government (project HF2002-0081, FPU grant AP2001-2545), the Autonomous Government of Galicia (projects PGIDIT02SIN01E and PGIDIT02PXIB3051PR) and the University of A Coruña.

Email addresses: vilares@uvigo.es (M. Vilares), darriba@ei.uvigo.es (V.M. Darriba), jvilares@udc.es (J. Vilares), ribadas@uvigo.es (F.J. Ribadas).

Our goal is syntactic, and no attention is devoted to ill-formed lexicons or semantic correction, focusing instead on enhancing robustness with respect to parse errors, incompleteness or deviations from standard language. To comply with these requests, we integrate an error repair algorithm [1] and a strategy to deal with incomplete sentences in a parser for context-free grammars (CFG's). We also provide a dynamic programming approach which allows us both to save in computational efficiency and to simplify the formal definition.

2 The standard parser

We parse a sentence $w_{1\ldots n} = w_1 \ldots w_n$ according to an unrestricted CFG $\mathcal{G} = (N, \Sigma, P, S)$, where N is the set of non-terminals, Σ the set of terminal symbols, P the rules and S the start symbol. The empty string is represented by ε. We generate from $\mathcal{G}$ a *push-down transducer* (PDA) for the language $\mathcal{L}(\mathcal{G})$. Although any shift-reduce strategy is adequate, we choose an LALR(1) device provided by ICE [1] [2], a generation environment for incremental parsers on CFG's. A PDA is 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, Σ the set of input symbols, Δ 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 δ a finite set of transitions $\delta(p, X, a) \ni (q, Y)$ with $p, q \in \mathcal{Q}$, $a \in \Sigma \cup \{\varepsilon\}$ and $X, Y \in \Delta \cup \{\varepsilon\}$. Let $\mathcal{A}$ 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 and $\alpha \in (\Sigma \cup N)^*$, and ax is the remaining input where the symbol a is the next to be shifted, $x \in \Sigma^*$. The application of $\delta(p, X, a) \ni (q, Y)$ results in a configuration $(q, Y\alpha, x)$ where a has been scanned, X has been popped, and Y has been pushed.

To get polynomial complexity, we avoid duplicating stack contents when ambiguity arises. Instead of storing all the information about a configuration, we determine the information we need to trace in order to retrieve it. This information is stored in a table $\mathcal{I}$ of *items*, $\mathcal{I} = \{[q, X, i, j],\ q \in \mathcal{Q},\ X \in \{\varepsilon\} \cup \{\nabla_{r,s}\},\ 0 \leq i \leq j\}$; where q is the current state, X is the top of the stack, and the positions i and j indicate the substring $w_{i+1} \ldots w_j$ spanned by the last terminal shifted to the stack or by the last production reduced. The symbol $\nabla_{r,s}$ indicates that the part $A_{r,s+1} \ldots A_{r,n_r}$ of a rule $A_{r,0} \rightarrow A_{r,1} \ldots A_{r,n_r}$ has been recognized.

We describe the parser using *parsing schemata* [3]; a triple $\langle \mathcal{I}, \mathcal{H}, \mathcal{D} \rangle$, with $\mathcal{I}$ the table of items previously defined, $\mathcal{H} = \{[a, i, i+1],\ a = w_{i+1}\}$ an initial set of triples called *hypotheses* that encodes the sentence to be parsed [2],

[1] For Incremental Context-free Environment.

[2] The empty string, ε, is represented by the empty set of hypotheses, $\emptyset$. An input string $w_{1\ldots n}$, $n \geq 1$ is represented by $\{[w_1, 0, 1],\ [w_2, 1, 2],\ \ldots,\ [w_n, n-1, n]\}$.

and $\mathcal{D}$ a set of *deduction steps* that allow new items to be derived from already known ones. Deduction steps are of the form $\{\eta_1, \ldots, \eta_k \vdash \xi \,/\, conds\}$, meaning that if all antecedents $\eta_i \in \mathcal{I}$ are present and the conditions *conds* are satisfied, then the consequent $\xi \in \mathcal{I}$ should be generated. In the case of ICE, $\mathcal{D} = \mathcal{D}^{\text{Init}} \cup \mathcal{D}^{\text{Shift}} \cup \mathcal{D}^{\text{Sel}} \cup \mathcal{D}^{\text{Red}} \cup \mathcal{D}^{\text{Head}}$, where:

$$\mathcal{D}^{\text{Shift}} = \{[q, \varepsilon, i, j] \vdash [q', \varepsilon, j, j+1] \;\Big/\; \begin{array}{l} \exists\, [a, j, j+1] \in \mathcal{H} \\ \mathit{shift}_{q'} \in \mathit{action}(q, a) \end{array} \}$$

$$\mathcal{D}^{\text{Sel}} = \{[q, \varepsilon, i, j] \vdash [q, \nabla_{r,n_r}, j, j] \;\Big/\; \begin{array}{l} \exists\, [a, j, j+1] \in \mathcal{H} \\ \mathit{reduce}_r \in \mathit{action}(q, a) \end{array} \}$$

$$\mathcal{D}^{\text{Red}} = \{[q, \nabla_{r,s}, k, j][q, \varepsilon, i, k] \vdash [q', \nabla_{r,s-1}, i, j] \,/\, q' \in \mathit{reveal}(q)\}$$

$$\mathcal{D}^{\text{Init}} = \{\vdash [q_0, \varepsilon, 0, 0]\} \qquad \mathcal{D}^{\text{Head}} = \{\, [q, \nabla_{r,0}, i, j] \vdash [q', \varepsilon, i, j] \,/\, q' \in \mathit{goto}(q, A_{r,0})\,\}$$

with $q_0 \in \mathcal{Q}$ the initial state, and *action* and *goto* entries in the PDA tables. We say that $q' \in \mathit{reveal}(q)$ iff $\exists\, Y \in N \cup \Sigma$ such that $\mathit{shift}_q \in \mathit{action}(q', Y)$ or $q \in \mathit{goto}(q', Y)$, that is, when there exists a transition from q' to q in $\mathcal{A}$. This set is equivalent to the dynamic interpretation of non-deterministic PDA's:

- A deduction step *Init* is in charge of starting the parsing process.
- A deduction step *Shift* corresponds to pushing a terminal a onto the top of the stack when the action to be performed is a shift to state q'.
- A step *Sel* corresponds to pushing the ∇_{r,n_r} symbol onto the top of the stack in order to start the reduction of a rule r.
- The reduction of a rule of length $n_r > 0$ is performed by a set of n_r steps *Red*, each of them corresponding to a pop transition replacing the two elements $\nabla_{r,s}\, X_{r,s}$ placed on the top of the stack by the element $\nabla_{r,s-1}$.
- The reduction of a rule r is finished by a step *Head* corresponding to a swap transition that recognizes the top element $\nabla_{r,0}$ as equivalent to the left-hand side $A_{r,0}$ of that rule, and performs the corresponding change of state.

These steps are applied until no further items can be generated. The splitting of reductions into a set of *Red* steps allows us to share computations, attaining a worst case time (resp. space) complexity $\mathcal{O}(n^3)$ (resp. $\mathcal{O}(n^2)$) with respect to the length n of the sentence [2]. The input is recognized iff the final item $[q_f, \nabla_{0,0}, 0, n+1],\ q_f \in \mathcal{Q}_f$, is generated.

3 The error repair strategy

Our next step is to extend the standard parser with an error repair strategy. Given that we choose to work using the technique and terminology described

in [1], we limit our description to the essential concepts.

3.1 The framework

Following Mauney and Fischer in [4], we talk about the *error* in the input to mean the difference between what was intended and what actually appears. We talk about the *point of error* as the point at which the difference occurs.

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

In order to locate the origin of the error at minimal cost, we 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 $w_{i'} \in detection(w_i)$.

The error is located in the left parse context, represented by the closest viable node, or in the right context, represented by the lookahead. Sometimes it can also be useful to isolate the parse branch in which the error appears.

Definition 3 *Let w_i be a point of error for $w_{1..n}$, we say that $[p, X, S_l^w, S_i^w]$ is an* error item *iff* $\exists\, a,\ \delta(p, \varepsilon, a) \neq \emptyset$. *We say that $[p, \varepsilon, S_{i'}^w, S_{i'}^w]$ is a* detection item *associated to w_i iff* $\exists\, a,\ \delta(p, A, a) \neq \emptyset$, *and A defining w_i, such that:*

$$\begin{array}{ll}
\delta(q_1, \varepsilon, w_{i'}) \ni (q_1, B_2), & \delta(q_1, B_2, w_{i'}) \ni (q_2, \varepsilon) \\
\vdots & \vdots \\
\delta(q_{n-1}, \varepsilon, w_{i'}) \ni (q_{n-1}, B_n), & \delta(q_{n-1}, B_n, w_{i'}) \ni (q_n, \varepsilon) \\
\delta(q_n, \varepsilon, w_{i'}) \ni (q_n, w_{i'}), & B_i \overset{+}{\Rightarrow} \varepsilon,\ \forall i \in [1, n]
\end{array}$$

The condition for error items implies that no scan action is possible for w_i. In the detection case, we disregard empty reductions which are not relevant here.

Definition 4 *A* modification M to a string *of length n, $w_{1..n} = w_1 \ldots w_n$, is a series of edit operations, $E_1 \ldots E_n E_{n+1}$, 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 M to w is written $M(w)$.*

We now restrict the modifications to focus on substrings, introducing the concept of error repair. We look for conditions that guarantee the ability to recover the parse from the error, while at the same time allowing us to isolate repair branches by using the concept of reduction. We are also interested in minimizing the 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 prefix in $\mathcal{L}(\mathcal{G})$, and $w \in \Sigma^*$, xw is not a prefix. We define a* repair of w following x *as $M(w)$, so that $\exists A \in N$ verifying:*

(1) $S \overset{+}{\Rightarrow} x_{1..i-1}A \overset{+}{\Rightarrow} x_{1..i-1}x_{i..m}M(w),\ i \leq m$ (3) $A \overset{*}{\Rightarrow} \gamma C\rho,\ \forall C \overset{+}{\Rightarrow} x_{i..m}M(w)$

(2) $B \overset{*}{\Rightarrow} \alpha A\beta,\ \forall B \overset{+}{\Rightarrow} x_{j..m}M(w),\ j < i$

We denote the set of repairs of w following x by $repair(x, w)$, and A by $scope(M)$, but this notion is not yet sufficient for our purposes. Our aim is to extend the repair to consider all points of detection associated to a given error, which implies considering different prefixes and repair zones.

Definition 6 *Let $e \in \Sigma$ be a point of error, we define the set of* repairs for e, *as $repair(e) = \{xM(w_{1..n}) \in repair(x, w_{1..n})/w_1 \in detection(e)\}$, where* detection$(e)$ *denotes the set of points of detection associated to e.*

We now need a mechanism to filter out undesirable repairs. To do so, we introduce criteria to only select those repairs with minimal cost.

Definition 7 *Let $I(a)$, $D(a)$, and $R(a)$ be positive insert, delete and replace costs for $a \in \Sigma$. The* cost of a modification $M(w_{1..n})$ *is given by* $\text{cost}(M(w_{1..n})) = \Sigma_{j \in J_\dashv} I(a_j) + \Sigma_{i=1}^{n}(\Sigma_{j \in J_i} I(a_j)) + D(w_i) + R(w_i)$, *where $\{a_j,\ j \in J_i\}$ is the set of insertions applied before w_i; and $\dashv$ the end of file.*

When several repairs are available on different points of detection, we need a condition to ensure that only those with minimal cost are considered.

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

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

It is also necessary to take into account the possibility of cascaded errors, that is, errors precipitated by a previous repair diagnosis. Prior 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:*

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

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.

Definition 10 *Let w_i be a 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* $\forall xM(y) \in viable(w_i, w_j)$, $\exists A \in N$ *defining* $w_{j'} \in detection(w_j)$ *such that* $A \stackrel{+}{\Rightarrow} \beta scope(M) \dots w_j$.

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.

3.2 The parsing scheme

To begin with, we assume that we are dealing with the first error detected. We extend the initial structure of items, as a quadruple $[p, X, i, j]$, with an error counter e; resulting in a new structure of the form $[p, X, i, j, e]$. Once the point of error w_i has been fixed, we can associate to it different points of detection $w_{i'_1}, \dots w_{i'_k}$. So, for each *error item*, defined from the fact that no action is possible from it when the lookahead is w_i, we investigate the list of its associated *detection items*; that is, those items representing the recognition of a terminal in the input string where we effectively locate the error. These detection items are located by using the back pointer, which indicates the input position where the last PDA action was applied. In practice, we recursively go back into its ancestors until we find the first descendant of the last node that would have had to be reduced if the lookahead had been correct. Once the detection items have been fixed, we apply the following steps:

$$\mathcal{D}^{\text{Shift}}_{\text{count}} = \{[q, \varepsilon, i, j, 0] \vdash [q', \varepsilon, j, j+1, 0] \;/\; \begin{array}{l} \exists [a, j, j+1] \in \mathcal{H} \\ shift_{q'} \in action(q, a) \end{array} \}$$

$$\mathcal{D}^{\text{Insert}}_{\text{error}} = \{[q, \varepsilon, i, j, 0] \vdash [q, \varepsilon, j, j, I(a)] \;/\; shift_{q'} \in action(q, a)\}$$

$$\mathcal{D}^{\text{Delete}}_{\text{error}} = \{[q, \varepsilon, i, j, 0] \vdash [q, \varepsilon, j, j+1, D(w_i)] \;/\; \exists [a, j, j+1] \in \mathcal{H}\}$$

$$\mathcal{D}^{\text{Replace}}_{\text{error}} = \{[q, \varepsilon, i, j, 0] \vdash [q, \epsilon, j, j+1, R(a)] \;/\; \begin{array}{l} \exists [b, j, j+1] \in \mathcal{H} \\ \exists shift_{q'} \in action(q, a),\ b \neq a \end{array} \}$$

This process continues until a repair applies a reduction verifying definition 5 covering both error and detection items and accepting a token in the remaining input string, as is shown in the left-hand-side of Fig. 1, where $[w_{i'''_1}, w_{i''_1}]$ delimits the scope of a repair detected at the point $w_{i'_1} \in detection(w_i)$. 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. Error counters are added at the time of reductions, even when error mode is finished:

$$\mathcal{D}^{\text{Sel}}_{\text{count}} = \{[q, \varepsilon, i, j, e] \vdash [q, \nabla_{r,n_r}, j, j, e] \;\Big/ \begin{array}{l} \exists\, [a, j, j+1] \in \mathcal{H} \\ reduce_r \in action(q, a) \end{array} \}$$

$$\mathcal{D}^{\text{Red}}_{\text{count}} = \{[q, \nabla_{r,s}, k, j, e][q', \varepsilon, i, k, e'] \vdash [q', \nabla_{r,s-1}, i, j, e + e'] \,/ q' \in reveal(q)\}$$

$$\mathcal{D}^{\text{Head}}_{\text{count}} = \{\, [q, \nabla_{r,0}, i, j, e] \vdash [q', \varepsilon, i, j, e] \,/ q' \in goto(q, A_{r,0})\,\}$$

We apply a principle of optimization, saving only those items with minimal counters for computation purposes.

When the current repair is not the first one, it can modify a previous repair in order to avoid cascaded repairs by adding the cost of the new error hypotheses, in order to profit from the experience gained from previous ones. 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. 1 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 $A_{i'_1}$.

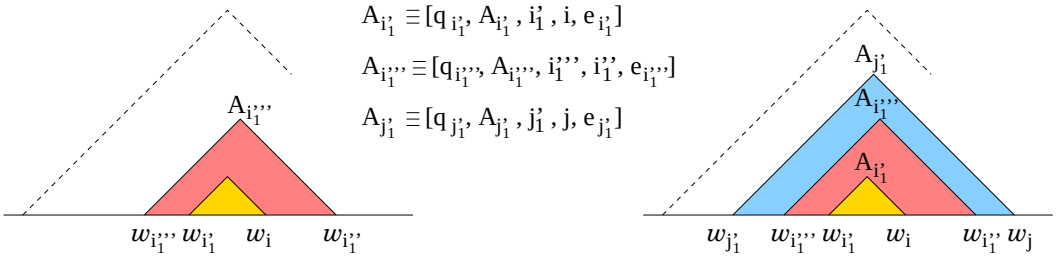

Fig. 1. Dealing with precipitated errors

Regional repairs have two properties. First, they are independent of the shift-reduce parser used. The second one is a consequence of the lemma below.

Lemma 11 *(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*

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

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

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

Given that $\mathit{scope}(M)$ 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 $\mathit{viable}(w_i, w_j)$. □

Corollary 12 *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*

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

PROOF.

It immediately follows from lemma 11. □

This allows us to get an asymptotic behavior close to global repair methods. This property has profound implications for the efficiency, as measured by time and space taken, the simplicity and the power of computing regional repairs.

Lemma 13 *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 12. □

3.3 Previous works

Error repair methods can be classified into local, global and regional strategies. Local repair algorithms [5,6] make modifications to the input so that at least one more original input symbol can be accepted by the parser. There are cases, however, in which their simplicity causes them to choose a poor repair.

Global algorithms [7] examine the entire program and make a minimum of changes to repair all the errors. Global methods give the best repairs possible, but they are not efficient. Since they expend equal effort on all parts of the program, including areas that contain no errors, much of that effort is wasted. Finally, the main problem to be dealt with in regional approaches [4] is how to determine the extent of the repair in order to avoid cascaded errors.

In between the local and global methods, *regional repair* algorithms fix a portion of the program including the error and as many additional symbols as needed to assure a good repair. Our proposal is a least-cost regional strategy, asymptotically equivalent to global repair ones. That is, in the worst case, space and time complexity are the same as those attained for global repairs and, in the best case, are the same as for local ones. The repair quality is equivalent to global approaches. Compared to other regional algorithms [4], we provide a least-cost dynamic estimation of this region, which is an advantage in the design of interactive tools, where efficiency is a priority challenge.

4 Parsing incomplete sentences

In order to handle incomplete sentences, we introduce two symbols, "?" stands for one unknown word, and "$*$" stands for an unknown sequence of words.

4.1 The framework

The problem can be stated in similar terms to error repair, with some restrictions: points of detection are reduced to the point of error, modifications are insertions and no variable defines the scope of the repair. Here, the stop of the parser can only be caused by the presence of unknown words, not by the inclusion of errors in the portion of input already parsed. It is therefore more appropriate to talk about *point of stop* associated to an unknown symbol.

Definition 14 *Let $w_{1..n}$ be an incomplete sentence, we say that w_i is a* point of stop *iff $w_i =?$ or $w_i = *$*

On the other hand, the parse may not only be completed from insertions, but it must be recovered using exclusively this kind of hypotheses. Finally, the parser must not continue to introduce insertions once the process is able to connect with the right context. Otherwise, we would be altering the original input, which is in contradiction with the initial hypothesis of correctness. So, the consideration of the scope of a modification here cannot be the same as in error repair, defined in the latter case in terms of grammatical reductions.

Definition 15 *Let $w_{1..n}$ be an incomplete sentence in $\mathcal{L}(\mathcal{G})$, and w_i a point of stop. We define a* recovery of w in the position i *as a modification $M(w_i)$ given by a sequence of insertions before w_i followed by the deletion of w_i, so that $S \stackrel{+}{\Rightarrow} w_{1..i-1}w_i\alpha \stackrel{+}{\Rightarrow} w_{1..i-1}M(w_i)\alpha$. We denote the set of recoveries of w in the position i by* recovery(w, i)*, and $M(w_i)$ by* scope(M)*.*

To filter out undesirable parses in order to reduce the complexity, we re-take the insertion costs, $I(a),\ a \in \Sigma$; selecting only those recoveries with minimal cost. So, the cost of a modification in a recovery process will here be given by $cost(M(w_i)) = \Sigma_{i=1}^{n}(\Sigma_{j \in J_i} I(a_j)) + D(w_i)$, where in this case $D(w_i) = 0$.

When several recoveries are available, we need a condition to ensure that only those with the same minimal cost are considered.

Definition 16 *Let $w_{1..n}$ be an incomplete sentence, and let w_i be a point of stop in w, we define the set of* completions for w in i*, as follows:*

$$completions(w_i) = \{M \in recovery(w_i) / cost(M) \leq cost(M'),\ \forall M' \in recovery(w_i)\}$$

In contrast to error repair, it is not now possible to consider the overlapping of recovery processes corresponding to different points of stop in an incomplete sentence. Thus, it makes no sense to talk about precipitated recoveries.

4.2 The parsing scheme

Once the parser detects that the next input symbol is a point of stop, we apply the set of rules $\mathcal{D}_{\text{incomplete}}$, which includes the following two sets of deduction steps, as well as $\mathcal{D}_{\text{count}}^{\text{Shift}}$ previously defined:

$$\mathcal{D}_{\text{incomplete}}^{\text{Shift}} = \{[q, \varepsilon, i, j] \vdash [q', \varepsilon, j, j+1] \;/\; \begin{array}{l} \exists\, [?, j, j+1] \in \mathcal{H} \\ shift_{q'} \in action(q, a)\ \} \\ a \in \Sigma \end{array}$$

$$\mathcal{D}_{\text{incomplete}}^{\text{Loop\_shift}} = \{[q, \varepsilon, i, j] \vdash [q', \varepsilon, j, j] \;/\; \begin{array}{l} \exists\, [*, j, j+1] \in \mathcal{H} \\ shift_{q'} \in action(q, X)\ \} \\ X \in N \cup \Sigma \end{array}$$

From an intuitive point of view, $\mathcal{D}_{\text{incomplete}}^{\text{Shift}}$ applies any shift transition independently of the current lookahead available, provided that this transition is applicable with respect to the PDA configuration and that the next input symbol is an unknown token. In relation to $\mathcal{D}_{\text{incomplete}}^{\text{Loop\_shift}}$, it applies to items

corresponding to PDA configurations for which the next input symbol denotes an unknown sequence of tokens, any valid shift action on terminals or variables. Given that in this latter case new items are created in the same starting itemset, shift transitions may be applied any number of times to the same computation thread, without scanning the input string.

All deduction steps are applied until every parse branch links up to the right-context by using a shift action, resuming the standard parse mode and defining a recovery for the current point of stop. In this process, when we deal with sequences of unknown tokens, we can generate nodes deriving only "$*$" symbols. This over-generation is of no interest in most practical applications and introduces additional computational work.

We are interested in generating completions rather than simple recoveries. Our aim is to replace those variables with the unknown subsequence terminal, "$*$". To solve this problem, we re-take the counters introduced in error mode, in order to tabulate the number of categories used to rebuild the noisy sentence. The final goal is to select an optimal reconstruction. Therefore, it makes no sense to differentiate between counter contributions due to the application of one or another parsing mechanism. When several items representing the same node are generated, only those with minimal counter are saved. Formally, we redefine the set of deduction steps as follows:

$$\mathcal{D}^{\text{Shift}}_{\text{incomplete}} = \{[q,\varepsilon,i,j,e] \vdash [q',\varepsilon,j,j+1,e+I(a)] \;\Big/\; \begin{array}{l} \exists\, [?,j,j+1] \in \mathcal{H} \\ \mathit{shift}_{q'} \in \mathit{action}(q,a)\ \} \\ a \in \Sigma \end{array}$$

$$\mathcal{D}^{\text{Loop\_shift}}_{\text{incomplete}} = \{[q,\varepsilon,i,j,e] \vdash [q',\varepsilon,j,j,e+I(X)] \;\Big/\; \begin{array}{l} \exists\, [*,j,j+1] \in \mathcal{H} \\ \mathit{shift}_{q'} \in \mathit{action}(q,X)\ \} \\ X \in N \cup \Sigma \end{array}$$

where $I(X)$ is the insertion cost for $X \in N \cup \Sigma$, and we maintain the definition domain previously considered for $\mathcal{D}^{\text{Red}}_{\text{count}}$, $\mathcal{D}^{\text{Sel}}_{\text{count}}$ and $\mathcal{D}^{\text{Head}}_{\text{count}}$. The incomplete sentence is recognized iff $[q_f, \nabla_{0,0}, 0, n+1, e]$, $q_f \in \mathcal{Q}_f$, is generated.

Lemma 17 *Let $w_{1..n}$ be an incomplete sentence with a point of stop in w_i, $i \in [1,n]$, then the time and space bounds for the completion algorithm are $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$, in the worst case, respectively.*

PROOF.

It follows from the complexity of the standard parse scheme. □

Previous proposals, such as Tomita *et al.* [8] and Lang [9], also apply dynamic programming, although the approach is different in each case. Lang introduces items as fragments of the PDA computations that are independent of the initial content of the stack, except for its two top elements. This relies on the concept of *dynamic frame* for CFG's [2] and, in particular, to the dynamic frame S^2. Tomita *et al.* use a shared-graph based structure to represent the stack forest. We work in a dynamic frame S^1, which means that items only represent the top of the stack. This results in improved sharing for both syntactic structures and computations.

In relation to the parsing scheme, Lang separates the execution strategy from the implementation of the interpreter, while Tomita *et al.*'s work can be interpreted simply as a specification of Lang's for LR(0) PDA's. We consider a LALR(1) scheme, which facilitates lookahead computation, whilst the state splitting phenomenon remains reasonable. This enables us to achieve high levels of sharing and efficiency as well as to increase the deterministic domain.

Neither Lang nor Tomita *et al.* avoid over-generation in nodes deriving only "$*$" symbols. Only Lang includes an additional phase to eliminate these nodes from the output parse shared forest. We solve both the consideration of an extra simplification phase and the over-generation on unknown sequences by considering the same principle of optimization applied on error counters.

5 The robust parser

In order to favor understanding, we differentiate two kinds of parse steps. We talk about *extensional steps* when they include conditions over shift actions in standard parsing mode, and we talk about *intensional steps* in any other case, i.e. when they are related to reduce actions in the kernel. Whichever is the case, the robust mode must guarantee the capacity to recover the parser from any unexpected situation derived from either gaps in the scanner or errors. To deal with this, it is sufficient to combine the deduction steps previously introduced. More exactly, we have that the extensional steps are defined by:

$$\mathcal{D}^{\text{Init}} \cup \mathcal{D}^{\text{Shift}}_{\text{count}} \cup \mathcal{D}^{\text{Insert}}_{\text{error}} \cup \mathcal{D}^{\text{Delete}}_{\text{error}} \cup \mathcal{D}^{\text{Replace}}_{\text{error}} \cup \mathcal{D}^{\text{Shift}}_{\text{incomplete}} \cup \mathcal{D}^{\text{Loop\_shift}}_{\text{incomplete}}$$

and the intensional ones by $\mathcal{D}^{\text{Red}}_{\text{count}} \cup \mathcal{D}^{\text{Sel}}_{\text{count}} \cup \mathcal{D}^{\text{Head}}_{\text{count}}$

where there is no overlapping between the deduction subsets. In effect, in relation to the extensional case, no collision is possible because the steps

in question are distinguished by conditions over the lookahead. For the intensional case, the steps remain invariable from the beginning, when we defined the standard parser. The ill-formed input string is recognized iff the final item $[q_f, \nabla_{0,0}, 0, n+1, e]$, $q_f \in \mathcal{Q}_f$, is generated.

Lemma 18 *Let $w_{1..n}$ be an incomplete sentence, then the time and space bounds for the robust parsing algorithm are $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$, in the worst case, respectively.*

PROOF.

It follows from lemmas 13 and 17. □

6 Experimental results

We consider the language, $\mathcal{L}$, of arithmetic expressions to illustrate our discussion, comparing the standard parsing on ICE [2], with the consideration of full robust parsing. We introduce two grammars, $\mathcal{G}_L$ and $\mathcal{G}_R$:

$\mathcal{G}_L$: E → E + T | T
T → (E) | number

$\mathcal{G}_R$: E → T + E | T
T → (E) | number

to generate the running language, $\mathcal{L}$. As a consequence, parses are built from the left-associative (resp. right-associative) interpretation for $\mathcal{G}_L$ (resp. $\mathcal{G}_R$), which allows us to estimate the impact of traversal orientation in the parse process. Our goal now is essentially descriptive, in order to illustrate the recovery mechanism and its behavior in a variety of situations. In this context, our example combines structural simplicity and topological complexity in a language which is universally known. In the same sense, larger languages do not provide extra criteria to be considered.

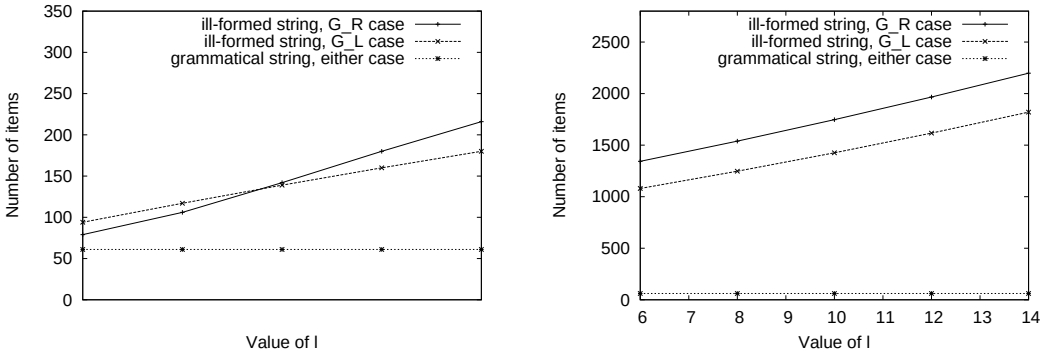

Fig. 2. Items generated for *unknown* and *error correction* examples

In this sense, we shall consider four different patterns to model ill-formed input strings. The first, that we shall call *error-correction*, is of the form

$$b_1 + \ldots + b_{i-1} + (b_i + \ldots + \\ (b_{[n/3]} + b_{[n/3]+1} b_{[n/3]+2} + \ldots + b_\ell b_{\ell+1} + b_{\ell+2} + \ldots + b_n$$

The second, that we shall call *unknown*, is of the form

$$b_1 + \ldots + b_{i-1} + (b_i + \ldots + \\ (b_{[n/3]} + b_{[n/3]+1} * + b_{[n/3]+3} * + \ldots + b_\ell * + b_{\ell+2} + \ldots + b_n$$

The third pattern, that we shall call *total overlapping*, is of the form

$$b_1 + \ldots + b_{i-1} + (b_i + \ldots + \\ (b_{[n/3]} + * b_{[n/3]+1} b_{[n/3]+2} + \ldots + * b_\ell b_{\ell+1} + b_{\ell+2} + \ldots + b_n$$

The last pattern, that we shall call *partial overlapping*, is of the form

$$b_1 + \ldots + b_{i-1} + (b_i + \ldots + (b_{[n/3]} + b_{[n/3]+1} b_{[n/3]+2} + \ldots + b_\ell b_{\ell+1} * b_{\ell+2} \ldots * b_n$$

where $i \in \{[n/3], \ldots, 1\}$ and $\ell = 3[n/3] - 2i + 1$, with $[n/3]$ being the integer part of $n/3$.

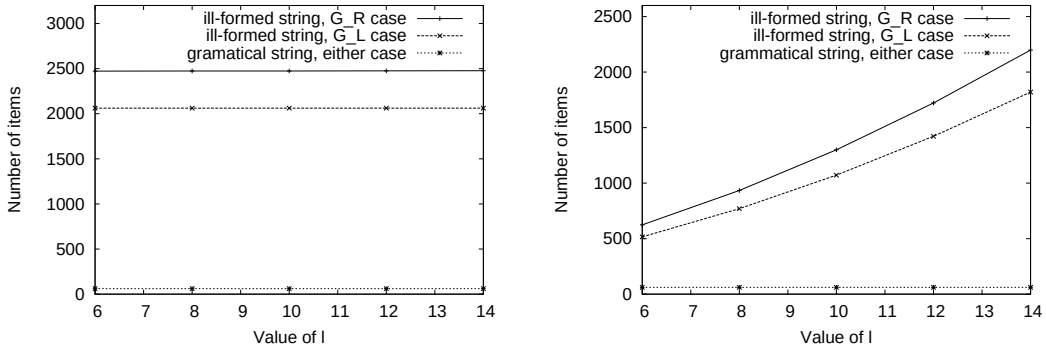

Fig. 3. Items generated for *total* and *partial overlapping* examples

These examples seek to illustrate the variety of situations to be dealt with in robust parsing. The *unknown* example only requires the treatment of unknown sequences, while the *error-correction* example only applies the error repair strategy. The *total overlapping* example forces the system to apply both unknown sequences recognition and error repair, although only the error recovery mechanisms are finally taken into account. Finally, the *partial*

overlapping example also combines the recognition of unknown sequences and error repair, but in this case both strategies have an active role.

In the case of *unknown* pattern, the set of minimal cost robust parse process includes the sentences obtained by inserting closed brackets in the positions indicated by the unknown sequence. In other patterns, the set of minimal cost robust parse process is formed by the sentences obtained by replacing tokens $b_{[n/3]+2k}$ with $k \in \{1, \ldots, [n/3] - i + 1\}$ by closed brackets. As a consequence, one minimal cost parse alternative is given by "$b_1 + \ldots + b_{i-1} + (b_i + \ldots + (b_{[n/3]} + b_{[n/3]+1}) + \ldots + b_\ell) + b_{\ell+2} + \ldots + b_n$"; whose parse cost we shall use as reference to illustrate the practical complexity of our proposal in these tests.

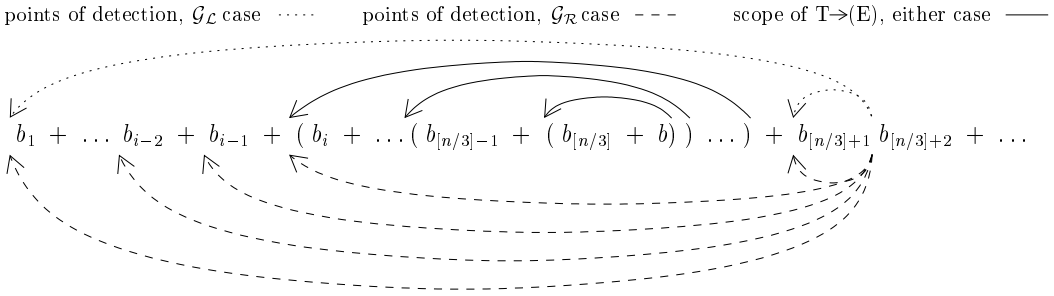

Fig. 4. Error detection points for *total overlapping* example

The results for the *unknown* and *error-correction* examples are shown in Fig. 2, in the left and the right-hand-side of the figure respectively. In the case of *total* and *partial overlapping* examples, the tests are shown in the left and the right-hand-side of Fig. 3. The results are provided for $\mathcal{G}_L$ and $\mathcal{G}_R$, with the number of items generated being taken as a reference for appreciating the efficiency, rather than temporal criteria, which are more dependent on implementation. These items are measured in relation to the position, ℓ, of the addend "b_ℓ" in the input, around which all the tests have been structured.

The first detail to note is the null slopes in the graphs of the *total overlapping* example, while for all the others the slopes are ascendent. This is due to the particular distribution of the zone where the robust parse operates. In effect, as is shown in Fig. 4, the error detection points from the very first point of error in the input string locate the beginning of the error correction zone [1] at the addend "b_1". In practice, as part of the more general robust parse process, the error correction strategy already covers all the input string, although only in the case of $\mathcal{G}_R$ does the error repair scope extend to global context. This apparent contradiction in the case of $\mathcal{G}_L$ is due to the fact that although the effective repair mechanisms do not have a global scope, most unsuccessful repairs are only rejected at the end of the robust parse process. As a consequence, for both grammars in this example the correction mechanisms are applied on all the input positions, and the location of "b_ℓ" has no influence on the number of items generated, as can be seen in Fig. 3. This is also illustrated in Fig. 5, representing on its left-hand-side the increase in the size

of the repair scope for both *error correction* and *partial overlapping* examples, and on its right-hand-side the same information for the *total overlapping* case.

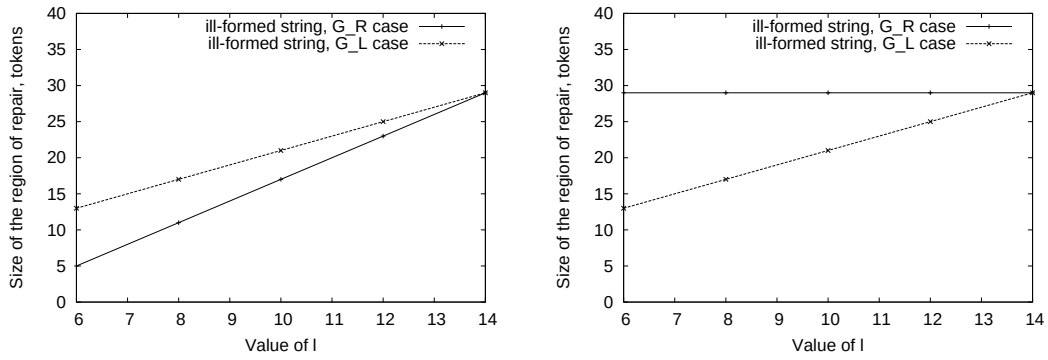

Fig. 5. Repair scope for *error correction*, *partial* and *total overlapping* examples

The situation is different in the *error correction* and *partial overlapping* examples, for which the size of the error repair zone increases with the position of "b_ℓ", as is shown in Fig. 6. In this sense, the figure illustrates both the dependence of the error repair region on the grammar used, and the asymptotic behavior of the error repair strategy [1] in dealing with cascaded errors.

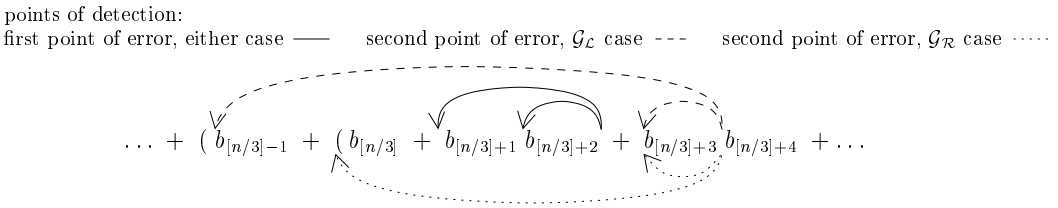

Fig. 6. Error detection points for *error correction* and *partial overlapping* examples

In relation to complexity, although the theoretical cost is the same for both the error repair strategy and the treatment of unknown sentences, in practice these tests show that the greater weight is due to error repair. This is illustrated by the results displayed for the *error correction* and the two *overlapping* examples on the right-hand-sides of Fig. 2 and Fig. 3, respectively. In effect, these results show that the number of items generated is appreciably larger in these cases, in contrast to the work developed for the *unknown* example, which we can see in the left-hand-side of Fig. 2, and for which no error repair process is applied. From an operational point of view, this behavior is a consequence of the contextual treatment in each case. So, the parse of unknown sequences only generates, for each symbol $*$, items in the current itemset. However, in the case of error repair the scope depends, for each error, on the grammatical structure and can range from one to the total collection of itemsets, as is shown in Figs. 4 and 6. Whichever is the case, the smoothness of the slopes proves the computational efficiency of our proposal.

7 Conclusions

Robust parsing is a central task in the design of dialogue systems, where the deterioration of the signal, and the presence of under-generation or over-generation phenomena due to covering grammatical problems make it difficult to perform continuous unrestricted language recognition. In this sense, robust parsing seeks to find interpretations that have maximal thresholds. Our proposal provides the capacity to efficiently recover the system from external syntactic factors or user errors. We concentrate on enhancing robustness by using the mechanisms offered by dynamic programming in order to improve performance and provide a formal parse definition. In contrast to previous works, we solve both extra simplification phases and the over-generation phenomena associated with the recognition of unknown sequences of unknown length. We also avoid distortions due to cascaded errors by integrating a regional repair mechanism that, in contrast to global approaches, limits the recovery effort to those areas in the input that contain errors.

References

[1] M. Vilares, V. Darriba, F. Ribadas, Regional least-cost error repair, in: S. Yu, A. Păun (Eds.), Implementation and Application of Automata, Vol. 2088 of LNCS, Springer-Verlag, Berlin-Heidelberg-New York, 2001, pp. 293–301.

[2] M. Vilares, Efficient incremental parsing for context-free languages, Ph.D. thesis, University of Nice. ISBN 2-7261-0768-0, France (1992).

[3] K. Sikkel, Parsing schemata, Ph.D. thesis, Univ. of Twente, The Netherlands (1993).

[4] J. Mauney, C. Fischer, Determining the extend of lookahead in syntactic error repair, ACM TOPLAS 10 (3) (1988) 456–469.

[5] S. Graham, C. Haley, W. Joy, Practical LR error recovery, in: Proc. of the SIGPLAN 79 Symposium on Compiler Construction, ACM, 1979, pp. 168–175.

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

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

[8] M. Tomita, H. Saito, Parsing noisy sentences, in: COLING'88, Budapest, Hungary, 1988, pp. 561–566.

[9] B. Lang, Parsing incomplete sentences, in: D. V. (ed.) (Ed.), COLING'88, Budapest, Hungary, 1988, pp. 365–371, vol. 1.