Robust parsing using dynamic programming*

M. Vilares[1], V.M. Darriba[1], J. Vilares[2], and L. Rodríguez[1]

[1] Department of Computer Science, University of Vigo
Campus As Lagoas s/n, 32004 Ourense, Spain
{vilares,darriba,leandro}@uvigo.es
[2] Department of Computer Science, University of A Coruña
Campus de Elviña s/n, 15071 A Coruña, Spain
jvilares@udc.es

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.

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.

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 [10] and a strategy to deal with incomplete sentences, in a parser for context-free grammars (CFG's). In this sense, we provide a dynamic programming approach which allows us both to save in computational efficiency and to simplify the formal definition framework.

2 The standard parser

Our aim is to 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

* Research partially supported by the Spanish Government under projects TIC2000-0370-C02-01 and HP2002-0081, and the Autonomous Government of Galicia under projects PGIDT01PXI10506PN, PGIDIT02PXIB30501PR and PGIDIT02SIN01E.

terminal symbols, P the rules and S the start symbol. The empty string will be represented by ε. We generate from $\mathcal{G}$ a *push-down transducer* (PDA) for the language $\mathcal{L}(\mathcal{G})$. In practice, we choose an LALR(1) [1] device generated by ICE[1] [9], although any shift-reduce strategy is adequate. 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 of the form $\delta(p, X, a) \ni (q, Y)$ with $p, q \in \mathcal{Q}$, $a \in \Sigma \cup \{\varepsilon\}$ and $X, Y \in \Delta \cup \{\varepsilon\}$. Let 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, 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 new 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* [7]; 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], 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 [1]. We say that $q' \in$ *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.

[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 $\{[w_1, 0, 1],\ [w_2, 1, 2],\ \ldots,\ [w_n, n-1, n]\}$.

– 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 corresponding to partial reductions, 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 [9]. The sentence 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 [10], we limit our description to the essential concepts; referring the reader to the original paper in order to get more details.

3.1 The parsing scheme

Following Mauney and Fischer in [5], 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. In this context, a *repair* should be understood as a modification on the input string allowing the parse both, to recover the standard process and to avoid the phenomenon of cascaded errors, that is, errors precipitated by a previous erroneous repair diagnostic. This is, precisely, the goal of the notion of *regional repair* defined in [10], which we succinctly introduce now.

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 $[p, X, i, j, e]$. 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. So, 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 deduction steps from them:

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

where $I(a)$, $D(a)$ and $R(a)$ are the costs for insertion, deletion and replacement of $a \in \Sigma$, respectively. This process continues until a repair applies a reduction covering both error and detection items. Once this has been performed on each detection item, we select the least-cost repairs and the parser goes back to standard mode. Error counters are added at the time of reductions, even when error mode is finished:

$$\mathcal{D}^{\text{Sel}}_{\text{count}} = \{[q, \varepsilon, i, j, e] \vdash [q, \nabla_{r,n_r}, j, j, e] \;\Big/ \begin{array}{l} \exists\; [a, j, j+1] \in \mathcal{H} \\ reduce_r \in action(q, a) \end{array} \}$$
$$\mathcal{D}^{\text{Red}}_{\text{count}} = \{[q, \nabla_{r,s}, k, j, e][q', \varepsilon, i, k, e'] \vdash [q', \nabla_{r,s-1}, i, j, e+e'] \;/ q' \in reveal(q)\}$$
$$\mathcal{D}^{\text{Head}}_{\text{count}} = \{\, [q, \nabla_{r,0}, i, j, e] \vdash [q', \varepsilon, i, j, e] \;/ q' \in goto(q, A_{r,0})\,\}$$

To avoid the generation of items only differentiated by the error counter we shall apply a principle of optimization, saving only for computation purposes those with minimal error counters e.

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 to profit from the experience gained from previous ones. This allows us to get, in a simple manner, an asymptotic behavior close to global repair methods [10]. As a consequence, although time (resp. space) complexity is, in the worst case, $\mathcal{O}(n^3)$ (resp. $\mathcal{O}(n^2)$), in practice it is considerably more reduced as we shall see in our experimental tests. The 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.

3.2 Previous works

Error recovery methods can be classified into local, global and regional strategies. Local repair algorithms [2, 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 [4] 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 [5] is how to determine the extent of the repair in order to avoid cascaded errors.

Our proposal is a least-cost regional error repair 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. In relation to local ones, no fixed repair region is considered. So, we avoid both wasted effort when it is excessive and the generation of cascaded errors when it is not sufficient. Compared to other regional algorithms [5], 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 extend the input alphabet. We introduce two symbols, "?" stands for one unknown word symbol, and "$*$" stands for an unknown sequence of input word symbols.

4.1 The parsing scheme

Once the parser detects that the next input symbol to be shifted is one of these two extra symbols, we apply the set of deduction steps $\mathcal{D}_{\text{incomplete}}$, which includes the following two sets of deduction steps, as well as $\mathcal{D}^{\text{Shift}}_{\text{count}}$ previously defined:

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

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

From an intuitive point of view, $\mathcal{D}^{\text{Shift}}_{\text{incomplete}}$ 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 $\mathcal{D}^{\text{Loop\_shift}}_{\text{incomplete}}$, 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. 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, which supposes an extra loss of parse efficiency. So, our goal is to replace these variables with the unknown subsequence terminal, "$*$". We solve this problem by re-taking the counters introduced in error repair mode, in order to tabulate the number of categories used to rebuild the noisy sentence. In effect, our final goal is to select, for a given ill-formed input, an optimal reconstruction. As a consequence, 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.

4.2 Previous works

Previous proposals, such as Tomita *et al.* [8] and Lang [3], also apply dynamic programming to deal with unknown sequences in order to reduce space and time complexity, although the approach is different in each case. From an operational point of view, 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 [9] 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 applied, 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 in the error repair process.

5 The robust parser

We are now ready to introduce the robust parsing construction. In order to favor understanding, we differentiate two kinds of parse steps. So, we talk about *extensional steps* when we deal with deduction steps including 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 final robust parse has a time (resp. space) complexity, in the worst case, $\mathcal{O}(n^3)$ (resp. $\mathcal{O}(n^2)$) with respect to the length n of the ill-formed sentence. The 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.

6 Experimental results

We consider the language, $\mathcal{L}$, of arithmetic expressions to illustrate our discussion, comparing the standard parsing on Ice [9], with the consideration of full robust parsing. We introduce two grammars, $\mathcal{G}_L$ and $\mathcal{G}_R$, given by the rules:

$$\begin{array}{llll} \mathcal{G}_L\colon & \text{E} \rightarrow \text{E} + \text{T} \mid \text{T} & \qquad \mathcal{G}_R\colon & \text{E} \rightarrow \text{T} + \text{E} \mid \text{T} \\ & \text{T} \rightarrow (\text{E}) \mid \text{number} & & \text{T} \rightarrow (\text{E}) \mid \text{number} \end{array}$$

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 mechanisms and its behavior in a variety of situations. In this context, our example joins 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.

In this sense, we shall consider four different patterns to model ill-formed input strings. The former, 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

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

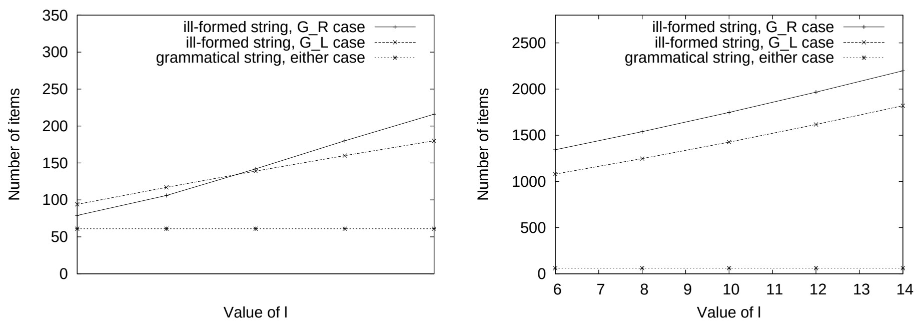


Fig. 1. Items generated for the *unknown* and *error correction* examples

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 $n/3$.

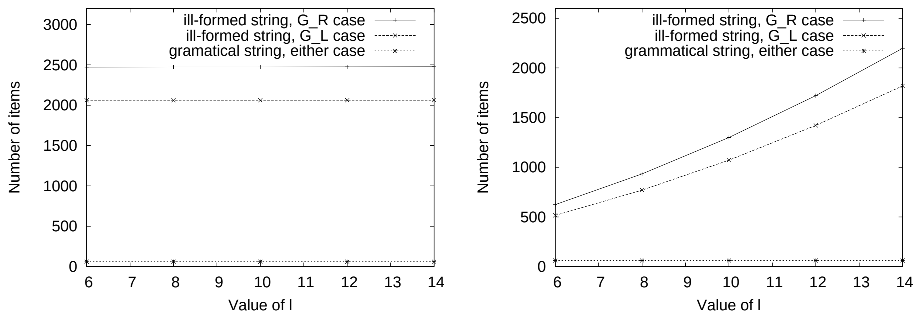
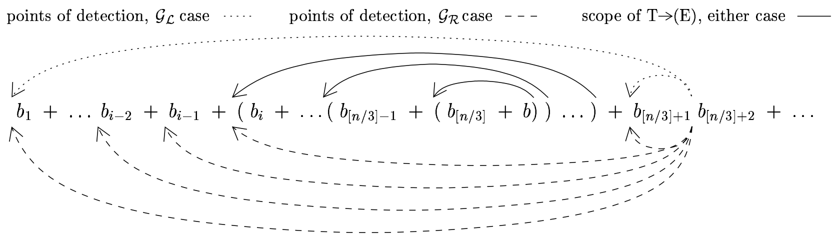


Fig. 2. Items generated for the *total* and *partial overlapping* examples

Our running examples seek to illustrate the variety of situations to be dealt with in robust parsing. So, 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 parse recovery process.

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 $k \in \{1, \dots, [n/3] - i + 1\}$ by closed brackets. As a consequence, one minimal cost parse alternative is given by "$b_1 + \dots + b_{i-1} + (b_i + \dots + (b_{[n/3]} + b_{[n/3]+1}) + \dots + b_\ell) + b_{\ell+2} + \dots + b_n$"; whose parse cost we shall use as reference to illustrate the practical complexity of our proposal in these experimental tests.



points of detection, $\mathcal{G}_\mathcal{L}$ case
points of detection, $\mathcal{G}_\mathcal{R}$ case
scope of T->(E), either case
b_1 + ... b_{i-2} + b_{i-1} + (b_i + ... ($b_{[n/3]-1}$ + ($b_{[n/3]}$ + b)) ...) + $b_{[n/3]+1}$ $b_{[n/3]+2}$ + ...

Fig. 3. Error detection points for the *total overlapping* example

The results are shown, for the *unknown* and *error-correction* examples, in Fig. 1, 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. 2. In all cases, the results are provided for both grammars $\mathcal{G}_L$ and $\mathcal{G}_R$, with the number of items generated by the system during the parse process being taken as a reference for appreciating the efficiency of our method, rather than purely temporal criteria, which are more dependent on its implementation. These items are measured in relation to the position, ℓ, of the addend "b_ℓ" in the input string, 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 we apply the robust parse mechanisms. In effect, as is shown in Fig. 3, the error detection points from the very first point of error in the input string locate the beginning of the error correction zone [10] 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. 2. This is also illustrated in Fig. 4, 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.

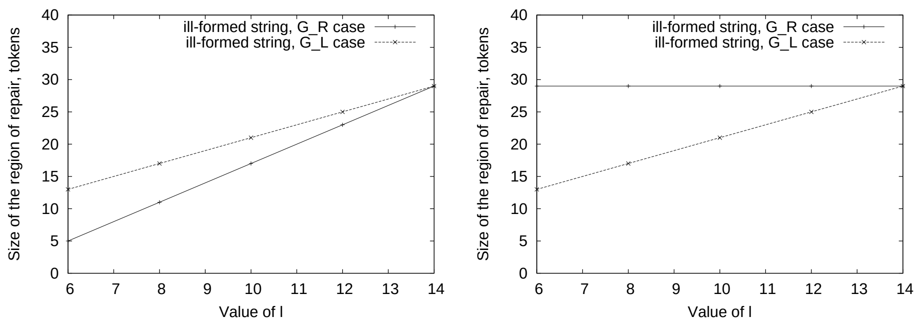


Fig. 4. Error repair scope for the *error correction* and *overlapping* examples

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. 5. 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 [10] in dealing with cascaded errors.

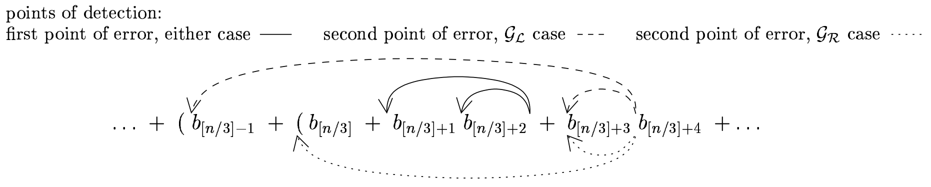

points of detection:
first point of error, either case
second point of error, $\mathcal{G}_\mathcal{L}$ case
second point of error, $\mathcal{G}_\mathcal{R}$ case
$\ldots + (\ b_{[n/3]-1} + (\ b_{[n/3]} + b_{[n/3]+1}\ b_{[n/3]+2} + b_{[n/3]+3}\ b_{[n/3]+4} + \ldots$

Fig. 5. Error detection points for the *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. 1 and Fig. 2, respectively. In effect, these results show that the amount 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. 1, 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. 3 and 5. 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 apparently impossible to perform continuous unrestricted language recognition.

In this sense, robust parsing seeks to find those interpretations that have maximal thresholds. Our proposal provides the capacity to recover the system from external syntactic factors or user errors; and also the possibility to do so efficiently. Here, our work concentrates on enhancing robustness by using the mechanisms offered by dynamic programming in order to improve performance and provide an optimal quality in the parse definition.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, U.S.A., 1986.
2. S.L. Graham, C.B. Haley, and W.N. Joy. Practical LR error recovery. In *Proc. of the SIGPLAN 79 Symposium on Compiler Construction*, pages 168–175. ACM, August 1979.
3. B. Lang. Parsing incomplete sentences. In D. Vargha (ed.), editor, *COLING'88*, pages 365–371, Budapest, Hungary, 1988. vol. 1.
4. G. Lyon. Syntax-directed least-errors analysis for context-free languages: A practical approach. *Communications of the ACM*, 17(1):3–14, 1974.
5. J. Mauney and C.N. Fischer. Determining the extend of lookahead in syntactic error repair. *ACM TOPLAS*, 10(3):456–469, 1988.
6. 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.
7. K. Sikkel. *Parsing Schemata.* PhD thesis, Univ. of Twente, The Netherlands, 1993.
8. M. Tomita and H. Saito. Parsing noisy sentences. In *COLING'88*, pages 561–566, Budapest, Hungary, 1988.
9. M. Vilares. *Efficient Incremental Parsing for Context-Free Languages.* PhD thesis, University of Nice. ISBN 2-7261-0768-0, France, 1992.
10. M. Vilares, V.M. Darriba, and F.J. Ribadas. Regional least-cost error repair. In S. Yu and A. Păun, editors, *Implementation and Application of Automata*, volume 2088 of *LNCS*, pages 293–301. Springer-Verlag, Berlin-Heidelberg-New York, 2001.