Continuous Grammars

MARTIN RUCKERT
ruckertm@acm.org
State University of New York at New Paltz

Abstract

After defining appropriate metrics on strings and parse trees, the classic definition of continuity is adapted and applied to functions from strings to parse trees. Grammars that yield continuous mappings are of special interest, because they provide a sound theoretical framework for syntax error correction. Continuity justifies the approach, taken by many error correctors, to use the function output (the parse tree), and all the additional information it provides, in order to find corrections to the function input (the string). We prove that all Bounded Context grammars are continuous and that all continuous grammars are Bounded Context Parseable grammars, giving a characterization of continuous grammars in terms of possible parsing algorithms.

1 Introduction

"The problem of recovery from syntax errors in compilers is not yet satisfactorily solved" states Richter[13] and the shear amount of literature on syntactic error handling (see [7]) attests to its difficulty. Most practical methods lack a sound theoretical foundation and even worse, some theoretical investigations[20] conclude that automatic tools will never be able to generate good error correctors.

Today, the best error-correction methods for LR(k) parsers, as described in [8], are characterized by a two-phase approach: After the error is detected, the first phase, called the *condensation phase*, attempts to parse the unread portion of the input, so gathering right context (*forward move*). Some methods also collect the left context of the error (*backward move*). The second phase, called the *correction phase*, uses the previously gathered information to modify the token sequence and/or stack configuration to restart the analysis and generates diagnostic messages.

To gain a foundation for this method, two questions must be answered:

- Is it possible to obtain a reliable analysis of the left or right context of the error as a basis for the correction phase?
- Is it possible to find the correction, based on the results of the condensation phase, for some reasonably defined class of errors?

For LR(k) grammars, the analysis of the right context might depend on the entire left context and in the presence of errors the complete left context is usually not available. The left context, as captured by the parse stack, might not be reliable; especially SLR or LALR parsers can produce unwanted reductions before shifting the error-token.

A pathological example can illustrate how hopeless the situation is: Given a string α and two arbitrary parse trees $\hat{\alpha}_1$ and $\hat{\alpha}_2$ for this string, it is possible to write two LR(k) grammars G_1 and G_2 with start symbols S_1 and S_2 that produce the respective parse trees $\hat{\alpha}_1$ and $\hat{\alpha}_2$ from α. Adding two new terminal symbols `a` and `b`, and the productions $S \to \mathtt{a}S_1\mathtt{aa}$ and $S \to \mathtt{b}S_2\mathtt{bb}$, we can easily combine both grammars into a new LR(k) grammar G. Now a single token error can change the string $\mathtt{a}\alpha\mathtt{aa}$ into $\mathtt{b}\alpha\mathtt{aa}$. A typical LR($k$) parser will detect this error when reading the double `a` at the very end of the string. With $\mathtt{b}S_2$ as stack configuration, the left context does not provide any clue for the correct repair, the parse tree $\hat{\alpha}_2$ is completely wrong, and it takes an arbitrary large amount of work to change $\hat{\alpha}_2$ into $\hat{\alpha}_1$ before parsing can continue.

Unfortunately, similar examples can be found in common programming languages. Just consider the following program-fragments written in C[4]:

Example 1 *Fragments of C programs*

```
if( a, *b, **c=0, d[3]); else
fi( a, *b, **c=0, d[3]);
int a, *b, **c=0, d[3] ;
```

Here, the comma separated list is either the controlling expression of an if-statement, using the comma operator, or the argument list for the function `fi`, using the comma as a separator, or a list of variable declarations. A small change in the input will send the parser into completely different directions—whether this direction is correct or not—producing quite different parse trees.

Further examples are used in [20] to conclude that good automatic error correction is impossible. In contrast, this paper has a more optimistic view. It defines a class of grammars that are reasonably powerful and provably "well behaved" in the presence of errors.

Of the two questions given above, this paper deals almost exclusively with the second. We assume a parser that extracts as much structure as possible from a given string of input tokens (condensation phase) and ask how difficult

POPL 99 San Antonio Texas USA

it is to repair the resulting parse trees. It turns out that the class of continuous grammars not only offers a solution to this problem, but is a proper subclass of the Bounded Context Parseable grammars[21], a class of grammars that provides an answer to the first question.

The core of this paper are the metrics defined in section 2. In contrast to previous work[20] the metrics given here are derived directly from the parsing process and use the rich representation of parse trees. Based on these metrics, section 3 explains the notion of continuous grammars. Grammars for programming languages are designed to yield a mapping from strings of input tokens to parse trees and a parser is a program to compute this mapping. Intuitively, this mapping is called *continuous*, if small changes in the token string will cause only small changes in the resulting parse tree. Specifically, a continuous grammar ensures that the effect of any single token error is confined to a bounded set of nodes of the parse tree. Section 4 investigates the relationship of continuous grammars to other classes of grammars. The final section 5 discusses the results and provides links to related work.

2 Notation and Definitions

To avoid a lengthy section of standard definitions, we follow, as closely as possible, the notation system and definitions of [2] chapter 4, pp. 166, and provide here only a short informal summary. Wherever our notations and definitions are new or different, full details are given.

2.1 Symbols, Grammars, and Derivations

a, b, c, x are used for terminal symbols; S, A, B, C, are used for nonterminal symbols; X, Y, Z for symbols in general; and α, β, γ, ρ, σ, τ for strings of symbols.

The length n of a string $\alpha = X_1 \cdots X_n$ is denoted as $|\alpha|$.

Grammars are described with rules of the form $A \to \alpha$. The maximum length of the right hand sides over all rules $m = \max\{|\alpha| : A \to \alpha\}$ is called the branching factor of the grammar.

Grammars define languages through the process of derivation. Given the string $\sigma A \tau$ and the rule $A \to \alpha$, we can derive the string $\sigma\alpha\tau$ by replacing the occurrence of the left hand side of the rule by its right hand side. We write $\sigma A \tau \Rightarrow \sigma\alpha\tau$. Derivation is the reflexive and transitive closure of $\Rightarrow$; it is written $\stackrel{*}{\Rightarrow}$.

Strings that can be derived from a special start symbol are called sentential forms. Sentences are sentential forms containing only terminal symbols, and the set of all sentences defined by the grammar is the language.

2.2 Parse Trees

Parse trees capture the same information as derivations but filter out the choice regarding the replacement order. We use $[\beta \stackrel{*}{\Rightarrow} \gamma]$ to denote the parse tree that is defined by the derivation $\beta \stackrel{*}{\Rightarrow} \gamma$. When the derivation is clear from the context or not essential for the current argument, we will write $\hat{\beta}$ for the parse tree $[\beta \stackrel{*}{\Rightarrow} \gamma]$.

A parse tree is an ordered list of nodes, each labeled with a symbol, together with a parent/child relation. For a given derivation, the corresponding parse tree is defined by induction over the length of the derivation. For the basis, a string $\beta = Y_1 Y_2 \cdots Y_n$ with the trivial derivation $\beta \stackrel{*}{\Rightarrow} \beta$ corresponds to a list of nodes labeled $Y_1, Y_2, \ldots,$ and Y_n. For the induction step, if $\beta \stackrel{*}{\Rightarrow} \sigma A \tau \Rightarrow \sigma\alpha\tau$ with $\alpha = X_1 X_2 \cdots X_m$, we add a list of new nodes labeled $X_1, X_2, \ldots,$ and X_m that correspond to the symbols of α, and these nodes are child nodes of the node that corresponds to A. The symbols of σ and τ on the right side of $\Rightarrow$ correspond to the same nodes as the same symbols on the left side.

Note that a parse tree according to our definition is a tree in the traditional sense, with a single root node, only if $|\beta| = 1$; otherwise a parse tree, here, is a list of several tree structures, one for each symbol of β.

Strings are actually special cases of parse trees. Using the trivial derivation $\beta \stackrel{*}{\Rightarrow} \beta$ strings are embedded in the larger domain of parse trees. This natural embedding simplifies the description of parsing as a transformation on parse trees. No special consideration needs to be given to "input strings". For example, all metrics defined in the next section are metrics on strings as well.

Parse trees, however, can contain more information than plain strings—namely the grammatical structure—and this information is essential for the definition of metrics appropriately reflecting the parsing process.

2.3 Metrics

In short, the difference between two parse trees is described by sets of nodes, and we will measure the size of these sets by something like the "diameter" of the set, limiting the distance of two elements of the set. We start by defining the distance $d_{\hat{\alpha}}$ of two nodes X and Y in a parse tree $\hat{\alpha}$:

If X and Y are two root nodes and $\alpha = \cdots X \sigma Y \cdots$, the distance is naturally defined by $d_{\hat{\alpha}}(X,Y) = d_{\hat{\alpha}}(Y,X) = |\sigma| + 1$.

This definition is extended to internal nodes X and Y of $[\alpha \stackrel{*}{\Rightarrow} \gamma]$ by defining $d_{\hat{\alpha}}(X,Y) = 0$, if X is an ancestor of Y or Y is an ancestor of X; otherwise, $d_{\hat{\alpha}}(X,Y) = \min\{d_{\hat{\beta}}(X,Y) | \alpha \stackrel{*}{\Rightarrow} \beta\}$.

In summary, the distance of two nodes X and Y is measured by the distance of the corresponding symbols X and Y in a derivation. Since, however, a parse tree can have many different derivations and many symbols in a derivation correspond to the same node, we take the shortest possible distance, or the minimum distance.

The ultimate reason to define the measure $d_{\hat{\alpha}}$ as given above is however the parsing process. $d_{\hat{\alpha}}(X,Y)$ is the shortest distance that the nodes X and Y will have as entries on the parse stack for all possible parsers. A bound on $d_{\hat{\alpha}}(X,Y)$ is therefore of practical importance: for an optimal parser, it limits the amount of nodes on the parse stack that need to be considered.

Note, that $d_{\hat{\alpha}}$ is not a topological metric; the inequality $d_{\hat{\alpha}}(Y,Z) \leq d_{\hat{\alpha}}(X,Z) + d_{\hat{\alpha}}(X,Y)$ does not hold. The reason is simply that the node X might be an ancestor of both Y and Z. The following lemma gives a restricted form of this inequality that is needed for the proof of Theorem 1.

Lemma 1 *Let $\hat{\alpha}$ be a parse tree, let X, Y, and Z be nodes in $\hat{\alpha}$, and let Y be the parent node of X. Then $d_{\hat{\alpha}}(Y,Z) \leq d_{\hat{\alpha}}(X,Z)$*

PROOF. If $d_{\hat{\alpha}}(Y,Z) = 0$, then there is nothing to prove, otherwise Y is not an ancestor of Z. If $d_{\hat{\alpha}}(X,Z) = 0$, then X and Z have an ancestor relation, and hence Y and Z are in an ancestor relation, implying $d_{\hat{\alpha}}(Y,Z) = 0$.

Otherwise, X is not an ancestor of Z and $d_{\hat{\alpha}}(X,Z) = \min\{d_{\hat{\beta}}(X,Z) | \alpha \stackrel{*}{\Rightarrow} \beta\}$, since X, having a parent node Y, can not be a root node. Assume that the minimum defining $d_{\hat{\alpha}}(X,Z)$ is reached in β.

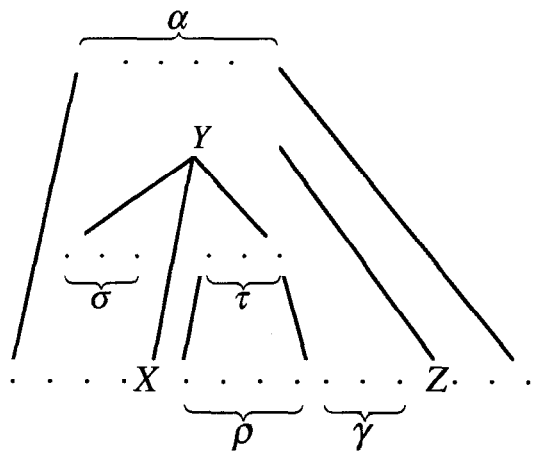

Figure 1: Parse Tree supporting the Proof of Lemma 1

The replacement $Y \to \sigma X \tau$ must be one step in the derivation $\alpha \overset{*}{\Rightarrow} \beta$. Without loss of generality, we assume that Z is to the right of X, then β has the form $\beta = \cdots X \rho \gamma Z \cdots$ with $\tau \overset{*}{\Rightarrow} \rho$ (see Figure 1). This implies that there is a derivation $\alpha \overset{*}{\Rightarrow} \cdots Y \gamma Z \cdots$ and therefore we have $d_{\hat{\alpha}}(Y,Z) \leq |\gamma| + 1 \leq |\rho\gamma| + 1 = d_{\hat{\alpha}}(X,Z)$. □

Given a parse tree $\hat{\alpha}$ and a set $\mathbf{S}$ of nodes of the parse tree, the size of $\mathbf{S}$, written as $|\mathbf{S}|$, is defined as the smallest number $n \in \mathbb{N}^0$ such that $d_{\hat{\alpha}}(X,Y) < n$ for all $X \in \mathbf{S}$ and $Y \in \mathbf{S}$. The size $|\mathbf{S}|$ of $\mathbf{S}$ measures not simply the number of nodes in $\mathbf{S}$ but is something like the diameter of $\mathbf{S}$, reflecting the fact that local changes are easier to accomplish than global changes.

As a measure of the distance of two parse trees $\hat{\alpha}$ and $\hat{\beta}$, we use the minimum amount of change needed to transform one parse tree into the other. To accomplish the transformation, first some nodes in $\hat{\alpha}$ are deleted, $\mathrm{Del}(\hat{\alpha},\hat{\beta})$ is used to denote this set of nodes. Then some nodes are inserted to obtain $\hat{\beta}$, this set of nodes is written as $\mathrm{Ins}(\hat{\alpha},\hat{\beta})$.

Now, we define the distance of two parse trees $\hat{\alpha}$ and $\hat{\beta}$, written $|\hat{\alpha}-\hat{\beta}|$, as the smallest number n such that there are sets $\mathrm{Del}(\hat{\alpha},\hat{\beta})$ and $\mathrm{Ins}(\hat{\alpha},\hat{\beta})$ with $|\mathrm{Del}(\hat{\alpha},\hat{\beta})|+|\mathrm{Ins}(\hat{\alpha},\hat{\beta})| = n$.

The distance $|\cdot|$ is not yet a topological metric on parse trees, because the "triangle-inequality" $|\hat{\alpha}-\hat{\beta}| \leq |\hat{\alpha}-\hat{\gamma}| + |\hat{\gamma}-\hat{\beta}|$ does not hold. The problem with this inequality is that it would allow to decompose the change from $\hat{\alpha}$ to $\hat{\beta}$ into a sequence of smaller changes. The smaller changes are reflected by smaller distances which, never the less, will yield an upper bound on the cumulative change. Of course any transformation of parse trees can be accomplished by changing one node at a time, and this would reduce the measure of distance to merely counting the number of different nodes. This model, however, would not be very realistic.

In practice, an error correction that affects two nodes far apart in the parse tree is much more difficult to find than a correction that affects two nodes close together. Further, error correctors usually try to correct the input one error at a time. This is reasonable, since syntax errors in real programs tend to be sparse[14], and efficient. In contrast, the search for a transformation that repairs all errors combined with the globally least cost is prohibitively expensive[1][11].

Therefore, our final measure of distance between parse trees adds up the diameters of node sets each repairing one token in the input string. To normalize this metric, we measure the size of the node set relative to the size of the input string. We use $||\cdot||$ to denote this "normalized single token error distance" and define it as follows: For two parse trees $[\alpha \overset{*}{\Rightarrow} \sigma]$ and $[\beta \overset{*}{\Rightarrow} \tau]$ with $|\sigma - \tau| = 1$ we define $||\hat{\alpha} - \hat{\beta}|| = |\hat{\alpha} - \hat{\beta}| / \max\{|\sigma|, |\tau|\}$. Of course $||\hat{\alpha} - \hat{\beta}|| = 0$ iff $\hat{\alpha} = \hat{\beta}$. If $\hat{\alpha} \neq \hat{\beta}$ there are sequences $[\alpha_0 \overset{*}{\Rightarrow} \sigma_0], \ldots, [\alpha_n \overset{*}{\Rightarrow} \sigma_n]$ with $\hat{\alpha}_0 = \hat{\alpha}$, $\hat{\alpha}_n = \hat{\beta}$, and $|\sigma_i - \sigma_{i+1}| = 1$ that decompose the change from $\hat{\alpha}$ to $\hat{\beta}$ into "single token errors". Considering all such sequences we define $||\hat{\alpha} - \hat{\beta}|| = \inf\{\sum_{i=0}^{n-1} ||\hat{\alpha}_i - \hat{\alpha}_{i+1}||\}$.

It is easy to verify that $||\cdot||$ indeed defines a metric and makes the set of parse trees a topological space. The continuous mappings on parse trees as defined in the next section are exactly the Lipschitz-continuous[1] (see e.g. [6]) functions regarding this topology. Because the measure $|\cdot|$ is more intuitive to use and closer to the parsing process, we will develop an equivalent definition of continuity using this measure and will continue to use it throughout this paper.

3 Continuity of a Grammar

3.1 Handle Pruning

Since strings are just special cases of our generalized parse trees, Grammars define mappings on parse trees. The computation of such a mapping is called parsing. It is the reverse process of derivation. Here, we specifically investigate bottom-up parsing by handle pruning.

A handle of a sentential form β is defined as a substring α of β and a rule $A \to \alpha$ such that the replacement of A by α is one step in the derivation of β. Note that our definition does not require α to be the leftmost substring with this property, a restriction that is often included in the definition of "handle" because LR(k) parsing, called "canonical parsing", is the main parsing technique under consideration. Our definition is less restrictive allowing non canonical parsing techniques[18, 19]. A derivation of a sentential form β can be reconstructed by successively identifying a handle of $\beta = \sigma\alpha\tau$ and replacing the substring α by A to yield $\sigma A \tau$ until no more handles are left. This process is called handle pruning.

Grammars for programming languages are usually designed to map sentences to uniquely[2] defined parse trees and therefore parsing by handle pruning defines a function mapping sentences to parse trees. In the following, the symbol f is used to denote such a function mapping β to a parse tree $f(\beta)$.

Since the nature of our investigation makes it necessary to look at arbitrary strings, the above definition needs to be generalized. Our generalization is motivated by the aim of providing a reliable analysis of the left and right context of a syntax error. Imagine a string ρ, which is a correct fragment of a program between two successive errors, and a substring α of ρ together with a rule $A \to \alpha$. The pruning of α will be valid, regardless of the possible continuation of ρ, if for all sentential forms β that have ρ as a substring the replacement of A by α is one step in the derivation of β.

In general, we define a handle of an arbitrary string γ as a substring α of γ and a production $A \to \alpha$ such that there

[1] It makes no sense to discuss plain continuous mappings relative to this topology since it is discrete (all mappings are continuous).

[2] All grammars here are unambiguous.

is a substring ρ of γ containing α such that for all sentential forms β that have ρ as a substring the replacement of A by α is one step in the derivation of β. The definition means that the substring α is in a local context ρ, which ensures the correctness of handle pruning independent of the global context γ, which might not be a sentential form at all.

Using the generalized definition of handle pruning, parse trees can be computed for arbitrary strings. With an unambiguous grammar, the parse tree $f(\gamma)$ is well defined for all sentences γ, and if γ is a program containing syntax errors, $f(\gamma)$ will be a conservative approximation to the correct parse tree.

There are usually still several possible ways to extend the function f to arbitrary strings. In the following, we assume that f denotes one such extension.

3.2 Continuity

The usual definition of continuity for a function f is:

$$\forall x\, \forall \epsilon > 0\, \exists \delta > 0\, \forall y\, |x - y| < \delta \rightarrow |f(x) - f(y)| < \epsilon$$

Intuitively this reads: if we can make the difference between x and y (the error in y) small enough (less than δ) the difference after application of f will also be small (less than any given ϵ). And applied to error correction it means: if f is continuous, an error corrector capable of correcting parse trees up to an error distance ϵ will be sufficient to correct all input errors smaller than δ.

This definition cannot be used directly because strings are finite objects and the natural topology is discrete. Unlike the case of real analysis, there exists a smallest distance between two different strings α and β. One could always choose $\delta < 1$ and satisfy the condition trivially. Therefore we replace the requirement $|x - y| < \delta$ by $|\alpha - \beta| = 1$, allowing a single token error (the smallest possible distance). The restriction to single symbol changes in α is not a serious limitation, since any large change can be decomposed into a sequence of single symbol changes.

If we cannot make the bound δ arbitrarily small, we cannot expect that it is possible to satisfy the condition for an arbitrarily small ϵ. But even though 1 is the smallest possible distance between two different strings, we perceive the change of one symbol in a 10,000 symbol sentence as a fairly small change, compared to the change of one symbol in a three symbol sentence. Therefore it is reasonable to require that $|f(\alpha) - f(\beta)|$ is bounded by a constant. This implies that the difference between $f(\alpha)$ and $f(\beta)$ is "relatively small" for sufficiently large α, for which, in turn, a one symbol change from α to β is also "relatively small".

In the above definition ϵ and δ may depend on x and one could consider to allow the bound on $|f(\alpha) - f(\beta)|$ to depend on α as well. The fact, however, that α and $f(\alpha)$ are finite objects gives a trivial upper bound: $2|\alpha| + 1$. Therefore, only a uniform bound for all α makes sense—similar to the definition of uniform continuity in real analysis.

We finally obtain the following definition: f is called continuous if

$$\exists n \in \mathbb{N}\, \forall \alpha\, \forall \beta\, |\alpha - \beta| = 1 \rightarrow |f(\alpha) - f(\beta)| < n$$

It is easy to prove that this definition is equivalent to

$$\exists n \in \mathbb{N}\, \forall \alpha\, \forall \beta\, ||f(\alpha) - f(\beta)|| < n||\alpha - \beta||$$

This is the usual definition of Lipschitz-continuous (see e.g. [6]) relative to the metric $||\cdot||$.

A grammar is called continuous, or a C grammar, if there exists an (extension) f that is continuous. The following example presents a grammar that is continuous.

Example 2
$S \rightarrow \mathtt{a}BA \quad S \rightarrow \mathtt{c}BC \quad A \rightarrow \mathtt{x} \quad C \rightarrow \mathtt{x} \quad B \rightarrow \mathtt{b}$
$B \rightarrow \mathtt{b}B$
Language = $\{\mathtt{ab}^n\mathtt{x}, \mathtt{cb}^n\mathtt{x}\}$

This grammar illustrates how a kind of unbounded left context can be used during parsing without, at the same time, propagating errors through large parts of the resulting parse tree and destroying continuity. Example 1, on the other hand, illustrates a discontinuity: a small error can propagate and change structure and meaning of arbitrarily large parts of a program.

Lemma 2 *The grammar of example 2 is continuous.*

PROOF. Assume an arbitrary string α. Any maximal substring that contains only b's can be reduced to a single B. Then, given a substring of the form $\mathtt{a}B\mathtt{x}$ the x can be reduced to A. The same idea holds for $\mathtt{c}B\mathtt{x}$. Finally, substrings of the form $\mathtt{a}BA$ and $\mathtt{c}BC$ can be reduced to S. In case α was a sentence of the language, this results in a parse tree with a single root.

Now consider obtaining β by changing a single token of α. Here only two representative cases are presented; all others are mere variations and do not add further insight. For each case, we show that $|f(\alpha) - f(\beta)|$ is bounded by a constant.

- Deleting/Inserting a b into a sequence of b's is equivalent to deleting/inserting the first b of a whole sequence of b's. The change to $f(\alpha)$ is a missing/extra B, the deletion of the old S, and the insertion of a new S. Therefore $|f(\alpha) - f(\beta)| = 2$.
- The maximum "damage" to a parse tree is achieved by inserting an a into a sequence of b's preceded by an c and followed by a x. As can be seen from Figure 2, $|\mathrm{Del}(f(\alpha), f(\beta))| = 2$ and $|\mathrm{Ins}(f(\alpha), f(\beta))| = d_{f(\alpha)}(B, A) + 1 = 4$. Therefore $|f(\alpha) - f(\beta)| \leq 6$. □

4 BC and BCP Grammars

4.1 BC Grammars

A grammar is called a BC(j,k) grammar—parseable with Bounded Context—, if, given any sentential form, it is possible to identify all handles of the sentential form looking at j symbols to the left and k symbols to the right of the handle (for some finite j and k).

BC(j,k) grammars never gained much importance, since shortly after Floyd[9] presented them in 1964, Knuth[12], in 1965, introduced LR(k) grammars and showed how very efficient parsers can be constructed for this larger class of grammars. All BC grammars, however, are continuous and so form an important subclass of continuous grammars. This is proved next.

Theorem 1 *BC$(j,k) \subset C$*

PROOF. Assume we are given a BC(j,k) grammar and two strings α and β with $|\alpha - \beta| = 1$.

Two cases have to be considered: β is obtained from α by inserting one symbol or by deleting one symbol. Both cases are completely symmetric, so it is sufficient to look at only one case.

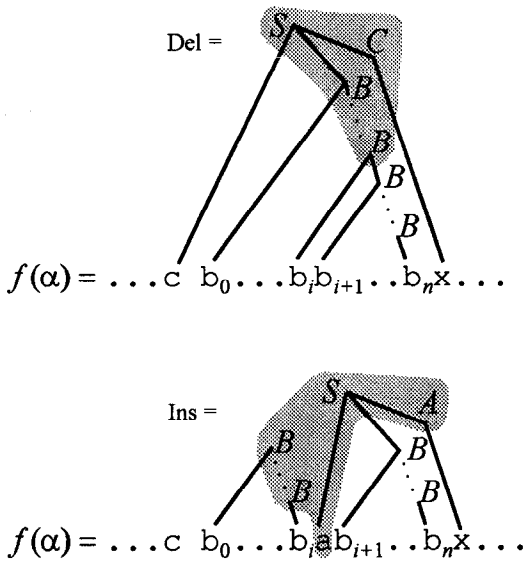

Figure 2: Parse Trees supporting proof of Lemma 2, Inserting an a

Assume $\alpha = X_1 \cdots X_n$, and $\beta = X_1 \cdots X_{i-1} X_{i+1} \cdots X_n$ is obtained by deleting a single symbol X_i from α.

To compute $f(\alpha)$ or $f(\beta)$, all substrings of $X_1 \cdots X_{i-k-1}$ and of $X_{i+j+1} \cdots X_n$ which can be identified as handles can be pruned first. Since the maximum context needed is only k symbols to the right and j to the left, none of the handle pruning is affected by the existence or deletion of X_i which can neither be part of these handles nor part of the necessary contexts. The handle pruning can continue this way as long as there are identifiable handles and none of the symbols $X_{i-k} \cdots X_{i+j}$ is part of these handles. So far the nodes generated in the computation of $f(\alpha)$ are exactly the same as those generated in the computation of $f(\beta)$.

From here on, the computation of $f(\alpha)$ and $f(\beta)$ might go in different directions, adding further nodes. One can choose $\mathrm{Del}(f(\alpha), f(\beta))$ to be the set of all nodes further added in the computation of $f(\alpha)$ and $\mathrm{Ins}(f(\alpha), f(\beta))$ to be the set of all nodes further added in the computation of $f(\beta)$. It is clear that each of these additional nodes has at least one of the symbols $X_{i-k}, \ldots, X_{i+j}$ as descendent node.

Using Lemma 1, one can conclude that
$|\mathrm{Ins}(f(\alpha), f(\beta))| \leq d_{f(\beta)}(X_{i-k}, X_{i+j}) \leq k + j - 1$ and
$|\mathrm{Del}(f(\alpha), f(\beta))| \leq d_{f(\alpha)}(X_{i-k}, X_{i+j}) \leq k + j$.
It follows immediately that $|f(\alpha) - f(\beta)| < 2k + 2j$, proving the continuity of f. □

Theorem 2 *$C \not\subseteq BC(j,k)$*

PROOF. Example 2 presents a continuous grammar that is not BC(j,k). □

4.2 BCP(j,k) Grammars

We call a grammar BCP(j,k)—Bounded Context Parseable—, if, given any sentential form, it is possible to identify **at least one** handle of the sentential form looking at j symbols to the left and k symbols to the right of the handle (for some finite j and k).

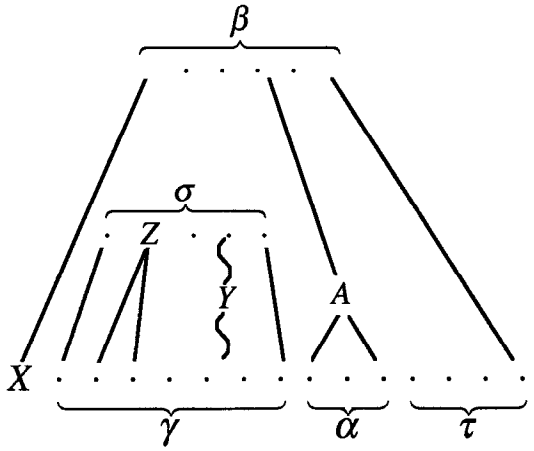

Figure 3: Parse Tree supporting the Proof of Lemma 3

BCP(j, k) Grammars were introduced by Williams[21] and studied extensively in the 1970's[17, 18, 19]. An interesting subclass of BCP grammars is the class of Bounded Right Context grammars[9]—basically the intersection of LR(k) and BCP(j, k). This class of grammars is large enough to generate all deterministic languages but does still allow for fast linear parsing algorithms[10][16]. The class of LR(k) grammars and the class of continuous grammars are incommensurate, there is, however, a general way to transform any LR(k) grammar into a BCP grammar[10]. While this technique yields BCP grammars automatically, it does not make the grammars continuous. How to write continuous grammars for popular programming languages is illustrated in [16].

We will prove that the BCP grammars form a true superset of the continuous grammars, thus locating C grammars just between BC and BCP grammars.

First, we prove a lemma estimating the "damage" caused by a missing symbol X that is crucial for the decision to reduce α to A later in the parser input. The damage will grow at least logarithmically with the distance between X and α.

Lemma 3 *If $\beta \stackrel{*}{\Rightarrow} X\gamma A\tau \Rightarrow X\gamma\alpha\tau$ then*

$$|\hat{\beta} - \gamma\alpha\tau| > \log_m(|\gamma|)/2\,,$$

where m is the branching factor of the grammar.

PROOF. By definition of $d_{\hat{\beta}}(X, A)$, there is a string σ with $\beta \stackrel{*}{\Rightarrow} X\sigma A\tau$ and $\sigma \stackrel{*}{\Rightarrow} \gamma$ such that $d_{\hat{\beta}}(X, A) = |\sigma| + 1$ (see Figure 3).

We prove by induction on $|\sigma|$ that there is a node Y in $\hat{\sigma}$ with $Y \notin \gamma$ such that $d_{\widehat{X\sigma A}}(X, Y) + d_{\widehat{X\sigma A}}(Y, A) > \log_m(|\gamma|)$

The lemma follows directly from this because X, Y, and A are elements of $\mathrm{Del}(\hat{\beta}, \gamma\alpha\tau)$, and $d_{\hat{\beta}}$ is the same as $d_{\widehat{X\sigma A}}$ for the distances of X, Y, and A.

- Assume $|\sigma| = 1$.

  Consider a path from the root node of $\hat{\sigma}$ down to an element of γ and a node Y on this path. Let n be the length of the path from the root down to Y. Note,

that here and in the following we do not count unit productions, that is nodes that have only one child, when computing the length of a path.

We prove, by induction on n that

$$d_{\widehat{X\sigma A}}(X,Y) + d_{\widehat{X\sigma A}}(Y,A) \geq n+2.$$

Basis $n = 0$:
We have $\sigma = Y$ and $d_{\widehat{X\sigma A}}(X,Y) + d_{\widehat{X\sigma A}}(Y,A) = 2$.

Induction step:
Assume Y has $k \geq 2$ child nodes $Y_1, \ldots, Y_k$.

We then have for all i with $1 \leq i \leq k$
$d_{\widehat{X\sigma A}}(X,Y_i) = d_{\widehat{X\sigma A}}(X,Y) + i - 1$ and
$d_{\widehat{X\sigma A}}(Y_i,A) = d_{\widehat{X\sigma A}}(Y,A) + k - i$. Therefore
$d_{\widehat{X\sigma A}}(X,Y_i) + d_{\widehat{X\sigma A}}(Y_i,A) =$
$d_{\widehat{X\sigma A}}(X,Y) + d_{\widehat{X\sigma A}}(Y,A) + k - 1 \geq n+2+1$.

The parse tree $\hat{\sigma}$ has $|\gamma|$ leaves and a maximum branching factor of m. Its height will be at least $\log_m(|\gamma|)$. Let Y be the second to the last node in a path with maximal length. Then $Y \notin \gamma$ and $d_{\widehat{X\sigma A}}(X,Y) + d_{\widehat{X\sigma A}}(Y,A) \geq \log_m(|\gamma|) - 1 + 2 > \log_m(|\gamma|)$.

- Assume $|\sigma| > 1$.

  Let Z be a symbol in σ that has a parse tree with a minimum number of leaves. Now delete $\hat{Z}$ from $\hat{\sigma}$ to obtain $\hat{\rho}$. If we delete from γ all the descendents of Z, the length of γ is reduced but still has at least half the original length. We can therefore find a node Y in $\hat{\rho}$ with $d_{\widehat{X\rho A}}(X,Y) + d_{\widehat{X\rho A}}(Y,A) > \log_m(|\gamma|/2) \geq \log_m(|\gamma|) - 1$. Reinserting $\hat{Z}$ into $\hat{\rho}$ will increase either the distance of X to Y or the distance of Y to A by 1, giving the desired result. □

Theorem 3 *$C \subset BCP(j,k)$*

PROOF. Given any grammar with branching factor m, we now construct sequences of sets, and use a superscript $k \in \mathbb{N}$ to identify the elements of a sequence.

Let $\mathbf{F}^k$ be the finite set of all strings σ such that
$|\sigma| < 2k+m$ and σ is a sentential form or an initial or final segment of a sentential form, or
$|\sigma| = 2k+m$ and σ is a substring of some sentential form.

Now construct a table $\mathbf{T}^k$, that contains for each string $\sigma \in \mathbf{F}^k$ and each handle of σ that can be identified, an entry giving the string and the handle. If the grammar is BCP(k,k), (or BCP(j,i) for some $j \leq k$ and $i \leq k$), then the table can be used to parse arbitrary sentences: Given a sentential form ρ there is a substring α ($|\alpha| \leq m$) of ρ that can be identified as a handle using at most k symbols left and right. Therefore, there is a substring of ρ containing α, which extends α to the left and to the right by at most k symbols and is sufficient to identify α as a handle. This substring can be extended further until either the end of ρ is reached or the length is $2k+m$. This final substring of ρ, together with the handle, will be in the table $\mathbf{T}^k$ constructed above. The table can be used to look up the handle and reduce it.

Assume now that the grammar is not BCP, then for each k a counter example to the above construction can be found. That is, for every k there is a sentential form ρ such that no substring of ρ has an entry in $\mathbf{T}^k$.

Let β be a minimum length substring of ρ that has at least one identifiable handle α with rule $A \to \alpha$. β exists because ρ, being a complete sentential form, allows the identification of all handles. Also, $|\beta| > 2k+m$ because smaller substrings do not allow the identification of handles, due to the choice of ρ.

Without loss of generality we can assume that $\beta = X\gamma\alpha\tau$ with $|X\gamma| \geq |\tau|$ (the handle is right of the middle).

We have $|\beta - \gamma\alpha\tau| = |X\gamma\alpha\tau - \gamma\alpha\tau| = 1$ and proceed by estimating $|f(\beta) - f(\gamma\alpha\tau)|$.

$\gamma\alpha\tau$ has no identifiable handle since β was minimal and hence $f(\gamma\alpha\tau) = \gamma\alpha\tau$.

Since $f(\beta)$ contains $X\gamma A\tau \Rightarrow X\gamma\alpha\tau$, we can use Lemma 3 to obtain

$$|f(\beta) - f(\gamma\alpha\tau)| = |f(\beta) - \gamma\alpha\tau| > \log_m(|\gamma|)/2$$

Using $|X\gamma\alpha\tau| > 2k+m$, $|\alpha| \leq m$, and $|X\gamma| \geq |\tau|$, we have $|\gamma| \geq k$ and finally $|f(\beta) - f(\gamma\alpha\tau)| > \log_m(k)/2$.

Since this is true for all k, a finite bound cannot exist, and consequently the grammar is not continuous. □

Theorem 4 *BCP(j,k) $\not\subset$ C*

PROOF. Example 3, below, presents a grammar that is BCP but not continuous.

Example 3
$S \to \mathtt{a}A \quad S \to \mathtt{c}C \quad A \to \mathtt{x} \quad C \to \mathtt{x}$
$A \to DA \quad C \to EC \quad D \to \mathtt{b} \quad E \to \mathtt{b}$
Language = $\{\mathtt{ab}^n\mathtt{x}, \mathtt{cb}^n\mathtt{x}\}$

The proper reduction of the trailing $\mathtt{x}$ as well as the reductions for the preceding $\mathtt{b}$'s, depends on the first symbol. Therefore the string $\alpha = \mathtt{b}^n\mathtt{x}$ has no identifiable handle, and the string $\beta = \mathtt{ab}^n\mathtt{x}$ can be reduced completely. We have $|\alpha - \beta| = 1$ and $|f(\alpha) - f(\beta)| = d_{f(\beta)}(\mathtt{a}, \text{parent of } \mathtt{x}) = n+1$. Thus, the grammar is not continuous. □

5 Summary

Parse trees are the product of the parsing process—tree structures found on the parse stack. We defined a metric on the nodes of these trees based on the minimum distance these nodes have as entries on the parse stack using standard bottom-up parsing techniques[3] and their generalizations[17, 16]. The size of a set of nodes was then defined by the maximum distance of its elements. As usual, the distance between two parsing situations, as captured by the complete parse stack, is measured considering the nodes to be deleted and then inserted to transform one situation into the other. The measure directly corresponds to the computational effort needed for an optimal "repair" of the parse state after an "error" is found.

There is only one assumption made here that is not trivially true for most parsers used today: the measure of distance assumes that all handles, even after the point of error, are reduced, if possible. This feature is available in some modern parsers[5] that can perform non-corrective syntax analysis[13].

Further, [16] presents a suitable parser generator for continuous LR(k) grammars which produces parsers as fast as traditional parsers generated by lex and yacc. It proves that the function f can be computed efficiently.

Continuous grammars can be written for popular programming languages as demonstrated in [15] where the parser generator described in [16] was used to implement a robust pretty printer for the programming language C using a continuous LR(k) grammar. Unfortunately there is, so far,

no known algorithm to decide the continuity of an arbitrary grammar. Practical examples and specific details on how to rewrite ordinary LR(k) grammars to make them continuous can be fond in [16].

We defined the class of continuous grammars, in a manner quite similar to the classic ϵ-δ-definition of real analysis. As a main result, we have BC $\subset$ C $\subset$ BCP. Further interesting results on the hierarchy of grammatical classes can be found in [17].

Continuity is defined independently of a particular parsing algorithm. While we know that a BCP parser is always sufficient for a continuous grammar, this does not preclude the use of ordinary LL or LR(k) parsers. Similar, continuity does not imply any particular mechanism of error recovery and repair. All it does is limiting the worst case. Given a discontinuous grammar, every error repair mechanism will fail on some cases. Too long this was taken for granted and even provably unavoidable. Given a continuous grammar, there will be no longer any excuse for a bad error recovery mechanism.

Usually, surprisingly little is known about the relation of parser input and parser output in the presence of input errors[20]. Continuity is a very important property in this respect, it provides precise bounds on the effects of all single token errors. To preserve continuity, parsing decisions for an unbounded segment of the input must not depend on a bounded segment of the input, as for instance in the first and last example.

The ideas and results presented here are of threefold use:

1. Since the class of continuous grammars is not defined in terms of a parsing algorithm, the main application of the ideas presented here are expected to be in the area of language design. Language designers might become better aware of the kind of features that make errors in programs hard to find and correct. To put it somewhat simply, it is a bad idea to have the same syntax for two different entities such that some context "far away" is necessary to find the proper interpretation (see Example 1).

2. Even if the language is already defined, the implementors have some choices left. The grammar used for parsing can still be changed to have a better basis for error diagnosis and recovery (for details see [16]). This paper points the implementor in the direction of BCP grammars and shows why BCP grammars are a good basis for parsers with improved error correction. It is interesting to note that research in HCI confirms that a good human computer interface should have no hidden state information.

3. Finally, a close analysis of the definitions and proofs given here reveal interesting bounds. In particular, the proof of Theorem 1 gives a bound which applies for BC grammars, or for those parts of a grammar that are BC, limiting the choice of nodes that must be examined and changed to find a minimum distance repair for single token errors.

The work presented here is part of the authors research in syntactic error correction. Commonly, parsing is seen as a front-end for code generation and as such a solved problem (lex, yacc). With code generation as the ultimate goal, the correctness of the input string becomes a natural assumption and syntactic errors are considered an exception. Every programmer, however, can attest to the fact that during program development syntactic errors are the rule rather than the exception. For this special situation, a dedicated parser with improved error diagnosis and interactive error correction is desirable. This is still an unsolved problem, but there is reason to hope that programming languages and grammars, designed with an eye on the problem of error correction, together with appropriate parsers, can perform error corrections of far better quality than available today.

References

[1] Alfred V. Aho and Thomas G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal on Computing*, 1(4):305–312, December 1972.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques and Tools.* Addison Wesley, Reading, MA, 1986.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques and Tools.* Addison Wesley, 1986.

[4] American National Standards Institute, New York, NY. *American National Standard for Information Systems —Programming Language—C, ANSI X3.159-1989*, 1990.

[5] Joseph Bates and Alon Lavie. Recognizing substrings of LR(k) languages in linear time. *ACM Transactions on Programming Languages and Systems*, 16(3):1051–1077, May 1994.

[6] Robert C. Buck. *Advanced Calculus.* McGraw-Hill, New York, NY, 1978. Definition of Lipschitz-continuous.

[7] Joachim Ciesinger. A bibliography of error-handling. *SIGPLAN Notices*, 14(1):16–26, January 1979.

[8] PierPaolo Degano and Corrado Priami. Comparison of syntactic error handling in LR parsers. *Software—Practice and Experience*, 25(6):657–679, June 1995.

[9] Robert W. Floyd. Bounded context syntactic analysis. *Communications of the ACM*, 7(2):62–67, 1964.

[10] Susan L. Graham. *Precedence Languages and Bounded Right Context Languages.* PhD thesis, Stanford University, 1971.

[11] Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462, July 1980.

[12] Donald E. Knuth. On the translation of languages from left to right. *Information and Controll*, 8:607–639, 1965.

[13] Helmut Richter. Noncorrecting syntax error recovery. *ACM Transactions on Programming Languages and Systems*, 7(3):478–489, July 1985.

[14] David G. Ripley and Frederick C. Druseikis. A statistical analysis of syntax errors. *Computer Languages*, 3(4):227–240, June 1978.

[15] Martin Ruckert. Conservative pretty printing. *SIGPLAN Notices*, 32(2):45–53, 1997.

[16] Martin Ruckert. Generating efficient substring parsers for BRC grammars. Technical Report 98-105, SUNY New Paltz, New Paltz, NY, July 1998.

[17] Thomas G. Szymanski. *Generalized Bottom-Up Parsing.* PhD thesis, Cornel University, Ithaca, NY, May 1973.

[18] Thomas G. Szymanski and John H. Williams. Noncanonical extensions of bottom-up parsing techniques. Technical Report 75-226, Cornel University, Ithaca, NY, January 1975.

[19] Kuo-Chung Tai. Noncanonical SLR(1) grammars. *ACM Transactions on Programming Languages and Systems*, 1(2):295–320, 1979.

[20] Charles Wetherell. Why automatic error correctors fail. *Computer Languages*, 2:179–186, 1977.

[21] John H. Williams. Bounded context parsable grammars. Technical Report 72-127, Cornel University, Ithaca, NY, April 1972.