# A Formal Model for Context-Free Languages Augmented with Reduplication

Walter J. Savitch

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093

**A model is presented to characterize the class of languages obtained by adding reduplication to context-free languages. The model is a pushdown automaton augmented with the ability to check reduplication by using the stack in a new way. The class of languages generated is shown to lie strictly between the context-free languages and the indexed languages. The model appears capable of accommodating the sort of reduplications that have been observed to occur in natural languages, but it excludes many of the unnatural constructions that other formal models have permitted.**

## 1 Introduction

Context-free grammars are a recurrent theme in many, perhaps most, models of natural language syntax. It is close to impossible to find a textbook or journal that deals with natural language syntax but that does not include parse trees someplace in its pages. The models used typically augment the context-free grammar with some additional computational power so that the class of languages described is invariably larger, and often much larger, than the class of context-free languages. Despite the tendency to use more powerful models, examples of natural language constructions that require more than a context-free grammar for weak generative adequacy are rare. (See Pullum and Gazdar 1982, and Gazdar and Pullum 1985, for surveys of how rare inherently noncontext-free constructions appear to be.) Moreover, most of the examples of inherently noncontext-free constructions in natural languages depend on a single phenomenon, namely the reduplication, or approximate reduplication, of some string. Reduplication is provably beyond the reach of context-free grammars. The goal of this paper is to present a model that can accommodate these reduplication constructions with a minimal extension to the context-free grammar model.

Reduplication in its cleanest, and most sterile, form is represented by the formal language $\{ww \mid w \in \Sigma^*\}$, where $\Sigma$ is some finite alphabet. It is well known that this language is provably not context-free. Yet there are numerous constructs in natural language that mimic this formal language. Indeed, most of the known, convincing arguments that some natural language cannot (or almost cannot) be weakly generated by a context-free grammar depend on a reduplication similar to the one exhibited by this formal language. Examples include the *respectively* construct in English (Bar-Hillel and Shamir 1964), noun-stem reduplication and incorporation in Mohawk (Postal 1964), noun reduplication in Bambara (Culy 1985), cross-serial dependency of verbs and objects in certain subordinate clause constructions in Dutch (Huybregts 1976; Bresnan et al. 1982) and in Swiss-German (Shieber 1985), and various reduplication constructs in English, including the *X or no X* construction as in: "reduplication or no reduplication, I want a parse tree" (Manaster-Ramer 1983, 1986). The model presented here can generate languages with any of these constructions and can do so in a natural way. To have some concrete examples at hand, we will review a representative sample of these constructions.

The easiest example to describe is the noun reduplication found in Bambara. As described by Culy (1985), it is an example of the simplest sort of reduplication in which a string literally occurs twice. From the noun *w* Bambara can form *w-o-w* with the meaning "whatever *w*." It is also possible to combine nouns productively in other ways to obtain new, longer nouns. Using these longer nouns in the *w-o-w* construction produces reduplicated strings of arbitrary length.

The *respectively* construction in English is one of the oldest well-known examples of reduplication (Bar-Hillel



0362-613X/89/010250-261$03.00

and Shamir 1964). It provides an example of reduplication other than exact identity. A sample sentence is:

John, Sally, Mary, and Frank are a
widower, widow, widow, and widower, respectively.

In these cases, it has been argued that the names must agree with "widow" or "widower" in gender, and hence the string from {widow, widower}* must be an approximate reduplication of the string of names. If one accepts the data, then this is an example of reduplication using a notion of equivalence other than exact identity. In this case the equivalence would be that the second string is a homomorphic image of the first one. However, one must reject the data in this case. One can convincingly argue that the names need not agree with the corresponding occurrence of "widow" or "widower," because gender is not syntactic in this case. It may be false, but it is not ungrammatical to say "John is a widow." (Perhaps it is not even false, since no syntactic rule prevents parents from naming a daughter "John.") However, such dependency is at least potentially *possible* in some language with truly syntactic gender markings. Kac et al. (1987) discuss a version of this construction using subject-verb number agreement that yields a convincing argument that English is not a context-free language.

One of the least controversial arguments claiming to prove that a particular natural language cannot be weakly generated by a context-free grammar is Shieber's (1985) argument about Swiss-German. In this case, the reduplication occurs in certain subordinate clauses such as the following:

... mer em Hans es huus hälfed aastriiche
... we Hans-DAT the house-ACC helped paint
'....we helped Hans paint the house.'

where to obtain a complete sentence, the above should be preceded by some string such as "Jan säit das" ('Jan says that'). In this case, a list of nouns precedes a list of an equal number of verbs and each noun in the list serves as the object of the corresponding verb. The cross-serial dependency that pushes the language beyond the reach of a context-free grammar is an agreement rule that says that each verb arbitrarily demands either accusative or dative case for its object. Thus if we substitute "de Hans" (Hans-ACC) for "em Hans" (Hans-DAT) or "em huus" (the house-DAT) for "es huus" (the house-ACC), then the above is ungrammatical because "hälfed" demands that its object be in the dative case and "aastriiche" requires the accusative case. Since the lists of nouns and verbs may be of unbounded length, this means that Swiss-German contains substrings of the forms

$$N_1 N_2 \cdots N_n V_1 V_2 \cdots V_n$$

where $n$ may be arbitrarily large and where each noun $N_i$ is in either the dative or accusative case depending on an arbitrary requirement of the verb $V_i$.

Bresnan et al. (1982) describe a similar construction in Dutch in which the strong agreement rule is not present and so the language (at least this aspect of it) can be weakly generated by a context-free grammar, even though it cannot be strongly generated by a context-free grammar. The context-free grammar to generate the strings would pair nouns and verbs in a mirror image manner, thereby ensuring that there are equal numbers of each. Since Dutch does not have the strong agreement rule that Swiss-German does, this always produces a grammatical clause, even though the pairing of nouns and verbs is contrary to intuition. However, in cases such as this, it would be desirable to have a model that recognizes reduplication as reduplication rather than one that must resort to some sort of trick to generate weakly the reduplicated strings in a highly unnatural manner. This is true even if one is seeking only weak generative capacity because, as the Dutch/Swiss-German pair indicates, if a minor and plausible addition to a construct in one natural language would make it demonstrably noncontext-free, then we can suspect that some other language may exhibit this or a similar inherently noncontext-free property, even when considered purely as a string set.

Some of these arguments are widely accepted. Others are often disputed. We will not pass judgment here except to note that, whether or not the details of the data are sharp enough to support a rigorous proof of noncontext-freeness, it is nonetheless clearly true that, in all these cases, something like reduplication is occurring. A model that could economically capture these constructions as well as any reasonable variant on these examples would go a long way toward the goal of precisely describing the class of string sets that correspond to actual and potential human languages.

We do not contend that the model to be presented here will weakly describe all natural languages without even the smallest exception. Any such claim for any model, short of ridiculously powerful models, is doomed to failure. Human beings taken in their entirety are general-purpose computers capable of performing any task that a Turing machine or other general-purpose computer model can perform, and so humans can potentially recognize any language describable by any algorithmic process whatsoever (although sometimes too slowly to be of any practical concern). The human language facility appears to be restricted to a much less powerful computational mechanism. However, since the additional power is there for purposes other than language processing, some of this power inevitably will find its way into language processing in some small measure. Indeed, we discuss one Dutch construction that our model cannot handle. We claim that our model captures most of the known constructions that make natural language provably not context-free as string sets, and that it does so with a small addition to the context-free grammar model. No more grandiose claims are made.

It is easy to add power to a model, and there are numerous models that can weakly generate languages representing all of these noncontext-free constructions.

However, they all appear to be much too powerful for the simple problems that extend natural language beyond the capacity of context-free grammar. One of the less powerful of the well-known models is indexed grammar, as introduced by Aho (1968) and more recently summarized in the context of natural language by Gazdar (1985). However, even the indexed languages appear to be much more powerful than is needed for natural language syntax. We present a model that is weaker than the indexed grammar model, simpler than the indexed grammar model, and yet capable of handling all context-free constructs plus reduplication.

A number of other models extend the context-free grammar model in a limited way. Four models that are known to be weakly equivalent and to be strictly weaker than indexed grammars are: the Tree Adjoining Grammars (TAGs) of Joshi (1985, 1987), the Head Grammars of Pollard (1984), the Linear Indexed Grammars of Gazdar (1985), and the Combinatory Categorial Grammars of Steedman (1987, 1988). For a discussion of this equivalence see Joshi et al. (1989). The oldest of these four models is the TAG grammar of Joshi, and we shall refer to the class of languages generated by any of these equivalent grammar formalisms as TAG languages. However, the reader should keep in mind that this class of languages could be represented by any of the four equivalent grammar formalisms. As we will see later in this paper, there are TAG languages that cannot be weakly generated by our model. Our model seems to exclude more unnatural strings sets than these models do. Of course, our model may also miss some natural string sets that are TAG languages. Recent work of Joshi (1989) appears to support our conjecture that the class of language described by our model is a subset, and hence a strict subset, of the TAG languages. However, all the details of the proof have not yet been worked out, and so any more detailed comparisons to TAG languages will be left for another paper.

This paper assumes some familarity with the notation and results of formal language theory. Any reader who has worked with context-free grammars, who knows what a pushdown automaton (PDA) is, and who knows what it means to say that PDAs accept exactly the context-free languages should have sufficient background to read this paper. Any needed background can be found in almost any text on formal language theory, such as Harrison (1978) or Hopcroft and Ullman (1979).

## 2 The RPDA Model

The model we propose here is an automata-based model similar to the pushdown automaton that characterizes the context-free languages. A formal definition will follow shortly, but the informal description given now should be understandable to anybody who has worked in this area. The model is called a **Reduplication PDA**, or more simply an **RPDA**. It consists of an ordinary PDA augmented with a special stack symbol, which is denoted $\downarrow$, and a special type of instruction to check for reduplication. The symbol $\downarrow$ is inserted in the stack just like any other stack symbol, and the stack grows above this symbol just as in an ordinary PDA. To check for an occurrence of a simple reduplication $ww$, the RPDA pushes $\downarrow$ onto the stack and then pushes the first $w$ onto the stack symbol by symbol. At that point the stack contains $\downarrow$ with $w$ on top of it, but while the stack symbols are ordered so that it would be easy to compare the $w$ in the stack with $w^R$ (i.e., $w$ reversed), they are in the wrong order to compare them with $w$. To overcome this problem an RPDA is allowed, in one step, to compare the entire string above the $\downarrow$ to an initial segment of the remaining input and to do so in the order starting at the special symbol $\downarrow$ rather than at the top of the stack. (The comparison consumes both the stack contents above the marker $\downarrow$ and the input with which it is compared.) One way to view this is to say that the RPDA can decide to treat the stack above the symbol $\downarrow$ like a queue, but once it decides to do so, all that it can do is empty the queue. The RPDA cannot add to the queue once it starts to empty it. If the RPDA decides to check $xy$ to see if $x = y$, and $x$ does not in fact equal $y$ (or any initial segment of $y$), then the computation blocks. While placing symbols on top of the symbol $\downarrow$, the stack may grow or shrink just like the stack on an ordinary PDA. The model allows more than one marker $\downarrow$ to be placed in the stack, and hence, it can check for reduplications nested within reduplications.

Because an RPDA is free to push something on the stack other than $w$ when processing some input $wx$, the model can check not only whether $w = x$, but also can check the more general question of whether some specific finite-state transduction maps $w$ onto $x$. A finite-state transduction is any relation that can be computed using only a finite-state machine. The finite-state transductions include all of the simple word-to-word equivalences used in known reduplication constructions. For example, for the Swiss-German subordinate clauses described by Shieber, $w$ is a string of nouns marked for either dative or accusative case, and $x$ is a string of verbs that each select one and only one of the cases for their corresponding, cross-serially located object noun. The finite-state transduction would map each noun in the accusative case onto a verb nondeterministically chosen from the finite set of verbs that take the accusative case, and would do a similar thing with nouns in the dative case and their corresponding class of verbs. Without this, or some similar generality, the only reduplication allowed would be exact symbols by symbol identity.

The formal details of the definitions are now routine, but to avoid any misunderstanding, we present them in some detail.

1. A **Reduplication PDA** (abbreviated **RPDA**) consists of the following items:

(i) A finite set of **states** $S$, an element $q_0$ in $S$ to serve

as the **start state**, and a finite subset $F$ of $S$ to serve as the **accepting states**;

(ii) A finite **input alphabet** $\Sigma$;

(iii) A finite **pushdown store alphabet** $\Gamma$such that $\Sigma \subseteq \Gamma$, a distinguished symbol $Z_0$ in $\Gamma$ to serve as the **start pushdown symbol**, and a distinguished **stack marker** $\downarrow$, which is an element of $\Gamma - \Sigma$;

(iv) A **transition function** $\delta$ that maps triples $(q, a, Z)$ consisting of a state $q$, an input symbol $a$, and a pushdown store symbol $Z$, onto a finite set of instructions, where each instruction is in one of the following two forms:

(1) An **ordinary PDA move**: $(p, push\ \alpha, \Delta)$, where $p$ is a state, $\alpha$ is a string of pushdown symbols, and $\Delta$ is one of the two instructions +1 and 0 standing for "advance the input head" and "do not advance the input head," respectively. (The word "push" serves no function in the mathematics, but it does help make the notation more readable.)

(2) A **check-copy move**: These instructions consist only of a state $p$. (As explained in the next two definitions, this is the state of the finite control after a successful move. A successful move matches that portion of the stack contents between the highest marker $\downarrow$ and the top of the stack against an initial segment of the remaining input and does so in the right order for checking reduplication.)

2. An **instantaneous description (ID)** of an RPDA $M$ is a triple $(q, w, \gamma)$, where $q$ is a state, $w$ is the portion of input left to be processed, and $\gamma$ is the string of symbols on the pushdown store. (The top symbol is at the left end of $\gamma$, which is the same convention as that normally used for ordinary PDA's.)

3. The **next ID relation** $\vdash$ is defined as follows:

$(p, aw, Z\alpha) \vdash (q, w, \beta\alpha)$, provided $(q, push\ \beta, +1) \in \delta(p, a, Z)$;

$(p, aw, Z\alpha) \vdash (q, aw, \beta\alpha)$, provided $(q, push\ \beta, 0) \in \delta(p, a, Z)$;

$(p, axw, Z\gamma\downarrow\alpha) \vdash (q, w, \alpha)$, provided $q \in \delta(p, a, Z)$ and $ax = (Z\gamma)^R$, where $(Z\gamma)^R$ denotes $Z\gamma$ written backwards (so the $a$ matches the symbol just above the stack marker $\downarrow$. Note that $Z\gamma$ cannot contain the symbol $\downarrow$, because $\downarrow$ is not in the input alphabet).

As usual, $\vdash^*$ denotes the reflexive-transitive closure of $\vdash$. Notice that the relation $\vdash$ is not a function. A given ID may have more than one next ID, and so these machines are nondeterministic.

4. An RPDA is said to be **deterministic** provided that $\delta(p, a, Z)$ contains at most one element for each triple $(p, a, Z)$.

5. The **language accepted** by the RPDA $M$ by final state is defined and denoted as follows: $L(M) = \{ w \mid (q_0, w, Z) \vdash^* (p, \Lambda, \gamma)$ for some $p \in F, \gamma \in \Gamma^*\}$, where $q_0$ is the start state and $F$ is the set of accepting states. ($\Lambda$ is used to denote the empty string.) If a language is accepted by some RPDA by final state it is called an **RPDA language**. If a language is accepted by some deterministic RPDA by final state, then it is called a **deterministic RPDA language**.

6. The **language accepted** by the RPDA $M$ **by empty store** is defined and denoted as follows: $N(M) = \{ w \mid (q_0, w, Z) \vdash^* (p, \Lambda, \Lambda)$ for some $p \in S\}$.

As in the case of ordinary PDAs, it turns out that for RPDAs acceptance by empty store is equivalent to acceptance by final state. The proof is essentially the same as it is for PDAs and so we will not repeat it here. The formal statement of the result is our first theorem.

**Theorem 1.** A language $L$ is accepted by some RPDA by final state if and only if it is accepted by some (typically different) RPDA by empty store.

As with ordinary PDAs, and for the same reasons, Theorem 1 does not hold for deterministic RPDAs.

## 3 Examples and Comparison to Other Classes

We next explain how RPDAs can be constructed to accept each of the following sample languages.

**Examples of RPDA languages.**

$L_0 = \{ww \mid w \in \{a, b\}^*\}$

$L_1 = \{wcw \mid w \in \{a, b\}^*\}$

$L_2 = \{wh(w) \mid w \in \{a, b\}^*\}$

where $h$ is the homomorphism $h(a) = c$ and $h(b) = dde$.

$L_3 = \{wxw \mid w \in \{a, b\}^*$ and $x \in \{c, d\}^*\}$

$L_4 = \{a_1 w_1 w_1 a_2 w_2 w_2 \cdots a_n w_n w_n a_n a_n a_{n-1} \cdots a_1 \mid a_i \in \{a, b\}, w_i \in \{c, d\}^*\}$

$L_5 = \{x_1 c x_2 c \cdots c x_n c a_n a_{n-1} \cdots a_1 \mid a_i \in \{a, b\}, x_i \in \{a, b\}^*, x_i$ is of the form $ww$ if and only if $a_i = b\}$

$L_6 = \{x_1 c x_2 c \cdots c x_n c a_1 a_2 \cdots a_n \mid a_i \in \{a, b\}, x_i \in \{a, b\}^*, x_i$ is of the form $ww$ if and only if $a_i = b\}$

$L_0$ is the simplest possible reduplication language. To accept this language, all that an RPDA need do is to insert the marker $\downarrow$ into the stack, copy symbols into the stack, guess when it has reached the midpoint, and then perform a check-copy move. If the center of the string is marked with a punctuation symbol, then the RPDA can be deterministic, so $L_1$ is a deterministic RPDA language.

The language $L_2$ illustrates the fact that reduplication need not be symbol-by-symbol identity. The RPDA to accept $L_2$ is similar to the one that accepts $L_0$, except that on reading an $a$ it pushes a $c$ into the stack instead of an $a$, and on reading a $b$ it pushes $edd$ on the stack instead of $b$.

$L_3$ illustrates the fact that reduplication may be checked despite the intervention of an arbitrarily long string. The construction of the RPDA is easy.

The language $L_4$ illustrates the facts that reduplication can be checked any number of times and that these checks may be embedded in a context free–like construction. To accept $L_4$ an RPDA would push $a_1$ and then $\downarrow$ onto the stack and proceed to check for a reduplication $w_1 w_1$ as described for $L_0$. If such a $w_1 w_1$ is found, that will leave just $a_1$ on the stack. The RPDA next pushes $\downarrow a_2$ on the stack (the $\downarrow$ is on top of the $a_2$)

and checks for $w_2\ w_2$. After processing an initial input string of the form

$$a_1\ w_1\ w_{\ 1}\ a_2\ w_2\ w_2 \cdots a_n\ w_n\ w_n$$

the stack will contain $a_n\ a_{n-1} \cdots a_1$ with $a_n$ on top of the stack. It can then easily check for the occurrence of a matching ending $a_n\ a_{n-1} \cdots a_1$. (It is also easy to check for an ending in the reduplicating order $a_1\ a_2 \cdots a_n$ using the techniques discussed below for $L_6$.)

$L_5$ and $L_6$ illustrate the fact that reduplication can be used as a distinguishable feature to carry some syntactic or semantic information. For example, the reduplicated strings might be nouns and the reduplication might be used to indicate a plural. The string of $a_i$s might be agreeing adjectives or verbs or whatever. $L_5$ using the mirror image construction is not meant to be typical of natural language but merely to illustrate that the reduplication might be embedded in some sort of phrase structure. $L_6$ shows that the RPDA model can obtain the same language with cross-serial dependency instead of mirror imaging.

To accept $L_6$ the RPDA needs to have two marker symbols $\downarrow$ in the stack at one time. To accept $L_6$ an RPDA would push the marker $\downarrow$ on the stack. This first marker will eventually produce the stack contents

(1) $a_n\ a_{n\ -1} \cdots a_1\ \downarrow$ (the top is on the left)

which it then compares to the ending string $a_1\ a_2 \cdots a_n$. To construct this string in the stack, it guesses the $a_i$ and uses a second marker to check its guesses. For example, if the RPDA guesses that $a_1 = b$ then it pushes $a_1 = b$ onto the stack and proceeds to check that $x_1$ is a reduplication string. To do this it pushes another marker $\downarrow$ onto the stack and checks $x_1$ in the way described for $L_0$ and other languages. If $x_1$ does not check out, then the computation aborts. If $x_1$ does check out, the stack contains $a_1\ \downarrow$ (the top is on the left) and the RPDA then guesses $a_2$. Say it guesses that $a_2 = a$ and hence must check that $a_2$ is *not* of the form $ww$. The RPDA then pushes $a_2$ onto the stack and performs the check. One straightforward way to perform the check is to push a marker $\downarrow$ on the stack and then read the first half of $x_2$ guessing at where it differs from the second half. The RPDA pushes its guess of the second half on the stack. If the RPDA correctly guesses the second half and if it ensures that it guesses at least one difference from the first half, then $x_2$ checks out. If $x_2$ checks out, then the stack will contain $a_2\ a_1\ \downarrow$ after all this. Proceeding in this way, the RPDA obtains the stack contents shown in (1) and then performs a final check-copy move to see if it matches the rest of the input string.

By examining these examples it is easy to see how an RPDA could deal with the reduplication constructions from natural language that were mentioned in the introduction. For example, in the Swiss-German subordinate clause construction, an RPDA would first push the stack marker $\downarrow$ onto the stack, then it would read the list of nouns, then for each noun it would nondeterministically choose a verb that requires the case of that noun. It would then push the chosen verb onto the stack, and when it reaches the list of verbs it would perform a check-copy move. Hence RPDAs seem capable of weakly generating languages that exhibit the properties that keep many natural languages from being context-free. As the next result indicates, their power is strictly between the context-free grammars and the indexed grammars. Most of the theorem is easy to prove, given known results. However, a proof that there is an indexed language that is not an RPDA language will have to wait until later in this paper when we will prove that the indexed language $\{a^n b^n c^n \mid n \geq 0\}$ is not an RPDA language.

**Theorem 2.** Context-Free Languages $\subset$ RPDA Languages $\subset$ Indexed Languages (and the inclusions are proper)

**Partial proof.** The first inclusion follows from the definitions. To see that the inclusion is proper recall that $\{ ww \mid w \in \{a, b\}^* \}$, which is well known to not be context-free, is an RPDA language. The second inclusion follows from the fact that an RPDA can be simulated by a one-way stack automaton, and all languages accepted by one-way stack automata are indexed languages, as shown in Aho (1969). The proof that there is an indexed language that is not an RPDA language will be proven in a later section of this paper.□

## 4 Variations on the RPDA Model

There are a number of variations on the RPDA model that one might try. One tempting variant is to replace the check-copy move with a stack-flipping move. This variant would allow the entire stack contents above the marker $\downarrow$ to be flipped so that, in one move, the pushdown-store contents $\alpha \downarrow \gamma$ would change to $\alpha^R \downarrow \gamma$. (The "top" is always the left end.) This would allow the machine to check for reduplication. However, it also allows the machine to check for everything else. This flipping stack variant is equivalent to a Turing machine because it can simulate a Turing machine tape by continually flipping the stack and using its finite control to "rotate" to any desired stack position. For example, $\alpha\beta \downarrow \gamma$ can be transformed into $\beta\alpha \downarrow \gamma$ by moving one symbol at a time from the top of the stack to just above the marker $\downarrow$. The top symbol is moved by remembering the symbol in the finite-control, flipping the stack, placing the remembered symbol on the stack, and flipping again.

One way to avoid the "Turing tar pit" in this flipping stack variant would be to deprive the machine of its stack marker $\downarrow$ after a flip. This would appear to limit the number of flips and so prevent the Turing machine simulation outlined. However, a nondeterministic machine could simply place a large supply of markers in the stack so that when one was taken away another would be at hand. To foil this trick, one might limit that machine to one stack marker, but this would restrict the machine so that it cannot naturally handle reduplications nested inside of reduplications. In Section 8,

RPDAs with only a single marker (and possessing one other restriction at the same time) are studied. That section concludes with a discussion of why some natural language constructs appear to require multiple markers.

When using a copy-check move, an RPDA can read an arbitrarily long piece of input in one move. This definition was made for mathematical convenience. A realistic model would have to read input one symbol per unit time. However, the formal model does not seriously misrepresent realistic run times. If the model were changed to read the input one symbols at a time while emptying the stack contents above the marker, then the run time would at most double.

## 5 Closure Properties

The next two theorems illustrate the fact that RPDA languages behave very much like context-free languages. The proofs are straightforward generalizations of well-known proofs of the same results for context-free languages. Using the PDA characterization of context-free languages and the proofs in that framework, one need do little more than replace the term "PDA" by "RPDA" to obtain the corresponding results for RPDA languages.

**Theorem 3.** The class of RPDA languages is closed under the following operations: intersection with a finite-state language, union, star closer, and finite-state transduction (including the special cases of homomorphism and inverse homomorphism).

**Theorem 4.** The class of deterministic RPDA languages is closed under the operations of intersection with a finite-state language and complement.

One detail of the proof of Theorem 3 does merit some mention. It may not, at first glance, seem obvious that the class of RPDA languages is closed under intersection with a regular set, since the proof that is usually used for ordinary PDAs does not carry over without change. In the ordinary PDA case, all that is needed is to have a finite-state machine compute in parallel with the PDA. In the case of an RPDA this is a bit more complicated, since the RPDA does not read its input symbol by symbol. In a check-copy move, an RPDA can read an arbitrarily long string in one move. However, the finite-state control can easily keep a table showing the state transitions produced by the entire string above the marker $\downarrow$. When a second $\downarrow$ is inserted into the stack, the old transition table is stored on the stack and a new transition table is started. The other details of the proof are standard.

Theorem 3 implies that the class of RPDA languages is a full AFL (Abstract Family of Languages), which in some circles invests the class with a certain respectability. This is because such closure properties determine much of the character of well-known language classes, such as context-free languages and finite-state languages. (A *Full AFL* is any class of languages that contains at least one nonempty language and that is closed under union, $\Lambda$-free concatenation of two languages, homomorphism, inverse homomorphism, and intersection with any finite-state language. See Salomaa 1973, for more details.)

The notion of a finite-state transduction is important when analyzing pushdown machines. If a finite-state control reads a string of input while pushing some string onto the stack (without any popping), then the string in the stack is a finite-state transduction of the input string. Unfortunately, the concept of a finite-state transduction is fading out of the popular textbooks. We will therefore give a brief informal definition of the concept.

**Definition.** A **finite-state transducer** is a nondeterministic finite-state machine with output. That is, it is a finite-state machine that reads its input in a single left-to-right pass. In one move it does all of the following: either read a symbol or move without consuming any input (called moving on the empty input) and then, on the basis of this symbol or the empty input, as well as the state of the finite-state machine, it changes state and outputs a string of symbols. (If, on first reading, you ignore moving on the empty string, the definition is very easy to understand. Moving on the empty string simply allows the transducer to produce output without reading any input.) There is a designated start state and a set of designated accepting states.

In a computation of a finite-state transducer there is an output string consisting of the concatenation of the individual strings output, but not all output strings are considered **valid**. The computation begins in the start state and is considered **valid** if and only if the computation ends in one of a designated set of accepting states. A string $y$ is said to be a **finite-state transduction** of the string $x$ **via the finite-state transducer** $T$ provided that there is a valid computation of $T$ with input $x$ and output $y$.

To motivate the following definition, consider changing a language so that plurals are marked by reduplication. For example, in English "papers" would change to "paperpaper." A sentence such as "Writers need pens to survive" would change to "Writerwriter need penpen to survive." This change of English can be obtained by first replacing all plural nouns with a special symbol ($a$ in the definition) and then performing a reduplication substitution as described in the definition. If the change were other than an exact copy, it would still satisfy the definition provided that a finite-state machine could compute the approximate copy. This operation allows us to take a language and introduce reduplication in place of any suitably easily recognized class of words and so obtain a language that uses reduplication to mark that class. The following theorem says that RPDA languages are closed under these sorts of operations.

**Definition.** Let $L$ be a language, $a$ a symbol, and $T$ a finite-state transduction. Define the language $L'$ to

consist of all strings $w$ that can be written in the form

$$x_0\, y_0\, x_1\, y_1 \cdots x_{n-1}\, y_{n-1}\, x_n \text{ where}$$

(i) each $x_i$ contains no $a$, s

(ii) $x_0\, ax_1\, a \cdots x_{n-1}\, ax_n \in L$, and

(iii) each $y_i$ is of the form $vv'$ where $v'$ is a finite-state transduction of $v$ via $T$.

A language $L'$, obtained in this way, is called a **reduplication substitution** of the language $L$ **via** $T$ **by substituting reduplication strings for** $a$. More simply, $L'$ is called a **reduplication substitution** of $L$ provided there is some symbol $a$ and some such finite-state transduction $T$ such that $L'$ can be obtained from $L$ in this way.

**Theorem 5.** If $L'$ is a reduplication substitution of a context-free language, then $L'$ is an RPDA language.

Among other things, Theorem 5 says that if you add reduplication to a context-free language in a very simple way, then you always obtain an RPDA language. In Section 8, we will prove a result that is stronger than Theorem 5 and so we will not present a proof here.

## 6 NonRPDA Languages

To prove that certain languages cannot be accepted by any RPDA, we will need a technical lemma. Despite the messy notation, it is very intuitive and fairly straightforward to prove. It says that if the stack marker $\downarrow$ is used to check arbitrarily long reduplications, then these reduplications can be pumped up (and down). For example, suppose an RPDA accepts a string of the form $usst$ and does so by pushing $\downarrow$, then $s$ onto the stack, and then matching the second $s$ by a check-copy move. It must then be true that, if $s$ is long enough, then $s$ can be written in the form $s_1\, s_2\, s_3$ and for all $i \geq 0$, the RPDA can do a similar pushing and checking of $s_1 s_2^i s_3$ to accept $us_1 s_2^{\,i}\, s_3\, s_1\, s_2^{\,i}\, s_3 t$.

**Pumping Lemma 1.** For every RPDA $M$, there is a constant $k$ such that the following holds. If *length(s)* $> k$ and

$$(p_1, rst, \downarrow \beta) \vdash^* (p_2, st, s^R \downarrow \beta) \vdash (p_3, t, \beta),$$

where the indicated string $\downarrow \beta$ is never disturbed, then $r$ and $s$ may be decomposed into $r = r_1\, r_2\, r_3$ and $s = s_1\, s_2\, s_3$, such that $s_2$ is nonempty and for all $i \geq 0$, $(p_1, r_1\, r_2^{\,i}\, r_3\, s_1\, s_2^{\,i}\, s_3\, t, \downarrow \beta) \vdash^* (p_2, s_1\, s_2^{\,i}\, s_3\, t, (s_1\, s_2^{\,i}\, s_3)^R \downarrow \beta) \vdash (p_3, t, \beta)$.

**Proof.** Without loss of generality, we will assume that $M$ pushes at most one symbol onto the stack during each move. Let $k$ be the product of the number of states in $M$ and the number of stack symbols of $M$. Consider the subcomputation

$$(p_1, rst, \downarrow \beta) \vdash^* (p_2, st, s^R \downarrow \beta)$$

Let $s = a_1 a_2 \ldots a_m$ where the $a_i$ are single symbols and $m > k$. Let $q_1, q_2, \ldots, q_m$ be the state of $M$ after it places this occurrence of $a_i$ onto the stack so that, after this point, $a_i$ is never removed from the stack until after this subcomputation. Since $m > k$, there must be $i < j$ such that $a_i = a_j$ and $q_i = q_j$. Set

$$s_2 = a_{i+1}\, a_{i+2} \ldots a_j$$

and then define $s_1$ and $s_3$ by the equation $s = s_1\, s_2\, s_3$. Define $r_2$ to be the portion of input consumed while pushing $s_2$ onto the stack, and then define $r_1$ and $r_3$ by the equation $r = r_1\, r_2\, r_3$. It is then straightforward to show that the conclusion of the lemma holds.□

Our second pumping lemma for RPDAs draws the same conclusion as a weak form of the pumping lemma for context-free languages. Since the pumping lemma for context-free languages will be used in the proof of the second pumping lemma for RPDAs, we reproduce the context-free version for reference.

**Weak Pumping Lemma for CFLs.** If $L$ is a context-free language, then there is a constant $k$, depending on $L$, such that the following holds: If $z \in L$ and *length* $(z) > k$, then $z$ can be written in the form $z = uvwxy$ where either $v$ or $x$ is nonempty and $uv^i\, wx^i\, y \in L$ for all $i \geq 0$.

The following version of the pumping lemma for RPDAs makes the *identical* conclusion as the above pumping lemma for context-free languages.

**Pumping Lemma 2.** If $L$ is an RPDA language, then there is a constant $k$, depending on $L$, such that the following holds: If $z \in L$ and *length* $(z) > k$, then $z$ can be written in the form $z = uvwxy$, where either $v$ or $x$ is nonempty and $uv^i\, wx^i\, y \in L$ for all $i \geq 0$.

**Proof.** Let $M$ be an RPDA accepting $L$ and let $k$ be as in the Pumping Lemma 1. We decompose $L$ into two languages so that $L = L_1 \cup L_2$. Define $L_1$ to be the set of all strings $z$ in $L$ such that the Pumping Lemma 1 applies to at least one accepting computation on $z$. In other words, $z$ is in $L_1$ if and only if there is a computation of $M$ of the form

$$(q_0, z, Z_0) \vdash^* (p_1, rst, \downarrow \beta) \vdash^* (p_2, st, s^R \downarrow \beta) \vdash (p_3, t, \beta) \vdash^* (p_f, \Lambda, \gamma)$$

where $q_0$ is the start state, $p_f$ is an accepting state, and *length(s)* $> k$. By Pumping Lemma 1, it follows that the conclusion of Pumping Lemma 2 applies to all strings in $L_1$. (In this case we can even conclude that $x$ is always nonempty. However, we will not be able to make such a conclusion for strings in $L_2$.)

$L_2$ is defined as the set of all strings accepted by a particular ordinary PDA $M_2$ that simulates many of the computations of the RPDA $M$. Define $M_2$ to mimic the computation of $M$ but to buffer the top $k + 1$ symbols of the stack in its finite-state control, and make the following modifications to ensure that it is an ordinary PDA: if $M_2$ has to mimic a check-copy move that matches $k$ or fewer symbols above the marker $\downarrow$, it does so using the stack buffer in its finite-state control. If $M_2$ ever needs to mimic a check-copy move that matches more than $k$ stack symbols above the $\downarrow$, it aborts the computation in a nonaccepting state. $M_2$ can tell if it needs to abort a computation by checking the finite stack buffer in its finite-state control. If the buffer contains $k + 1$ symbols but no marker $\downarrow$, and if $M$ would do a check-copy move, then $M_2$ aborts its computation.

By definition, $L = L_1 \cup L_2$. (The sets $L_1$ and $L_2$ need

not be disjoint, since $M$ may be nondeterministic, and hence, a given string may have two different accepting computations. However, this does not affect the argument.) Because it is accepted by an ordinary PDA, $L_2$ is a context-free language, and the Pumping Lemma for context-free languages holds for it and some different $k$. Hence, by redefining $k$ to be the maximum of the $k$s for $L_1$ and $L_2$, we can conclude that the Pumping Lemma 2 holds for $L = L_1 \cup L_2$ and this redefined $k$.□

The next theorem and its proof using the pumping lemma illustrates the fact that RPDAs, like context-free grammars, can, in some sense, check only "two things at a time."

**Theorem 6.** $L = \{a^n b^n c^n \mid n \geq 0\}$ is not an RPDA language.

**Proof.** Suppose $L$ is an RPDA language. We will derive a contradiction. By Pumping Lemma 2, there is a value of $n$ and strings $u$, $v$, $w$, $x$, and $y$ such that $a^n b^n c^n = uvwxy$ with either $v$ or $x$ nonempty and such that $uv^i wx^i y \in L$ for all $i \geq 0$. A straightforward analysis of the possible cases leads to the conclusion that $uv^2 wx^2 y$ is not in $L$, which is the desired contradiction.□

Since it is known that the language $L$ in Theorem 6 is a TAG language, and since the TAG languages are included in the indexed languages, we obtain the following corollaries. The second corollary is the promised completion of the proof of Theorem 2.

**Corollary.** There is a TAG language that is not an RPDA language.

**Corollary.** There is an indexed language that is not an RPDA language.

There are many versions of the pumping lemma for context-free languages. (See Ogden 1968; Harrison 1978; Hopcroft and Ullman 1979.) Most versions make the additional conclusion that *length(vwx)* $\leq k$. Often one can prove that a language is not context-free without using this additional conclusion about the length of $vwx$. In other words, we can often prove that a language is not context-free by using only the weak form of the pumping lemma given above. One such language is the one given in Theorem 6. If you review the proof of Theorem 6, then you will see that all we needed was the Pumping Lemma 2. Moreover, that pumping lemma has the identical conclusion as that of the Weak Pumping Lemma for context-free languages. This leads us to the following informal metatheorem:

**Metatheorem.** If $L$ can be proven to not be context-free via the Weak Pumping Lemma for CFLs, then $L$ is not an RPDA language.

This is not an official theorem since the phrase "via the Weak Pumping Lemma" is not mathematically precise. However, the metatheorem is quite clear and quite clearly valid in an informal sense. It can be made precise, but that excursion into formal logic is beyond the scope of this paper.

To see the limits of this metatheorem, note that the language $\{ww \mid w \in \{a, b\}^*\}$ is an RPDA language, and so to prove that it is not a context-free language, we should need more than the Weak Pumping Lemma. Indeed, one cannot get a contradiction by assuming only that the Weak Pumping Lemma applies to this language. A proof that this language is not context-free must use some additional fact about context-free languages, such as the fact that we can assume that *length(vwx)* $\leq k$, where $vwx$ is as described in the pumping lemma.

The metatheorem is another indication that the RPDA languages are only a small extension of the context-free languages. If it is easy to prove that a language is not context-free (i.e., if the language is "very noncontext-free"), then the language is not an RPDA language either.

## 7 A Construction Missed by the Model

As we have already noted, both Dutch and Swiss-German contain constructions consisting of a string of nouns followed by an equal (or approximately equal) number of verbs. Hence these languages contain substrings of the form

$$N_1 N_2 \cdots N_n V_1 V_2 \cdots V_n$$

In the case of Swiss-German, additional agreement rules suffice to show that these constructions are beyond the reach of context-free grammar, although not beyond the reach of RPDAs. (See the discussion of Shieber 1985 earlier in this paper.) Because Dutch lacks the strong agreement rule present in Swiss German, the same proof does not apply to Dutch. Manaster-Ramer (1987) describes an extension of this construction within Dutch and argues that this extension takes Dutch beyond the weak generative capacity of context-free grammar. Although we are admittedly oversimplifying the data, the heart of his formal argument is that two such strings of verbs may be conjoined. Hence, Dutch contains substrings that approximate the form

$$N_1 N_2 \cdots N_n V_1 V_2 \cdots V_n \textit{ en ('and')} V_1 V_2 \cdots V_n$$

The Dutch data support only the slightly weaker claim that the number of nouns is less than or equal to the number of verbs. Hence, Manaster-Ramer's argument is, in essence, that Dutch contains a construction similar to the following formal language:

$$L = \{a^i b^j c^j \mid i \leq j\}$$

He uses this observation to argue, via the Pumping Lemma for Context-Free Languages, that Dutch is not a context-free language. A careful reading of his argument reveals that, with minor alterations, the argument can be made to work using only the Weak Pumping Lemma. Hence by the metatheorem presented here (or a careful review of his proof), it follows that his argument generalizes to show that the language $L$ is not an RPDA language. Hence, if one accepts his data, the same argument shows that Dutch is not an RPDA language.

The RPDA model could be extended to take account of this and similar natural language constructions missed by the model. One possibility is simply to allow the RPDA to check an arbitrary number of input strings to see if they are finite-state transductions of the string above the marker $\downarrow$. There are a number of ways to do this. However, it seems preferable to keep the model clean until we have a clearer idea of what constructions, other than reduplication, place natural language beyond the reach of context-free grammar. The RPDA model, as it stands, captures the notion of context-free grammar plus reduplication, and that constitutes one good approximation to natural language string sets.

## 8 Reduplication Grammars

Although we do not have a grammar characterization of RPDA languages, we do have a grammar class that is an extension of context-free grammar and that is adequate for a large subclass of the RPDA languages. The model consists of a context-free grammar, with the addition that the right-hand side of rewrite rules may contain a location for an unboundedly long reduplication string of terminal symbols (as well as the usual terminal and nonterminal symbols).

**Definition.** A **reduplication context-free grammar (RCFG)** is a grammar consisting of terminal, nonterminal, and start symbols as in an ordinary context-free grammar, but instead of a finite set of productions, it has a finite set of rule schemata of the following form: $(A \rightarrow \alpha, T)$ where $A$ is a nonterminal symbol, $\alpha$ is a string of terminal and/or nonterminal symbols, and where $T$ is a finite-state transducer. (Thus, $A \rightarrow \alpha$ is an ordinary context-free rule, but it will be interpreted differently than normal.)

The **production set** associated with the schema $(A \rightarrow \alpha, T)$ is the set of all context-free rules of the form: $A \rightarrow ww'\alpha$, where $w$ is a string of terminal symbols, and $w'$ is obtained from $w$ by applying the finite-state transduction $T$ to the string $w$.

The **next step relation** $\Rightarrow$ for derivations is defined as follows:

$\alpha \Rightarrow \beta$ if there is some context-free rule in some production set of some rule schema of the grammar such that $\alpha \Rightarrow \beta$ via this rule in the usual manner for context-free rewrite rules. As usual, $\overset{*}{\Rightarrow}$ is the reflexive-transitive closure of $\Rightarrow$.

The language generated by an RCFG, $G$, is defined and denoted in the usual way: $L(G) = \{ w \mid w$ a string of terminal symbols and $S \overset{*}{\Rightarrow} w\}$, where $S$ is the start symbol.

Notice that an RCFG is a special form of infinite context-free grammar. It consists of a context-free grammar with a possibly infinite set of rewrite rules, namely the union of the finitely many production sets associated with the schemata. However, there are very severe restrictions on which infinite sets of productions are allowed. Also notice that RCFGs generalize context-free grammars. If we take $T$ to be the transduction that accepts only the empty string as input and output, then the set of productions associated with the schema $(A \rightarrow \alpha, T)$ consists of the single context-free production $A \rightarrow \alpha$. In particular, every context-free grammar is (except for notational detail) an RCFG.

Recall that a context-free grammar in Greibach Normal Form is one in which each production is of the form

$$A \rightarrow a\alpha$$

where $a$ is a terminal symbol and $\alpha$ is a string consisting entirely of nonterminals. It is well known that every context-free language can be (weakly) generated by a context-free grammar in Greibach Normal Form. The schemata described in the definition of RCFGs have some similarity to context-free rules in Greibach Normal Form, except that they start with a reduplication string, rather than a single terminal symbol, and the remaining string may contain terminal symbols. Also the leading reduplication string may turn out to be the empty string. Thus, these are very far from being in Greibach Normal Form. Yet, as the proof of the next result shows, the analogy to Greibach Normal Form can sometimes be productive.

**Theorem 7.** If $L$ is a reduplication substitution of a context-free language, then there is an RCFG $G$ such that $L = L(G)$.

**Proof.** Let $G'$ be a context-free grammar, $T$ a finite-state transduction and $a$ a symbol such that $L$ is obtained from $L(G')$ via $T$ by substituting reduplication strings for $a$. Without loss of generality, we can assume that $G'$ is in Greibach Normal Form. The RCFG $G$ promised in the theorem will be obtained by modifying $G'$. To obtain $G$ from $G'$ we replace each $G'$ rule of the form

$$A \rightarrow aA_1 A_2 \cdots A_n,$$

where $a$ is the symbol used for the reduplication substitution, by the schema

$$(A \rightarrow A_1 A_2 \cdots A_n, T)$$

The remaining rules of $G'$ are left unchanged except for the technicality of adding a finite-state transduction that accepts only the empty string as input and output, and so leaves the rule unchanged for purposes of generation. A routine induction then shows that the resulting RCFG $G$ is such that $L(G) = L$.□

Parsing with an RCFG does not require the full power of an RPDA, but only requires the restricted type of RPDA that is described next.

**Definition.** A **simple RPDA** is an RPDA such that, in any computation:

(i) there is at most one occurrence of the marker $\downarrow$ in the stack at any one time, and

(ii) as long as the marker symbol $\downarrow$ is in the stack, the RPDA never removes a symbol from the stack.

More formally, an RPDA $M$ is a *simple RPDA* provided that the following condition holds: if the instruction $(p, push\ \alpha, \Delta) \in \delta(q, a, Z)$ can ever be used

when $\downarrow$ is in the stack, then $\alpha = \beta Z$ for some $\beta$ and $\downarrow$ does not occur in $\alpha$.

Like Theorem 1, the following equivalence for simple RPDAs is trivial to prove by adapting the proof of the same result for ordinary PDAs.

**Theorem 8.** A language $L$ is accepted by some simple RPDA by final state if and only if it is accepted by some (typically different) simple RPDA by empty store.

The next theorem says that RCFGs are equivalent to simple RPDAs.

**Theorem 9.** For any language $L$, $L$ is generated by some RCFG if and only if $L$ is accepted by some simple RPDA.

**Proof.** Suppose that $G$ is an RCFG such that $L = L(G)$. We can construct a simple RPDA that accepts $L(G)$. All we need do is adapt the standard nondeterministic top-down algorithm for accepting a context-free language by empty store on an ordinary PDA. We then obtain a simple RPDA that accepts $L(G)$ by empty store. The details follow.

The RPDA starts with the start nonterminal in the stack and proceeds to construct a leftmost derivation in the stack. If a nonterminal $A$ is on the top of the stack, then it nondeterministically chooses a schema ( $A \rightarrow \alpha$, $T$) and does all of the following:

1. Pops $A$ and pushes $\alpha$. (As usual, the symbols of $\alpha$ go in so the leftmost one is on the top of the stack.)
2. Pushes the marker symbol $\downarrow$ onto the stack.
3. Nondeterministically advances the input head past some string $w$ while simultaneously computing a string $w'$ such that $w'$ is a finite-state transduction of $w$ via $T$. The string $w'$ is pushed onto the stack as it is produced.
4. Executes a check-copy move to verify that $w'$ is an initial segment of the remaining input, thereby also using up the input $w'$.

If the top symbol is a terminal and there is no $\downarrow$ in the stack, then it simply matches the stack symbol to the input symbol, consuming both the stack symbol and the input symbol.

A routine induction shows that the RPDA accepts exactly the language $L = L(G)$.

Conversely, suppose that $M$ is a simple RPDA such that $L(M) = L$. Without loss of generality, we will assume that $M$ always pushes at least one symbol on the stack after pushing the marker symbols $\downarrow$, that every marker symbol $\downarrow$ on the stack is eventually used in a copy-check move, and that the marker symbol $\downarrow$ is not left in the stack at the end of any accepting computation. We reprogram $M$ to obtain an ordinary PDA $M'$ that accepts a different but related language $L'$. $M'$ is defined as follows: $M'$ has all the input symbols of $M$ plus one new symbol, denoted $<q, p>$, for each pair of $M$ states $(q, p)$. Intuitively, a new symbol $<q, p>$ is used to stand in for a reduplication string that $M$ would process starting in state $q$ and ending up in state $p$ after a successful check-copy move. $M'$ mimics $M$ step by step as long as $M$ would not have the marker $\downarrow$ in the stack and as long as the input is not one of the new symbols $<q, p>$. If $M'$ reads a new symbol $<q, p>$, and $M'$ is simulating $M$ in the state $q$, then $M'$ guesses an input symbol $a$ of $M$ and simulates $M$ on input $a$. If $M$ would consume the input symbol $a$ without pushing the marker $\downarrow$ on the stack, then $M'$ aborts its computation. If $M$ would eventually push the marker $\downarrow$ on the stack while scanning (and possibly consuming) $a$, then $M'$ continues to simulate $M$, guessing additional input symbols for $M$ until it needs to simulate $M$ performing a check-copy move. At this point it assumes that the check-copy move succeeds. If that simulated check-copy move leaves the simulated $M$ in the simulated state $p$, then $M'$ consumes $<q, p>$ and continues the simulation of $M$. If any of these conditions are not met, then $M'$ simply aborts its computation.

Remember that, intuitively, a new symbol $<q, p>$ is used to stand in for a reduplication string that $M$ would process starting in state $q$ and ending up in state $p$ after a successful check-copy move. For any state $q$ in which $M$ would push $\downarrow$ on the stack, $M$ will go on to push a finite-state transduction of the input onto the stack until it wants to execute a check-copy move. Let $T(q, p)$ be that finite-state transducer with start state $q$ and the single accepting state $p$ such that $T$ simulates $M$ starting in state $q$ and pushing symbols on the stack and such that $M$ accepts if and only if it ends the simulation in a state that allows a check-copy move that will leave $M$ in state $p$. (Aside from start and accepting state, all the $T(q, p)$ are essentially the same transducer.) Now, $M'$ accepts some context-free language $L'$ with the following property:

(A) Suppose $x_0 < q_1, p_1 > x_1 < q_2, p_2 > x_2 \cdots < q_n, q_n > x_n$ is such that each $x_i$ contains no new symbols and suppose that the strings $u_i$ and $v_i$ $(i \leq n)$ are such that each $v_i$ is a finite-state transduction of $u_i$ by $T$ $(q_i, p_i)$. Under these assumptions, $x_0 < q_1, p_1 > x_1 < q_2, p_2 > x_2 \cdots < q_n, q_n > x_n \in L' = L(M')$ if and only if $x_0\, u_1\, v_1\, x_1\, u_2\, v_2 \cdots u_n\, v_n\, x_n \in L(M)$

Finally let $G'$ be a context-free grammar in Greibach Normal Form for the context-free language $L'$. Construct an RCFG, $G$, as follows:

(i) For each rule of $G'$ of the form $A \rightarrow < q, p > A_1\, A_2 \ldots A_n$ add the following schema to $G$:
($A \rightarrow A_1\, A_2 \ldots A_n$, $T(q, p)$)

(ii) For all other rules of $G'$ simply add the rule to $G$ unchanged (except for cosmetic changes in notation to make them look like schemata).

By (A) and a routine induction it follows that $L(G) = L(M)$.□

Theorem 9 makes simple RPDAs sound better behaved than regular RPDAs and if there were no evidence to the contrary, the weaker model would be preferred. However, the fact that natural languages can

have complicated phrase structures embedded within a reduplication construction indicates that simple RPDAs may not be adequate for natural language syntax. If one assumes a language like English but with syntactic gender, strong agreement rules, and a well-behaved use of *respectively*, then one can easily see why one might want more power than that provided by a simple RPDA. An example of a kind of sentence that seems beyond the reach of simple RPDAs is the following:

> Tom, who has had three wives, Sally, who has had seven husbands, Mary, who lost John, Hank, and Sammy to cancer, heart disease, and stroke, respectively, and Frank, who had only one wife and lost her last January, are a widower, widow, widow, and widower, respectively.

The natural way to handle these sorts of sentences with an RPDA is to have two markers $\downarrow$ in the stack at once, and we conjecture that a single marker will not suffice.

English does not have the syntactic gender and strong agreement rules that would allow us to prove, via this construction, that English is not context-free. We merely put it forth as an example of a potential natural language situation.

## 9 Summary

We have seen that the RPDA model is very similar to the PDA characterization of context-free languages. Thus from an automata theoretic point of view, RPDA languages are very much like context-free languages. We have seen that both classes have similar closure properties, and so they are similar from an algebraic point of view as well. Moreover, the context-free languages and the RPDA languages have similar pumping lemmas that exclude many of the same unnatural language sets and even exclude them for the same reasons. Hence, the class of RPDAs are only mildly stronger than context-free grammars. However, the model is sufficiently strong to handle the many reduplication constructions that are found in natural language and that seem to place natural language outside of the class of context-free languages. The RPDA languages do not, as yet, have a grammar characterization similar to that of context-free grammar, but the RCFG grammars are context-free like grammars that do capture at least a large subclass of the RPDA languages.

## Acknowledgments

This research was supported in part by NSF grant DCR-8604031. Bill Chen and Alexis Manaster-Ramer provided a number of useful discussions on this material. A preliminary version of this work was presented at the Workshop on Mathematical Theories of Language, LSA Summer Institute, Stanford University, Summer 1987. Comments of the workshop participants, particularly those of Aravind Joshi, K. Vijay-Shanker, and David Weir, helped shape the current version of this work. A number of remarks by two anonymous referees also help in the preparation of the final draft of this paper. I express my thanks to all these individuals for their help in this work.

## References

Aho, Alfred V. 1968 Indexed Grammars—an Extension to Context Free Grammars. *Journal of the Association for Computing Machinery* 15: 647–671.

Aho, Alfred V. 1969 Nested-Stack Automata. *Journal of the Association for Computing Machinery* 16: 383–406.

Bar-Hillel, Y. and Shamir, E. 1964 Finite State Languages: Formal Representations and Adequacy Problems. In: Bar-Hillel, Y. (ed.), *Language and Information*. Addison-Wesley, Reading, MA: 87–98.

Bresnan, J.; Kaplan, R. M.; Peters, S. and Zaenen, A. 1982 Cross-Serial Dependencies in Dutch. *Linguistic Inquiry* 13: 613–635.

Culy, Christopher 1985 The Complexity of the Vocabulary of Bambara. *Linguistics and Philosophy* 8: 345–351.

Gazdar, Gerald 1985 Applicability of Indexed Grammars to Natural Languages, Report No. CSLI-85-34. Center for the Study of Language and Information. Palo Alto, CA.

Gazdar, Gerald and Pullum, Geoffrey K. 1985 Computationally Relevant Properties of Natural Languages and their Grammars. *New Generation Computing* 3: 273–306.

Harrison, Michael A. 1978 *Introduction to Formal Language Theory*. Addison-Wesley, Reading, MA.

Hopcroft, John E. and Ullman, Jeffrey D. 1979 *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Huybregts, M. A. C. 1976 Overlapping Dependencies in Dutch. *Utrecht Working Papers in Linguistics* 1: 24–65.

Joshi, Aravind K. 1985 Tree Adjoining Grammars: How Much Context-Sensitivity is Required to Provide Reasonable Structural Descriptions? In: Dowty, D. R.; Karttunen, L. and Zwicky, A. M. (eds.), *Natural Language Processing: Psycholinguistic, Computational, and Theoretic Perspectives*. Cambridge University Press, New York, NY.

Joshi, Aravind K. 1987 An Introduction to Tree Adjoining Grammars. In: Manaster-Ramer, A. (ed.), *Mathematics of Language*. John Benjamins, Philadelphia, PA: 87.

Joshi, Aravind K. 1989 Processing Crossed and Nested Dependencies: An Automaton Perspective on the Psycholinguistic Results. Preprint, Department of Computer and Information Sciences, University of Pennsylvania, Philadelphia, PA.

Joshi, Aravind K.; Vijay-Shanker, K. and Weir, D. J. 1989 The Convergence of Mildly Context-Sensitive Grammar Formalisms. Report MS-CIS-89-14, LINC LAB 144. Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA.

Kac, M. B., Manaster-Ramer, A. and Rounds, W. C. 1987 Simultaneous-Distributed Coordination and Context-Freeness. *Computational Linguistics* 13: 25–30.

Manaster-Ramer, Alexis 1983 The Soft Formal Underbelly of Theoretical Syntax. *Papers from the Nineteenth Regional Meeting*. Chicago Linguistic Society 254–262.

Manaster-Ramer, Alexis 1986 Copying in Natural Languages, Context-Freeness, and Queue Grammars. *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics* 85–89.

Manaster-Ramer, Alexis 1987 Dutch as a Formal Language. *Linguistics and Philosophy* 10: 221–246.

Ogden, W. F. 1968 A Helpful Result for Proving Inherent Ambiguity. *Mathematical Systems Theory* 2: 191–194.

Pollard, Carl J. 1984 *Generalized Phrase Structure Grammars, Head Grammars, and Natural Languages*. Ph.D. thesis, Stanford University, Stanford, CA.

Postal, Paul 1964 Limitations of Phrase Structure Grammars. In: Fodor, J.A.; and Katz, J.J. (eds.), *The Structure of Language: Readings in the Philosophy of Language*. Prentice-Hall, Englewood Cliffs, N.J.: 137–151.

Pullum, Geoffrey K. and Gazdar, G. 1982 Natural Languages and Context Free Languages. *Linguistics and Philosophy* 4: 471–504.

Roach, Kelly 1987 Formal Properties of Head Grammars. In: Manaster-Ramer, A. (ed.), *Mathematics of Language*. John Benjamins, Philadelphia, PA : 293–347.

Rounds, William C.; Manaster-Ramer, A.; and Friedman, J. 1987 Finding Natural Language a Home in Formal Language Theory. In: Manaster-Ramer, A. (ed.), *Mathematics of Language*. John Benjamins, Philadelphia, PA : 87-114.

Salomaa, A. 1973 *Formal Languages*. Academic Press, New York, NY.

Shieber, Stuart M. 1985 Evidence Against the Context-Freeness of Natural Language. *Linguistics and Philosophy* 8: 333–343.

Steedman, M. J. 1987 Combinatory Grammars and Parasitic Gaps. *Natural Language and Linguistic Theory* 5: 403–439.

Steedman, M. J. 1988 Combinators and Grammars. In: Oehrle, R.; Bach, E.; and Wheeler, D. (eds.), *Categorial Grammars and Natural Language Structures*. Reidel, Dordrecht, The Netherlands : 417–442.

Vijay-Shanker, K.; Weir, D. J.; and Joshi, A. K. 1987 On the Progression from Context-Free to Tree Adjoining Languages. In: Manaster-Ramer, A. (ed.), *Mathematics of Language*, John Benjamins, Philadelphia, PA : 389–401.

Weir, D.J.; Vijay-Shanker, K.; and Joshi,A. K. 1986 The Relationship of Tree Adjoining Grammars and Head Grammars. *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, New York, NY : 67–74.