Reactivity and Grammars: An Exploration

Howard Barringer, David Rydeheard*
EMAIL: {Howard.Barringer, David.Rydeheard}@manchester.ac.uk

Dov Gabbay†
EMAIL: Dov.Gabbay@kcl.ac.uk

July 16, 2009

Abstract

We consider the relationship between grammars and formal languages, exploring the following idea: Normally, when considering the process of deriving a string using a grammar, all structures remain fixed except for the string which is changed only by replacement of substrings. By introducing a more dynamic view of this process, we may allow the grammar to change in various ways as the derivation proceeds, or we may change the notion of application of a rule to a string, or the intermediate strings may be modified between application of rules. We call these more dynamic approaches to language generation 'reactive grammars' and explore, in this paper a range of such reactivities. Some of these are related to previously introduced notions of generative grammars, others appear to be new. Reactivity of computational structures has been explored in other areas, e.g. in Kripke structures and in the general areas of evolvable and adaptive systems.

1 Background and motivation

A standard approach to defining models of computation is to introduce fixed structures over which computation may take place. Such structures include graphs, algebras, grammars, automata, etc. An alternative is to allow the structures themselves to change as computation proceeds. Such computational systems are called 'evolutionary', 'adaptive' or (the term we shall use) 'reactive'. The advantages of this additional flexibility are several: (1) they allow us to model desirable aspects of computation in dynamic settings where we require systems which may reconfigure themselves during computation, (2) they allow us to model the influence of external activity under which computation may have to adapt by changing computational structure, (3) they allow us to re-examine the relationship of a computational model with its semantics. The semantics may be a function computed, or a language generated or recognised, or a logical expression satisfied (e.g. for modal logics and Kripke models). The extra degree of freedom that reactivity introduces allows us additional mechanisms for generating a required semantics. It is this feature of reactive systems that we consider in this paper, where we explore ideas of reactivity in grammars and the relationship of these with the generation of formal languages.

For grammars, reactivity is realised through transformations that interact with the derivation steps by which a grammar generates a word in a language. The interaction may for example, at each step of a derivation, control which grammar rules may be applied, or change the notion of application of a rule to a string, or modify rules as they are used, or transform the intermediate strings as they are generated in a derivation. Introducing such reactive elements into the definitions of grammars allows us to re-examine the mathematical structures involved in language recognition and generation, to relate languages according to their reactive structure as well as their production rules, and to model more dynamically changing computational systems.

Before exploring reactivity in grammars, we briefly consider reactivity in the context of Kripke structures and automata. This is related to some of the forms of reactivity in grammars that we introduce later and has already been studied in [8]. Examples drawn from this study illustrate how different forms of reactivity interact with the semantics of these structures.

A Kripke struture has the form (S, R, a), where S is a non-empty set of worlds, $R \subseteq S \times S$ is the accessibility relation and $a \in S$ is the actual world. The semantic evaluation of a formula φ in a world

*School of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, UK
†Department of Computer Science, Kings College London, The Strand, London, WC2R 2LS, UK

$t \in S$ is inductively defined and involves values of its subformulas in other worlds s related to t. Thus, for example, we have:

$$t \models \Box\varphi \text{ iff in all accessible worlds } s \text{ (i.e. such that } tRs) \text{ we have } s \models \varphi.$$

The above is a static view, not a reactive view. The structure does not change as we evaluate. If we take a dynamic view of the evaluation process, we imagine the structure as a graph with a directed relation R (so aRb is viewed as a directed connection $a \longrightarrow b$). When we stand at point t and we want to evaluate $\Box\varphi$, we view this as an invitation to stroll along the arrows (accessibility relation R), go to the accessible worlds and check φ there. If φ holds in all of these worlds we report that $\Box\varphi$ holds at t. Now this point of view allows us to say that the model can change as we stroll through it from world to world. This is the idea of reactivity.

We adopt a similar point of view with automata. An automaton can be viewed either as a mathematical table M which, when given a state t and a letter σ can move to another state s chosen from $s \in M(t, \sigma)$. A more reactive view is to see the automaton as having some internal "will" and it can change its table M as it reacts to input. So for example if $\{s\} = M(t, \sigma)$ and the automaton is getting the input σ three times, then s may become 'exhausted' and change to a state $s' \neq s$ when getting the fourth σ. This means that $M(t, \sigma) = \{s'\}$ if σ is a fourth occurrence in the input.

We have a similar situation with grammars or with proofs. We use a rule to rewrite the system and the use of this rule may then change the system. A simple example of such a change is where the rule deactivates itself on use and is then no longer available unless re-activated later.

Note that the reactive change is in response to the history of use and is not a metalevel (possibly time-dependent) intervention. When the changes involve switching rules/arrows on or off, we represent these graphically by double(-headed) arrows, reserving single(-headed) arrows for the transitions, as in the figure below.

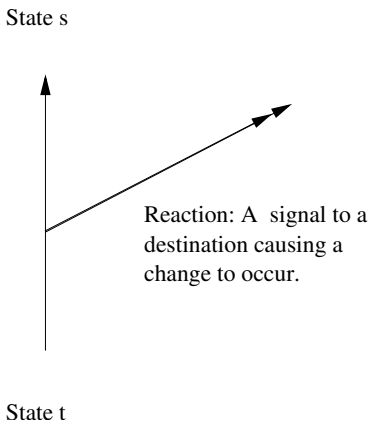

Figure 1: Transition and reaction arrows

Reactivity may be perceived differently from system to system. In some systems, reactivity is perceived as a fault. Consider a washing machine which is continuously being used on the same cycle. If it changes and this overused cycle is no longer available, we see it is as a fault. In some other areas, the reactive point of view is new and offers its own meaning. In normative agent-based systems, the double arrow which changes the system can be viewed as immediate object-level punishment or response to an agent making a forbidden move. With abstract grammars, reactivity is a formal notion. The grammar rules get switched on and off depending on which rules are used. This means that the current rules available at any point depend on the history of which rules have been used up to that point. This is not an unusual point of view for grammars and indeed in the early years of automata and grammar investigations, many new grammars were introduced, adopting a computational point of view yielding variations and generalisations. In this paper, using the general theme of reactivity and of computational variation, we explore a range of reactive grammars, some of which have been considered before; others appear to be new.

Example 1 (Kripke semantics) Figure 2 is an example of a Kripke modal logic model.

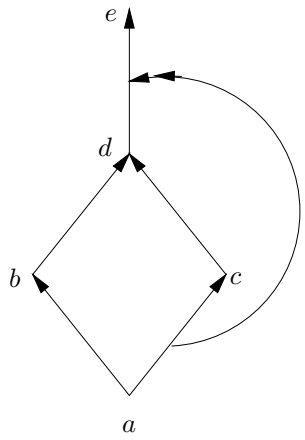

Figure 2: A reactive Kripke structure

For the moment, ignore the double arrow. The single arrows indicate accessibility. Described traditionally, we have $S = \{a, b, c, d, e\}$ and $R = \{(a, c), (a, b), (b, d), (c, d), (d, e)\}$. We also have the double arrow which is really $R' = \{((a, c), (d, e))\}$.

Now consider a formula of modal logic, say

$$A = \Diamond\Diamond\Box\bot.$$

The evaluation process at node a of the model runs as follows:

$$\begin{aligned} a \vDash \Diamond\Diamond\Box\bot \text{ iff } & \exists x\exists y(aRx \wedge xRy \wedge \forall z(yRz \rightarrow z \vDash \bot)) \\ \text{iff } & \exists x\exists y(aRx \wedge xRy \wedge \sim \exists z(yRz)). \end{aligned}$$

This is a static way of looking at the definition. It is a mathematical definition of satisfaction.

Let us adopt a dynamic point of view. We stand at node a and want to check whether $\Diamond\Diamond\Box\bot$ holds at a. We send messages from a to b and from a to c and give them the task to check and report whether $\Diamond\Box\bot$ holds. These messengers in turn send additional messages to d asking whether $\Box\bot$ holds and the people at d send messengers to e to ask whether $\bot$ holds.

The information is passed backwards in the chain and a final decision is made at node a.

Looking at the evaluation process this way allows for several modifications and practical questions:

1. Are the messengers running in parallel or under any coordination or can one person do the job sequentially?
2. How long does it take to go to the various nodes?
3. How much does it cost to
   (a) move from node to node
   (b) get a value at a node
4. How can things go wrong?

The Kripke model is *reactive* if things can change as we traverse the model. The double arrows indicate how things change.

So a double arrow from $(a \rightarrow c)$ to $(d \rightarrow e)$ can indicate that when a messenger moves from a to c then a signal is sent to disconnect the arrow from d to e. So the accessible points that a messenger sees standing at point d depend on the path by which the point was reached. If the journey was via b then e is accessible. If the journey was via c, then the connection $d \rightarrow e$ is cancelled and the step from d to e is not available. Thus in model $\mathbf{m} = (S, R \cup R', a)$, we do have $a \vDash \Diamond\Diamond\Box\bot$, because there is a way to get to d (via c) and in this case $d \vDash \Box\bot$.

The model $\mathbf{m}$ is equivalent to the model depicted in Figure 3.

The nodes indicate the paths. The circle around abd and acd indicates they are really the same point (i.e. for any atomic q, $abd \vDash q$ iff $acd \vDash q$).

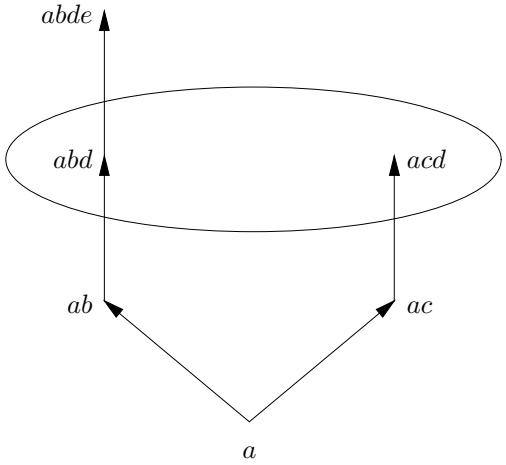

Figure 3:

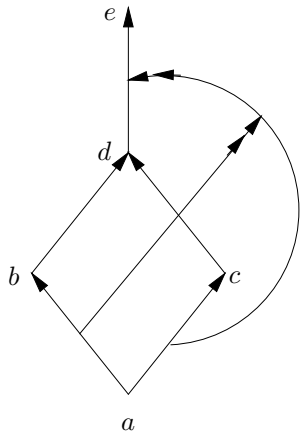

Figure 4:

Such models (with double arrows) are called *reactive models*.

Figure 4 is a more complicated example of a reactive model.

We add in Figure 4 to the double arrow $((a,c),(d,e))$, $R_3 = \{((a,b),((a,c),(d,e)))\}$.

In this model, timing of the movements of the messengers is important. So if the messenger from a to b sets out before the messenger from a to c, then the double arrow $(a,c) \rightarrow (d,e)$ will be cancelled and when the messenger from a to c sets out to c, nothing will happen to the connection $d \rightarrow e$.

We see here that there is a lot of scope in defining the nature of the models and how reactivity behaves.

What view do we take of the models of Figures 2 and 4?

There are several options:

1. Mathematical view, as more complex networks/graphs.

2. Reactive view, as system which react to algorithms (people) using them (traversing them). They change in a prescribed manner while they are used.

□

In fact, as one of the authors has established [8], reactive models have an increased expressivity, in the following sense:

Theorem 1 *There exist modal logics (with a $\Box$) which are complete for a class of reactive models but are not complete for any class of ordinary (Kripke) models.*

Example 2 (Finite automata) An automaton is a state machine which responds nondeterministically to inputs and changes state. An input string is accepted by the automaton at its initial state, if after responding to the string as input it ends up in a terminal state.

An automaton can be represented by a multimodal Kripke model. Figure 5 is an example of such an automaton with two letter inputs $\{1, 2\}$ and states $\{a, b\}$.

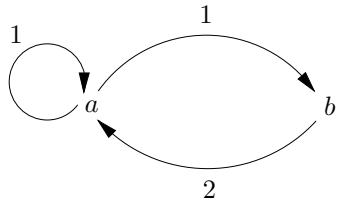

Figure 5: A simple automaton

The initial state is a the final state b. Let q be an atom and let q hold only at the final state. The model of Figure 5 has two relations $R_1 = \{(a, a), (a, b)\}$ and $R_2 = \{(b, a)\}$. For any sequence of numbers from $\{1, 2\}$ of the form $x_1, \ldots, x_n$ we check whether the model satisfies $a \vDash \Diamond_{x_1}\Diamond_{x_2} \ldots \Diamond_{x_n} q$. It accepts words of the form $1^{m_1}21^{m_2}2 \ldots 1^{m_{k-1}}21^{m_k}$ with $m_1, \ldots, m_k \geq 1$.

If we transfer the reactive idea to automata then, through the correspondence above, we get that a reactive automaton can change with every move into a new automaton, as some of its transitions change. So given automaton $\mathcal{A}$ at state s, it changes when seeing word σ into a new automaton $\mathcal{A}'$ at state s'.

One conclusion is immediate, any reactive finite automaton is equivalent to another nonreactive ordinary automaton. □

We do gain something though:

Theorem 2 *Every ordinary automaton with n states $n = p_1, \ldots, p_k$ is equivalent to a reactive automaton with m states, $m = p_1 + \ldots + p_k$.*

The idea of reactivity does not allow automata to recognise more sets of words but it does allow us to economize on states.

Example 3 (An infinite state reactive automaton) The Kripke model point of view of automata does allow for a uniform presentation of automata. Consider an (infinite state) automaton for recognising all words of the form $0^n1^n, n = 0, 1, 2, \ldots$ ($n = 0$ is the empty word). We need some infinity here, that of a stack getting fuller and fuller and then emptier. We can construct this as a reactive model as in Figure 6. The thicker (red) arrows are the transitions of the model; the thinner (blue) arrows are reactivities. Furthermore, all arrows are, by default, self-cancelling, i.e. whenever we traverse an arrow, the arrow is automatically cancelled. Initially, we assume all arrows are off apart from the first upward red arrow from X_0 to X_1. We start walking up from X_0 to X_1. The blue arrows activate the next upward arrow and the return downwards.

Let $q = \Box\bot$. Let X_0 be both the starting point and the terminal point. Then we have that exactly wffs of the form $\Diamond_1^n\Diamond_2^n\Box\bot$ hold at X_0. □

Example 4 (An infinite state with infinite branching automaton) Consider the model of Figure 7. This corresponds to an infinite reactive automaton which accepts (or generates) $0^n1^n2^n \ldots (2k + 1)^n, n = 0, 1, 2, \ldots, k = 0, 1, 2, \ldots$.

There are states X_i for $i = 0, 1, \ldots$ and the associated towers of the thicker (red) arrows accept (or generate) the symbols $0, 1, 2, \ldots$, i.e. traversal of the bottom leftmost red arrow from X_0 to X_1 will accept (or generate) a 0, whereas traversal of the red arrow from X_0 to X_1 in the column above 2 will accept (or generate) a 2. State X_0 is the final state. The thinner (blue) arrows are reactivities that flip the activation status of both red and blue arrows. First note that all arrows are self-cancelling, i.e. as an arrow is traversed its activation status is switched off (and an arrow can only be traversed if its activation status is on). Also, for ease of presentation, we have used a convention that a blue arrow flipping the activation of a red arrow will also flip the activation of all of its associated blue arrows. Thus, for example, the blue arrow labelled (1) will flip the activation of the arrow $X_1 \rightarrow X_0$ in the 1 column. Blue arrows are also used to flip the activation status of other blue arrows as, for example, is

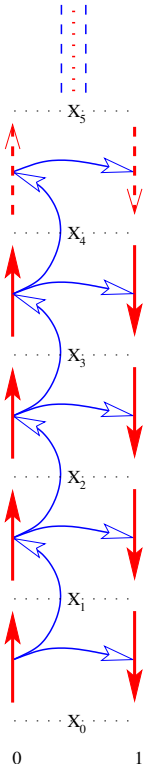

Figure 6: Reactive model for 0^n1^n

the case for the arrow labelled (2). Thus, when the red arrow from X_2 to X_1 is traversed in column 1, the arrow labelled (2) flips the activation status of the blue arrow labelled (3).

The activation status of all the arrows is off apart from the $X_0 \rightarrow X_1$ arrow in column 0 and its three blue reactivity arrows. The automaton then has the following operational behaviour. It can initially traverse the red $X_0 \rightarrow X_1$ arrow accepting 0. As a consequence the red arrows $X_1 \rightarrow X_2$ in column 0 and $X_1 \rightarrow X_0$ in column 1 are switched on. In addition, the blue arrow from $(X_0, X_1) \rightarrow (X_1, X_2)$ is switched on. The automaton is then able to either accept a further 0, moving one place up the leftmost tower, or generate a 1 by moving across and down in the second column (labelled 1). The automaton is thus able to accept, say, a sequence of n 0s. Without loss of generality let $n = 4$ and assume that the next symbol accepted is a 1. The machine thus starts moving down the second column labelled 1. However, in order to accept only words of the form $0^4 1^4 2^4 3^4 \ldots$ it must remember the turnaround point. This is encoded by disabling the reactivity $(X_3, X_4) \rightarrow (X_4, X_5)$ in the column labelled 2. Note, of course, that similar switching occurred for all lower transitions, however, as we progress down the 1 tower, those reactivities are switched back again with the effect that the automaton will be able to switch, sequentially, the correct number of up going red arrows in the third column. Note also that all the reactivities labelled (4) will have also been switched so that when the red arrows in the third column are traversed the option of immediately descending in the fourth column is removed up to the turnaround point when the machine is back in state X_4.

We thus claim this infinite reactive automaton accepts $0^n 1^n 2^n \ldots (2k+1)^n, n = 0, 1, 2, \ldots, k = 0, 1, 2, \ldots$.

□

1.1 Some historical comments

This paper is a contribution to a continuing exploration that started with the observation that, in the 'Chomsky hierarchy' [2] of formal languages, many languages, particularly those arising in computer science, are not context-free, and so they are present in the hierarchy as instances of context-sensitive languages, or languages requiring full Turing computability. An alternative treatment for presenting

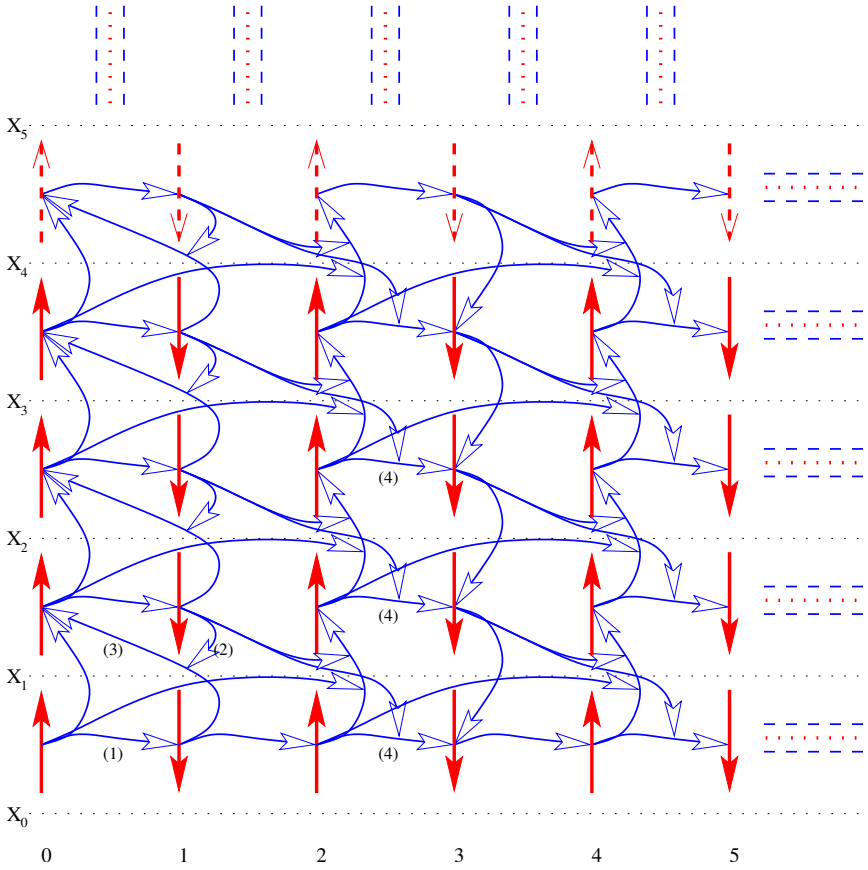

Figure 7: Infinite reactive automaton for $0^n1^n2^n3^n \ldots (2k+1)^n$

and processing many of these languages is to use a context-free grammar together with a mechanism for controlling the choice of derivation step at each point in a derivation. By selecting only certain derivations to be legitimate, various classes of non-context-free languages can be described. Such presentations appear more 'natural' in that they reflect the inherent structure of these languages whose basis is a context-free phrase structure. They also provide alternative (and possibly more efficient) algorithms for processing such languages.

This general approach to grammar is presented in [6] under the name of 'Regulated Rewriting'. The mechanisms for controlling derivations vary from author to author. An early proposal was that of 'matrix grammars' [10] whose rules are grouped in sequences (called 'matrices') and derivation steps iterate through the rules in a sequence. Many other mechanisms have been proposed, amongst which are: indexed grammars [1], programmed grammars [16], regulated grammars [7], grammars with state [11, 14], grammars with control sets [9] and co-operating grammars [13]. Some of these approaches treat the derivation sequences themselves as strings in an additional language (e.g. [9]). Properties of the language of derivation sequences then determine properties of the generated language. Results concerning the equivalence of expressive power of these formalisms can be established in some cases.

In this paper, we add yet more to the number of such mechanisms! This is not really a case of 'the more, the merrier', rather the opposite. However, our emphasis here is somewhat different, although we will investigate the expressivity of the reactive grammars that we consider. We are interested in the general notion of reactivity in computational systems that may change their behaviour and evolve as computation progresses. There is an increasing focus on the development and analysis of computational systems that exhibit 'evolutionary' behaviour, that is, systems that may either compute normally or may invoke evolutionary changes which can modify the structure of the system itself. For various models of computation, these evolutionary changes take very different forms. For automata consisting of (labelled) transition systems, we have already seen some forms of evolutionary change in examples above. As well as changing the transition relation (i.e. the connectivity of nodes), we may change the labelling, or, if

data is stored at nodes, change this data. As a counterpart to reactive automata, we here investigate reactive grammars.

In grammars, the 'normal computation' is the process of substituting non-terminals using rules of a (usually context-free) grammar. The evolutionary steps are the 'reactive elements' which may be introduced into the generation process at various points in the derivation. Reactivity may be present so that (1) at each step of a derivation, we may control which grammar rules may be applied (and this control itself can be of various forms, including additional 'reaction rules' which determine the availability of rules), or (2) we may change the notion of application of a rule to a string, or (3) we may modify rules as they are used, or (4) we may transform intermediate strings as they are generated in a derivation, before applying further rules. We consider some of these cases in this paper. Some of these are equivalent to already introduced forms of grammar, others appear to be new. We also consider examples of what we call 'embedded reactivity' in which substrings of a string can themselves determine changes in either the grammar, the current string or the derivation steps allowed. The extra degree of freedom that reactivity adds allows us to re-examine the mathematical structures involved in language recognition and generation, and to relate languages according to their reactive structure as well as their production rules. We have yet to explore the latter idea.

Reactive grammars have already appeared in applications other than language recognition and generation. For example, in the runtime verification of computational systems, reactive rule-based systems arise naturally as in RuleR [5] (see also the related Eagle [4] and MetateM[3] systems). Other non-linguistic areas where non-context-free grammars appear include graph-based problems [15] and modelling biological systems [12].

Our exposition is driven by example – the examples indicate the complex structure of language generation when production rules of grammars are combined with notions of reactivity. We first consider 'switching grammars' in which the use of a production rule determines which rules are available for the next derivation step. We then consider 'string transformer grammars' in which each production rule of the grammar has an associated string transformer - a transformation of the string resulting from application of the rule. This is a powerful notion of reactivity and we will show that it can be used to model derivation strategies in grammars. Finally, we consider several examples of 'embedded reactivity' in which substrings of a string may determine reactivity. This is all very much an initial exploration of the space of possibilities in this area.

2 Switching Grammars

We begin with a standard definition of grammar:

Definition 1 *A grammar, G, is a four-tuple consisting of a set of non-terminal symbols, N, a set of terminal symbols, Σ, a set of production rules, with each rule of the form $\alpha A\beta \to \gamma$ for $A \in N$, $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, and an initial non-terminal symbol $S \in N$.*

Recall that a grammar is *context-free* if all the production rules are of the form $A \to \gamma$ where $A \in N$ and $\gamma \in (N \cup \Sigma)^*$, and a grammar is *right regular* if all production rules are of the form $A \to wX$ or $A \to w$ with $w \in \Sigma^*$ and $A, X \in N$. A grammar is said to be *ϵ-free* when all production rules have $\gamma \in (N \cup \Sigma)^+$.

To begin the exploration of reactivity in grammars, we introduce a very simple form of reactivity in, what we call, *switching grammars*. Here the application of a production rule can switch production rules of the grammar on or off. These are a special case of programmed grammars [16] where only 'success sets' are associated with production rules. The ϵ-free switching grammars are, effectively, an alternative presentation of (ϵ-free) state grammars [11, 14]. This simple form of reactivity is considered here to illustrate some of the issues which arise in reactive grammars and because it is related to the next form of reactivity that we consider, that of 'string transformer grammars'.

Definition 2 *A switching grammar is a tuple consisting of a finite set of non-terminal symbols, N, a finite set of terminal symbols, Σ, a set of rule labels, R, a set of R-labelled production rules P, each rule being of the form $r : \alpha A\beta \to \gamma, \rho$; for $r \in R$, $A \in N$, $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, $\rho \subseteq R$, together with an initial non-terminal symbol $S \in N$ and an initial rule set $\rho_0 \subset R$. A switching grammar is ϵ-free if its production rules have $\gamma \in (N \cup \Sigma)^+$.*

Definition 3 *Let $G = \langle N, \Sigma, R, P, S, \rho_0 \rangle$ be a switching grammar. G derives the string $s\gamma t$ from $s\alpha A\beta t$ and switches from ρ_i to ρ_j in one step (written as $\langle s\alpha A\beta t, \rho_i \rangle \Rightarrow \langle s\gamma t, \rho_j \rangle$) iff for some $r \in \rho_i$, $r : \alpha A\beta \rightarrow \gamma, \rho_j \subseteq P$. A word $w \in \Sigma^*$ is derived in G iff there is a sequence of single step derivations $\langle w_0, \rho_0 \rangle \Rightarrow \langle w_1, \rho_1 \rangle \Rightarrow \ldots \Rightarrow \langle w_n, \rho_n \rangle$ such that $w_k \in N \cup \Sigma^*$ and $\rho_k \subseteq R$, for $k \in 0..n$, and $S = w_0$ and $w = w_n$. The language generated by G is the set of its derivable words w.*

Let us look at an example of a switching grammar.

Example 5 (A right regular switching grammar) Consider a switching grammar G where $N = \{A\}$, $\Sigma = \{a, b\}$, $R = \{r_0, r_1, r_2, r_3\}$, the production rule set is

$$\{r_0 : A \longrightarrow a, \{r_2, r_3\}, r_1 : A \longrightarrow aA, \{r_2, r_3\}, r_2 : A \longrightarrow b, \{r_0, r_1\}, r_3 : A \longrightarrow bA, \{r_0, r_1\}\}$$

with initial non-terminal $S = A$ and initial rule set $\{r_0, r_1\}$. G generates the language $(ab)^*a \cup (ab)^*$.

Note, (a) this is a right regular grammar, and (b) this grammar generates the same language as the (non-switching) right regular grammar $\langle \{A, B\}, \{a, b\}, \{A \rightarrow a, A \rightarrow aB, B \rightarrow b, B \rightarrow bA\}, A \rangle$. □

The comment in this example suggests the following result.

Theorem 3 *A language is generated by a switching right regular grammar iff it is a regular language.*

Proof Clearly any right regular grammar can be put as a switching regular grammar. Simply form a production rule set P_R by uniquely labelling each production rule of the regular grammar with a label from R and then make R the initial rule set.

The other direction is established in the following manner. Let $G = \langle N, \Sigma, R, P, S, \rho_0 \rangle$. Create a new set of non-terminal symbols $N_R = \{n_r \mid n \in N \text{ and } r \in R\} \cup \{S_R\}$. Create a new production rule set

$$\begin{aligned} Q_R = \; & \{A_r \rightarrow u \mid (r : A \rightarrow u, \rho) \in P \text{ for } u \in \Sigma^*\} \cup \\ & \{A_r \rightarrow vB_s \mid (r : A \rightarrow vB, \rho) \in P \text{ for } v \in \Sigma^* \text{ and } s \in \rho\} \cup \\ & \{S_R \rightarrow S_{r_0} \mid r_0 \in \rho_0\}. \end{aligned}$$

By construction, the grammar $G_R = \langle N_R, \Sigma, Q_R, S_R \rangle$ is regular. Furthermore, G_R generates exactly the same set of words as G. □

More illustrative of the expressivity of switching grammars is the following example which generates the language $\{a^n b^n c^n \mid n \geq 1\}$. This language is a standard example of a language that cannot be generated by unrestricted rule application in a context-free grammar. There are however simple context-sensitive grammars for this language and, as we shall show below, a range of reactive context-free grammars of various forms which also generate the language. As an example of a context-sensitive presentation, consider the following (non-switching) grammar,

$$\begin{aligned} & S \rightarrow aSBC \\ & S \rightarrow aBC \\ & CB \rightarrow BC \\ & aB \rightarrow ab \\ & bB \rightarrow bb \\ & bC \rightarrow bc \\ & cC \rightarrow cc \end{aligned}$$

with initial symbol S. Notice that because of the interchange steps (between B's and C's), generating the word $a^n b^n c^n$ requires $O(n^2)$ derivation steps.

Example 6 (The language $\{a^n b^n c^n \mid n \geq 1\}$) Consider a switching context-free grammar generating this language:

$$G = \langle \{A, B, C\}, \{a, b, c\}, \{r_0, r_1, r_2, r_3, r_4\}, P, A, \{r_0\} \rangle$$

where

$$P = \left\{ \begin{array}{l} r_0 : A \rightarrow BC, \{r_1, r_2\} \\ r_1 : B \rightarrow ab, \{r_3\} \\ r_2 : B \rightarrow aBb, \{r_4\} \\ r_3 : C \rightarrow c, \{\} \\ r_4 : C \rightarrow cC, \{r_1, r_2\} \end{array} \right\}$$

The following is an example derivation:

$$A, \{r_0\} \Rightarrow BC, \{r_1, r_2\} \Rightarrow aBbC, \{r_4\} \Rightarrow aBbcC, \{r_1, r_2\} \Rightarrow aabbcC, \{r_3\} \Rightarrow aabbcc, \{\}$$

After the first step replacing A by BC, the grammar reacts by only allowing rules which replace B to be applied. The second step replaces B by aBb and then only allows the rule that expands C by cC. And so on. Thus the language generated is $\{a^n b^n c^n \mid n \geq 1\}$. □

Theorem 4 *The class of ϵ-free switching context-free grammars is identical to the (ϵ-free) state context-free grammars and hence correspond to the class of context-sensitive languages.*

See Kasai [11] for the relation between state and context-sensitive grammars. The identity of ϵ-free switching context-free grammars and (ϵ-free) context-free state grammars is trivial.

3 Switch reactive grammars

In switch reactive Kripke models, the accessibility of one state from another may be turned on and off. In traversing such a structure only the accessibilities currently on or activated are available. Accessibilities are changed according to additional arcs which determine the reactivity. Such reactive arcs may themselves be turned on and off according to other reactive arcs. This produces a typed hierarchy of arcs, the base arcs being the accessibilities between states. The exact mechanisms for turning arcs on and off (i.e. changing the activation status of arcs) is explored in [8].

We now turn to grammars which have an analogous reactivity, extending that of switching grammars. We call these *switch reactive grammars*. It turns out that there is a range of possibilities which we explore here.

The idea is that associated with each production rule in a grammar is a set of 'reaction rules' which determine what rules (basic production rules and reaction rules) are active (or 'available') at the next step. Reaction rules may themselves have associated reaction rules.

The possibilities here are generated by the following issues:

1. Base rules and reaction rules

We may wish to distinguish the way that rules operate according to whether they are

- base rules, which include the production rules of the grammar, but also may include additional rules for the presentation of grammars,
- reaction rules, whose role is the activation and de-activation of other rules, including reaction rules.

Thus a grammar consists, in general, of a base grammar and a collection of reaction rules.

2. Transient and persistent rules

What happens to an active rule when it is used (or 'fired')? It may

- become de-activated – this is the case of *transient* rules,
- remain activated – this is the case of *persistent* rules.

The transience or persistency of rules may vary either (a) by allowing both forms in a grammar (defining the nature of each rule), or (b) by distinguishing base rules from reaction rules.

3. Operation of reaction rules

There are various possibilities as to how the reaction rules operate: They may

- switch (or 'flip') the activation status of the target rule when the source rule fires – turning on rules that are off, and turning off rules that are on.
- act only 'positively' i.e. turn on target rules that are off and leave target rules on that are on (these may then be turned off by firing, if they are transient).

Let us now look at example of these various cases.

Example 7 (Transient base rules, persistent reactivity and switching reaction rules) Consider a grammar for the language $\{a^n b^n \mid n \geq 1\}$. It has production rules:

$$\begin{aligned} c_0 &: S \to AB \\ c_1 &: A \to a \\ c_2 &: A \to aA \\ c_3 &: B \to b \\ c_4 &: B \to bB \end{aligned}$$

The reaction rules are:

$$\begin{aligned} r_0 &: c_0 \to c_1, c_2 \\ r_1 &: c_1 \to c_2, c_3 \\ r_2 &: c_2 \to c_1, c_4 \\ r_3 &: c_4 \to c_1, c_2 \end{aligned}$$

Let the rules that are initially 'on' be c_0 together with all the reaction rules. Let the base rules be transient and the reaction rules be persistent. Finally, let the reaction rules operate by switching: flipping the on/off status.

Initially, rule c_0 fires (and becomes de-activated). The reaction rule for c_0 is r_0, which thus flips the status of c_1 and c_2, turning both on. At the next step, if c_1 is fired (and is then de-activated), rule c_2 is switched off and rule c_3 switched on, to generate the string ab. Alternatively, if c_2 is fired (and is then de-activated), rule c_1 is switched off and rule c_4 switched on, to generate the string $aAbB$. Then c_1 and c_2 are activated to continue the derivation.

This scheme may be extended to generate the non-context-free language $\{a^n b^n c^n \mid n \geq 1\}$.

The above grammar is directly equivalent to (has the same derivations as) the standard switching grammar:

$$\begin{aligned} c_0 &: S \to AB,\ \{c_0, c_1\} \\ c_1 &: A \to a,\ \{c_3\} \\ c_2 &: A \to aA,\ \{c_4\} \\ c_3 &: B \to b,\ \{\} \\ c_4 &: B \to bB,\ \{c_1, c_2\} \end{aligned}$$

Notice however, that the reaction mechanisms are different. In the case of the switching grammar, at each step, all rules are turned off and only then is the set of rules for the next step turned on. □

In this example then, introducing reaction rules does not result in anything more than standard switching grammars.

Let us now consider an example of a different form.

Example 8 (All rules persistent, switching reaction rules) Here, all rules are persistent and reaction works by 'flipping' the activation state of its target arrow.

$$\begin{aligned} c_0 &: S_0 \to aS_1 \\ c_1 &: S_1 \to bS_2 \\ c_2 &: S_2 \to dS_2 \\ c_3 &: S_2 \to c \end{aligned}$$

The reaction rule is:

$$r : c_2 \to c_3$$

Let all the rules be 'on' initially. Then rule c_3 is 'on' only after an even number of firings of rule c_2. So the language generated is $\{abd^{2n}c \mid n \geq 0\}$.

This reactive grammar is directly equivalent to (i.e. has the same derivations as) the normal grammar:

$$\begin{aligned} c_0 &: S_0 \to aS_1 \\ c_1 &: S_1 \to bS_2 \\ c_2 &: S_2 \to dS_2' \\ c_2 &: S_2' \to dS_2 \\ c_3 &: S_2 \to c \end{aligned}$$

□

In fact, for this form of reactivity, even for higher-order reactivity (when reaction rules act on reaction rules), it appears that the grammar is equivalent, by an expansion and copying technique, to a normal (non-reactive) grammar.

Let us consider another form of switch reactive grammar. In this case, all rules are transient.

Example 9 (All rules transient, switching reaction rules) Let us consider again the language of Example 1, $\{a^n b^n \mid n \geq 1\}$. The base rules are the same:

$$\begin{aligned} c_0 &: S \to AB \\ c_1 &: A \to a \\ c_2 &: A \to aA \\ c_3 &: B \to b \\ c_4 &: B \to bB \end{aligned}$$

with c_0 initially 'on'. Let us see how to add transient reaction rules so that this grammar is restricted to generate the required language. Consider, for example, the rule c_2. Part of its reaction is to switch rule c_4 together with all its reaction rules. But the latter reaction rules include a rule to switch rule c_2 and all its reaction rules...

So far, we have labelled the reaction rules with *rule names*, but they have not been used. Now, in order to describe this dependency of reaction rules on others, the names are crucial and allow us to define this via mutual recursion, as follows:

$$\begin{aligned} r_0 &: c_0 \to c_1, c_2, r_1, r_2 \\ r_1 &: c_1 \to c_2, c_3, r_2 \\ r_2 &: c_2 \to c_1, c_4, r_1, r_4 \\ r_3 &: \\ r_4 &: c_4 \to c_1, c_2, r_1, r_2 \end{aligned}$$

The rule name r_3 labels an empty rule. Notice that r_2 changes the activation of r_4 and vice versa. Initially r_0 is active. A derivation proceeds by firing c_0, which then activates c_1 and c_2 via reaction rule r_0. If c_1 then fires, c_2 is turned off and c_3 turned on via reaction rule r_1, etc.

Notice, that *if no rule names are introduced for the reaction rules* and instead we use a hierarchical assembly of 'reactions' of the form $c_0 \to c_1, c_2$, then the mutual recursion can only be expressed through infinite rules or an infinite set of rules. □

In fact, grammars of the form in Example 3 are equivalent to switching grammars, as we now show.

Example 10 (Example 3 continued) First notice that switching grammars specify only which rules are to be active at the next step, instead of the 'flipping' reactivity used in Example 3. However, we can convert Example 3 to this positive form of reaction, in which case the grammar is:

$$\begin{aligned} c_0 &: S \to AB \\ c_1 &: A \to a \\ c_2 &: A \to aA \\ c_3 &: B \to b \\ c_4 &: B \to bB \end{aligned}$$

$$\begin{aligned} r_0 &: c_0 \to c_1, c_2, r_1, r_2 \\ r_1 &: c_1 \to c_3 \\ r_2 &: c_2 \to c_4, r_4 \\ r_3 &: \\ r_4 &: c_4 \to c_1, c_2, r_1, r_2 \end{aligned}$$

From this combination of base rules and the reaction rules, we construct a switching grammar whose labels are derived from those of the base and reaction rules as follows:

$$\begin{aligned} c_{0r_0} &: S \to AB, \{c_{1r_1}, c_{2r_2}\} \\ c_{1r_1} &: A \to a, \{c_{3r_3}\} \\ c_{2r_2} &: A \to aA, \{c_{4r_4}\} \\ c_{3r_3} &: B \to b, \{\} \\ c_{4r_4} &: B \to bB, \{c_{1r_1}, c_{2r_2}\} \end{aligned}$$

□

In summary, we have explored various types of switch reactive grammars and shown their equivalence to normal non-reactive grammars or to switching grammars. Some of these conversions are economical (do not lead to a great expansion of the grammar). Others (e.g. Example 2) are less so.

4 A reduction result

We present a result showing how, using switch reactive grammars, a linear grammar may be reduced, in terms of the number of non-terminal symbols, to a reactive linear grammar generating the same language. In fact, we establish a 'simulation' between a grammar and its reduction. The reduction in the number of non-terminal symbols is considerable: from k^n to kn. The form of switch reactive grammars used is that all rules are transient (or self-cancelling), i.e. as soon as a rule is fired its activation status is switched off. Reaction rules are switching, i.e. change the activation status of their targets.

Theorem 5 *A linear grammar G with k^n non-terminal symbols can be simulated by a linear switch reactive grammar $\mathbb{T}(G)$ with kn non-terminal symbols.*

Proof outline. Suppose G is a linear grammar with k^n non-terminal symbols – i.e. the non-terminals are in an n-dimensional cube of k-entries in each dimension, so each non-terminal symbol may be represented as an n-tuple of symbols from k sets.

Let $[a^i_j]$ be a matrix of distinct elements for $i = 1, \ldots, n$ and $j = 1, \ldots, k$. Let $x = (x_1, \ldots, x_n)$ be a vector such that $x_i \in \{a^i_1, \ldots, a^i_k\}$. There are exactly k^n such vectors.

We construct a reactive grammar $\mathbb{T}(G)$ with non-terminals $[a^i_j]$ which simulates G.

Let us show how it is done for the case $k = 3$, $n = 5$ (these numbers are chosen simply to allow us to draw the appropriate figures).

The aim is to simulate the effect of the rule of G

$$x \to w_1 y w_2$$

where w_1 and w_2 are terminal letters. (For regular grammars, consider rules of the form $x \to y w_2$ instead.)

Let $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$. The construction is illustrated in Figure 8. The arrows in this diagram are grouped into various types:

Group 1. These are the (red) arrows $(x_i, x_{i+1}) : x_i \to x_{i+1}$, $i = 1, 2, 3, 4$ and $(y_j, y_{j+1}) : y_j \to y_{j+1}$, $j = 1, 2, 3, 4$. They can be either on or off. Each chain of such arrows forms the basis of the encoding of a non-terminal from the original grammar.

Group 2. The possibly multi-headed (blue) arrows are reactivity arrows. The source is an arrow and the target is a set of arrows, e.g. $(\alpha, \{\beta_1, \beta_2, \ldots, \beta_m\})$. This reads as follows: if you use the rule α then flip on or off the activation status of the target rules $\beta_1, \ldots, \beta_m$, e.g. if β_1 is on, flip to off, or if it is off, flip it to on.

Group 3. Other arrows connect arrows of any group to those of any group. For example the multi-headed (green) arrow in the diagram marked (3) denotes the grammar rule $x_5 \to w_1 y_1 w_2$.

Note the following: The vector $x = (x_1, x_2, x_3, x_4, x_5)$ is a non-terminal of G. This is represented in the diagram by the left-hand side, with the vertical arrows, and the hierarchy of curved arrows between them. The last curved arrow, to $(x_4 \to x_5)$, contains all the information of the vector x. So from this arrow, we send arrows to activate y and all its arrows.

How do we simulate $x \to w_1 y w_2$?

As an example, let the non-terminal x be encoded by the vector $(a^1_1, a^2_1, a^3_2, a^4_1, a^5_3)$ of non-terminals from the reduced grammar and y encoded by the vector $(a^1_1, a^2_2, a^3_1, a^4_1, a^5_1)$. Thus the original grammar

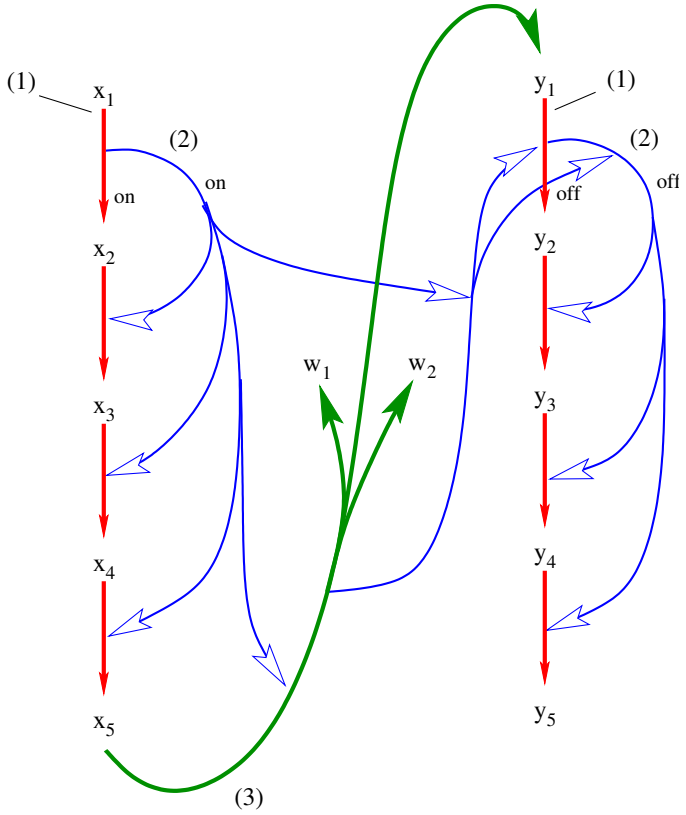

Figure 8: An example of the construction for a rule of G.

rule $x \to w_1 y w_2$ is transformed to the following collection of core grammar rules

$$
\begin{aligned}
a_1^1 &\to a_1^2 \\
a_1^2 &\to a_2^3 \\
a_2^3 &\to a_1^4 \\
a_1^4 &\to a_3^5 \\
a_3^5 &\to w_1 a_1^1 w_2 \\
a_1^1 &\to a_2^2 \\
a_2^2 &\to a_1^3 \\
a_1^3 &\to a_1^4 \\
a_1^4 &\to a_1^5
\end{aligned}
$$

with the associated reactivity rules

$$
\begin{aligned}
&((a_1^1, a_1^2), \{\, (a_1^2, a_2^3), (a_2^3, a_1^4), (a_1^4, a_3^5), \\
&\qquad (a_3^5, w_1 a_1^1 w_2), ((a_3^5, w_1 a_1^1 w_2), \{(a_1^1, a_2^2), ((a_1^1, a_2^2), \{(a_2^2, a_1^3), (a_1^3, a_1^4), (a_1^4, a_1^5)\})\})\}) \\
&((a_3^5, w_1 a_1^1 w_2), \{(a_1^1, a_2^2), ((a_1^1, a_2^2), \{(a_2^2, a_1^3), (a_1^3, a_1^4), (a_1^4, a_1^5)\})\}) \\
&((a_1^1, a_2^2), \{(a_2^2, a_1^3), (a_1^3, a_1^4), (a_1^4, a_1^5)\})
\end{aligned}
$$

We start with the grammar rule $a_1^1 \to a_1^2$ and its associated reactivity rule switched on. Then, for a word W containing the non-terminal a_1^1, the rule $a_1^1 \to a_1^2$ can be applied to produce a new word derived from W by replacing a_1^1 by a_1^2 in the usual way. However, at the same time, the reactivity rule is fired and the activation status of the remaining decoding (core grammar) rules $a_1^2 \to a_2^3$, $a_2^3 \to a_1^4$ and $a_1^4 \to a_3^5$ are switched on, as is the core grammar rule $a_3^5 \to w_1 a_1^1 w_2$ together with its associated reactivity rule. The derivation will then continue to "decode" the original non-terminal x. When the core grammar rule

$a_3^5 \to w_1 a_1^1 w_2$ is applied, its reactivity will switch on the header rule and its reactivity for the decoding of the (original) y nonterminal.

The system is set up with arrows for every rule $u \to w_1' v w_2'$ in the original grammar G. Figure 8 thus illustrates the set-up for the rule $x \to w_1 y w_2$.

Note that if the original grammar is regular, so is the reactive one.

Cycles and choice

Let us consider an encoding of the following two linear rules

$$X \to a \qquad\qquad X \to bX$$

for non-terminal symbol X and terminal symbols a and b, extracted from a grammar with 5^4 non-terminals. The reactive grammar will encode the non-terminals of the grammar as vectors of length 4.

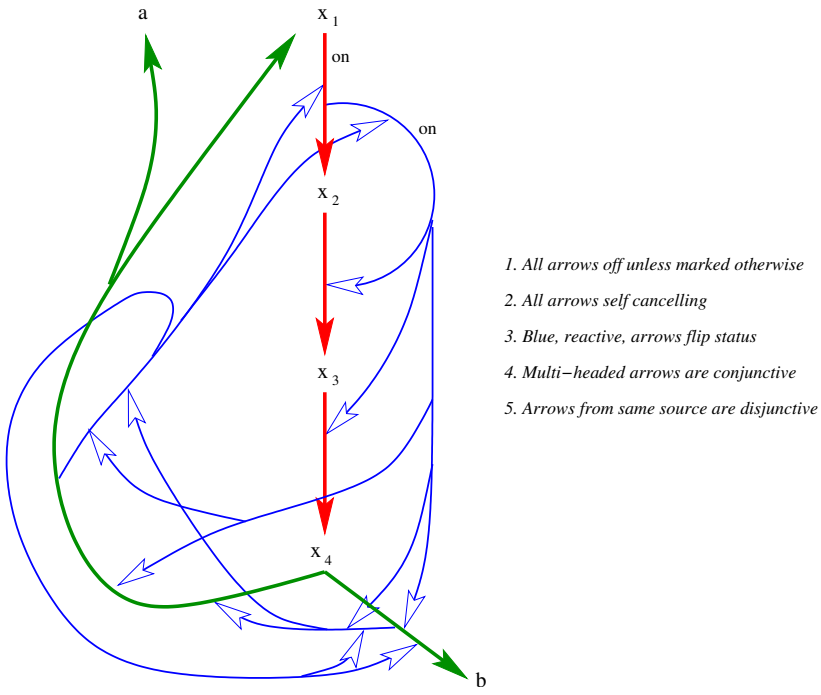

Figure 9: Cycle and choice

For ease of initial presentation, we label each grammar and reactivity rule for the reactive linear (sub)grammar depicted in Figure 9.

Grammar rules	Reactivities
$g_1 : x_1 \to x_2$	$r_1 : (g_1, \{g_2, g_3, g_4, r_4, g_5, r_5\})$
$g_2 : x_2 \to x_3$	$r_4 : (g_4, \{g_5, r_5\})$
$g_3 : x_3 \to x_4$	$r_5 : (g_5, \{g_4, r_4, g_1, r_1\})$
$g_4 : x_4 \to b$	
$g_5 : x_4 \to a x_1$	

A rule can be named by its source and target, e.g. g_1 by (x_1, x_2), hence, the above can be represented

by labelling just one rule, as below.

Grammar rules

$$\begin{aligned} x_1 &\to x_2 \\ x_2 &\to x_3 \\ x_3 &\to x_4 \\ x_4 &\to b \\ x_4 &\to a x_1 \end{aligned}$$

Reactivities

$$\begin{aligned} &((x_1, x_2), \{\ (x_2, x_3), (x_3, x_4), \\ &\qquad (x_4, b), ((x_4, b), \{(x_4, a x_1), r_5\}), \\ &\qquad (x_4, a x_1), r_5\}) \\ &r_5 : ((x_4, a x_1), \{\ (x_4, b), ((x_4, b), \{(x_4, a x_1), r_5\}), \\ &\qquad\qquad (x_1, x_2), ((x_1, x_2), \{\ (x_2, x_3), (x_3, x_4), \\ &\qquad\qquad\qquad (x_4, b), ((x_4, b), \{(x_4, a x_1), r_5\}), \\ &\qquad\qquad\qquad (x_4, a x_1), r_5\})\}) \end{aligned}$$

Whilst the above example encodings indicate how a linear grammar with k^n non-terminals can be represented by a reactive linear grammar with kn non-terminals, it is clear there may be a very large number of reactivity rules. Furthermore, the reactivities will be mutually recursive as soon as there are non-terminal cycles in the original grammar, as illustrated in the above example.

4.1 Is there a reduction for non-linear context-free grammars?

It is natural to question whether a similar reduction result can be obtained for non-linear context-free grammars. So here we briefly explore what happens when the above scheme is applied to context-free production rules. Consider a rule

$$X \to YZ$$

for non-terminals X, Y and Z. Let the vector of non-terminals $(x_1, x_2, \ldots, x_n)$ from the simulating grammar encode X, and then similarly for Y and Z. The context-free rule would then be translated into a set of grammar rules such as

$$\begin{array}{llll} x_1 \to x_2 & x_n \to y_1 z_2 & y_1 \to y_2 & z_1 \to z_2 \\ x_2 \to x_3 & & y_2 \to y_3 & z_2 \to z_3 \\ \vdots & & \vdots & \vdots \\ x_{n-1} \to x_n & & y_{n-1} \to y_n & z_{n-1} \to z_n \end{array}$$

together with reactivities

$$\begin{aligned} &((x_1, x_2), \{(x_2, x_3), \ldots (x_{n-1}, x_n), (x_n, y_1 z_1), ((x_n, y_1 z_1), \{\ldots\})\}) \\ &((x_n, y_1 z_1), \{\ldots\}) \\ &((y_1, y_2), \{(y_2, y_3), \ldots (y_{n-1}, y_n), \ldots\}) \\ &((z_1, z_2), \{(z_2, z_3), \ldots (z_{n-1}, z_n), \ldots\}) \end{aligned}$$

Superficially this seems adequate. Unfortunately, there are problems. Firstly it is important to remember that, for any given i, the non-terminals denoted by x_i, y_i and z_i are not necessarily disjoint (of course, those from different levels in the encoding chain are necessarily disjoint). Since the non-terminals Y and Z are jointly active, there may be conflict between their respective encoding chains. For example, if $y_1 = z_1 = a_1^1$ and $y_2 = z_2 = a_3^2$ then the application of the rule $y_1 \to y_2$ to the derived word $y_1 z_1$ will deactivate itself (all rules are self cancelling) with the consequence that the rule $z_1 \to z_2$ is deactivated. Thus the Z non-terminal will not get "decoded" and applied.

The above reduction result works for linear grammars because there is only one non-terminal present (i.e. active) in a derived word. This therefore suggests that perhaps the context-free case can be made to work by encoding, for example, a leftmost derivation strategy in which only the leftmost non-terminal in a derived word is allowed to be active. This would require a mechanism to ensure that the decoding rules and reactivities associated with non-leftmost non-terminals are disabled until the leftmost non-terminal

has been fully expanded. Consider again the rule $X \to YZ$ in the original grammar. For the situation where the sets of non-terminals derived from Y and Z are always disjoint, then this is possible. However, in general, there may be dependencies between the Y and Z non-terminals, as in Figure 10 which then require a stack in order to determine when a derivation from a non-terminal has been completed.

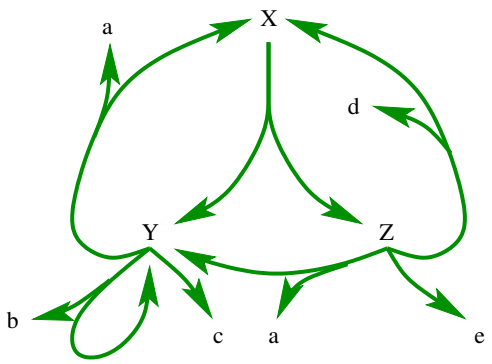

Figure 10: Non-terminal Graph

5 String transformer grammars

We now consider a quite different example of reactivity in grammars in which each string produced by the application of a rule may undergo a transformation before further rules may be applied. Such transformations may be thought of as 'mutations' of a string in which case their occurrence is spontaneous and unpredictable, or as a form of 'corruption' caused by, say, noise or other forms of degradation, in which case transformations of each string may be 'small' but unavoidable.

In fact, we consider a general formulation of such grammars which covers various interpretations of the string transformations. To do so we introduce a set of string transformers and define *string transformer grammars* in terms of rules where each rule has an associated string transformer. After application of the rule, the string undergoes the associated transformation before application of further rules. Special cases are standard grammars where no transformation takes place (i.e. allowing only the identity transformation), the case when transformations are independent of the rules used, and the case where string transformations may apply optionally. We concentrate our attention on certain forms of string transformations – those induced by actions on the non-terminal symbols in a string, which therefore interact directly with the application of rules.

Introducing this form of reactivity into grammars may considerably extend the expressivity of grammars. We begin to explore the classification of string transformer grammars. Moreover, this reactivity introduces an additional 'degree of freedom' into grammars and we consider how varying the string transformations affects the language generated.

Definition 4 *A string transformer grammar is a tuple consisting of a finite set of non-terminal symbols, N, a finite set of terminal symbols, Σ, a set of string transformers, $T \subseteq (N \cup \Sigma)^* \to (N \cup \Sigma)^*$, a set of production rules P with each rule of the form*

$$\alpha A \beta \xrightarrow{t} \gamma$$

for $A \in N$, $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, $t \in T$, and an initial non-terminal symbol $S \in N$.

Definition 5 *Let $G = \langle N, \Sigma, T, P, S \rangle$ be a string transformer grammar. G derives the string s from $u\alpha A\beta v$ in one step (written as $u\alpha A\beta v \Rightarrow s$) for $u, v, s \in (N \cup \Sigma)^*$ iff $\alpha A \beta \xrightarrow{t} \gamma$ and $t(u\gamma v) = s$. A word $w \in \Sigma^*$ is derived in G iff there is a sequence of single step derivations $s_0 \Rightarrow \ldots s_k \Rightarrow \ldots \Rightarrow s_n$ such that $s_k \in (N \cup \Sigma)^*$ and $S = s_0$ and $w = s_n$. The language generated by G is the set of its derivable words w.*

Note that the strings resulting from the application of string transformers need not be (and will not be, in general) strings generated from the underlying grammar. The definition is very general in terms

of the string transformers that are considered, allowing any function from the set of strings of terminals and non-terminal symbols to itself. As a consequence, if we wish to consider ϵ-freeness in this context, we need a modified definition. Restricted forms of string transformations are of particular interest. One such restriction is to limit string transformations to those that are induced by functions $f : N \to N$ from non-terminal symbols to non-terminal symbols. Such a function induces a string transformation by replacing each non-terminal symbol X in a string with the non-terminal symbol $f(X)$. Notice that interchanging non-terminal symbols in this way interacts with the application of production rules. It is in this interaction that we find new forms of expressivity of grammars. Often the form of the functions on non-terminal symbols that we consider may be succinctly expressed in terms of labelled symbols, that is the non-terminal symbols are of the form $\langle X, l\rangle$ with n a symbol and l a label from a label set Λ. A label transformer $f : \Lambda \to \Lambda$ defines a function f^* on these compound non-terminal symbols in an obvious way $f^*\langle X, l\rangle = \langle X, f(l)\rangle$.

Let us look at a few examples of string transformer grammars to show how words are generated and illustrate the power of these grammars.

First consider a right regular grammar written with labelled non-terminal symbols:

Example 11 (Odds and Evens) Consider the following string transformer grammar with labelled non-terminal symbols

$$G = \langle \{S\} \times \{even, odd\}, \{a\}, \{t_1, t_2, t_3\}, P, \langle S, even\rangle\rangle$$

where the set of production rules P is

$$\begin{aligned}
\langle S, even\rangle &\xrightarrow{t} a\langle S, odd\rangle \\
\langle S, odd\rangle &\xrightarrow{t} a \\
\langle S, odd\rangle &\xrightarrow{t} a\langle S, even\rangle
\end{aligned}$$

Notice that, for this example, all the rules have the same associated string transformer. When t is the identity function it is not difficult to see that G generates the language $\{a^{2n} \mid n > 0\}$. What do we get for non-identity transformers? Defining the transformers as functions on the label set, there are three other transformations to consider:

1. When $t = \{even \mapsto even, odd \mapsto even\}$, the first rule application yields the string $a\langle S, even\rangle$ since the generated label of S is *odd* which is then transformed under t to *even*. No finite word can therefore be generated and the language generated is thus empty.
2. $t = \{even \mapsto odd, odd \mapsto even\}$ also generates the empty language for the same reason.
3. $t = \{even \mapsto odd, odd \mapsto odd\}$ generates the language $\{a^{n+1} \mid n > 0\}$.

We may associate different string transformers with each production rule, as in the next example. Consider the above production rules, but with differing string transformers:

$$\begin{aligned}
\langle S, even\rangle &\xrightarrow{t_1} a\langle S, odd\rangle \\
\langle S, odd\rangle &\xrightarrow{t_2} a \\
\langle S, odd\rangle &\xrightarrow{t_3} a\langle S, even\rangle
\end{aligned}$$

where the string transformers are defined by the following functions on labels:

$$\begin{aligned}
t_1 :&\quad \{even \mapsto even, odd \mapsto even\} \\
t_2 :&\quad \{even \mapsto odd, odd \mapsto even\} \\
t_3 :&\quad \{even \mapsto odd, odd \mapsto odd\}.
\end{aligned}$$

The grammar generates the set of words $\{a^n \mid n0\}$. □

Notice that, in these examples, each right regular grammar generates a regular language. This is, in fact, a general result.

Theorem 6 *The class of languages generated by right regular string transformer grammars whose transformers are defined by functions on non-terminal symbols coincides with regular languages.*

Theorem 7 *Given two right regular grammars G_1 and G_2 that generate languages $\mathcal{L}_1$ and $\mathcal{L}_2$ respectively, there is a right regular string transformer grammar G with an identity string transformer which generates $\mathcal{L}_1$ and for which a different string transformer t makes G generate the language $\mathcal{L}_2$, moreover t is induced by a function on non-terminal symbols.*

Proof Without loss of generality we assume that the regular grammars G_1 and G_2 have disjoint non-terminal sets apart from their initial non-terminal symbol S. Let $G_i = \langle N_i, \Sigma_i, P_i, S\rangle$ for $i = 1, 2$. Construct a string transformer grammar $G_l = \langle N_1 \cup N_2, \{1, 2\}, \Sigma_1 \cup \Sigma_2, \{\iota\}, S_l, 1\rangle$ where the production rules in P are given by the set

$$\begin{array}{l}\{\langle S_l, 1\rangle \xrightarrow{\iota} \langle S, 1\rangle\} \cup \\ \{\langle X, 1\rangle \xrightarrow{\iota} \alpha[\langle Y, 1\rangle] \mid X \to \alpha(Y) \in P_1,\ X, Y \in N_1\} \cup \\ \{\langle X, 2\rangle \xrightarrow{\iota} \alpha[\langle Y, 2\rangle] \mid X \to \alpha(Y) \in P_2,\ X, Y \in N_2\}\end{array}$$

and ι is the identity string transformer. It is easy to see that G_l will generate the language of G_1.

Now replace the transformation ι by that defined by the relabelling $\{1 \mapsto 2, 2 \mapsto 2\}$. This revised grammar now generates the language of G_2. □

It is not the case that any standard grammar that generates a regular language also generates a regular language when extended with string transformers, even those defined by functions on non-terminal symbols.

We illustrate this with an example which also shows that a very simple grammar with simple string transformers can generate fairly complex languages. Indeed, our experience is that even in simple cases of string transformer grammars it is often very difficult to determine what the generated language is.

Example 12 (A simple grammar for a complex language) The context-free grammar

$$G = \langle \{S, A, B\}, \{a, b\}, P, S\rangle$$

where P is the set of production rules

$$\begin{array}{l}S \to AB \\ A \to aA \\ A \to a \\ B \to b\end{array}$$

generates the regular language $\{a^n b \mid n > 0\}$. A regular grammar for the same language is straightforward to define, for example:

$$\begin{array}{l}S \to aS \\ S \to b\end{array}$$

with S as the only non-terminal symbol and hence initial symbol. As an example of a string transformer grammar consider associating with each rule the string transformer induced by the function $\{A \mapsto B, B \mapsto A\}$ on non-terminal symbols.

Different derivation strategies give rise to different languages. The language generated by just leftmost derivations is $\{bb\}$ for we have only one derivation, namely:

$$S \Rightarrow BA \Rightarrow bB \Rightarrow bb$$

Note that $S \Rightarrow BA$ since $S \to AB$ and the transformation then yields the string BA. Similarly, $BA \Rightarrow bB$ as $B \to b$ and hence BA produces the bA which is then transformed to the string bB. The language generated by just rightmost derivations is $\{aa, aba, bab\}$, for we have

$$\begin{array}{l}S \Rightarrow BA \Rightarrow Aa \Rightarrow aa \\ S \Rightarrow BA \Rightarrow Aa \Rightarrow aBa \Rightarrow aba \\ S \Rightarrow BA \Rightarrow AaB \Rightarrow Bab \Rightarrow bab\end{array}$$

Note that both the above are regular languages. Consider now, however, the language generated through the unrestricted use of production rules. Here is an exhaustive list of derivations beginning with the initial symbol S.

$$\begin{array}{l}S \Rightarrow BA \Rightarrow bB \Rightarrow bb \\ S \Rightarrow BA \Rightarrow Aa \Rightarrow aa \\ S \Rightarrow BA \Rightarrow Aa \Rightarrow aBa \Rightarrow aba \\ S \Rightarrow BA \Rightarrow AaB \Rightarrow aaA \Rightarrow aaa \\ S \Rightarrow BA \Rightarrow AaB \Rightarrow aaA \Rightarrow aaaB \Rightarrow aaab \\ S \Rightarrow BA \Rightarrow AaB \Rightarrow aBaA \ldots \\ S \Rightarrow BA \Rightarrow AaB \Rightarrow Bab \Rightarrow bab\end{array}$$

All bar the sixth branch are complete derivations. The partial derivation given by the sixth branch has been stopped at a point at which it recurses. The above set is sufficient for us to obtain a description of the generated language in a closed form. Before doing so, let us continue the derivation from $aBaA$ for one more unfolding of the derivation tree to $aaBaaA$.

$$\begin{array}{l}
\ldots aBaA \Rightarrow abaB \Rightarrow abab \\
\ldots aBaA \Rightarrow aAaa \Rightarrow aaaa \\
\ldots aBaA \Rightarrow aAaa \Rightarrow aaBaa \Rightarrow aabaa \\
\ldots aBaA \Rightarrow aAaaB \Rightarrow aaaaA \Rightarrow aaaaa \\
\ldots aBaA \Rightarrow aAaaB \Rightarrow aaaaA \Rightarrow aaaaaB \Rightarrow aaaaab \\
\ldots aBaA \Rightarrow aAaaB \Rightarrow aaBaaA \ldots \\
\ldots aBaA \Rightarrow aAaaB \Rightarrow aBaab \Rightarrow abaab
\end{array}$$

If we examine the form of this recursion, we find that the language (of finite words) generated may be expressed in the following closed form:

$$\{a^n ba^n b \cup aaa^n \cup a^{n+1} ba^{n+1} \cup aaa(aa)^n b \cup a^n ba^{n+1} b \mid n \geq 0\}$$

The first term specifies the words derivable via the the first branch; the second term covers words derivable via the second and fourth branches; the third term covers those words derivable via the third branch; the fourth term for the fifth branch; and finally the fifth term covers those words derivable via the seventh branch. Note that the terms denote disjoint sets of finite words. The pumping lemma for regular languages can be used to show that the language is not regular. □

In Example 6, we saw that switching context-free grammars can define languages beyond context-free, using the example of $a^n b^n c^n$. We now revisit this language showing how it may be defined using a context-free string transformer grammar. In fact, there are many ways of generating this language using simple string transformer languages, each corresponding to a restricted derivation strategy. This suggest a general result which we present later and which relates derivation strategies to string transformer grammars.

Example 13 (The language $\{a^n b^n c^n \mid n \geq 1\}$ again) In this example, we show how the labelling and associated transformers can be used to control the derivation and therefore restrict the language associated with a grammar. The context-free grammar $G = \langle \{A, B, C, S\}, \{a, b, c\}, P, S \rangle$ where P is the set of production rules

$$\begin{array}{ll}
1: & S \to ABC \\
2: & A \to a \\
3: & A \to aA \\
4: & B \to b \\
5: & B \to bB \\
6: & C \to c \\
7: & C \to cC
\end{array}$$

generates the language $a^n b^m c^l$ for $n, m, l \in \mathbb{N}$. Let us now consider a labelled version of this in which functions on the labels are used to ensure that the production rules numbered 3, 5 and 7 are always applied in that sequential order and, similarly, the rules numbered 2, 4 and 6 are applied sequentially. These constraints ensure that the only words generated are those for which $n = m = l$ holds.

More formally, define the string transformer grammar

$$G = \langle \{A, B, C, S\} \times \{0, 1, 2, 3, 4\}, \{a, b, c\}, \{t_1, t_2, t_3\}, P, S, 0 \rangle$$

where P is the set of production rules

$$\begin{array}{ll}
1: & \langle S, 0 \rangle \xrightarrow{t_1} \langle A, 1 \rangle \langle B, 0 \rangle \langle C, 2 \rangle \\
2: & \langle A, 2 \rangle \xrightarrow{t_2} a \\
3: & \langle A, 2 \rangle \xrightarrow{t_1} a \langle A, 2 \rangle \\
4: & \langle B, 3 \rangle \xrightarrow{t_3} b \\
5: & \langle B, 2 \rangle \xrightarrow{t_1} b \langle B, 2 \rangle \\
6: & \langle C, 4 \rangle \xrightarrow{t_3} c \\
7: & \langle C, 2 \rangle \xrightarrow{t_1} c \langle C, 2 \rangle
\end{array}$$

and the functions on non-terminal symbols are defined the following label transformers:

$$\begin{array}{l} t_1 \text{ is } +1 \text{ modulo } 3 \\ t_2 \text{ is the constant function } 3 \\ t_3 \text{ is } +1 \end{array}$$

This grammar has derivations

$$\begin{array}{ll} \langle S,0\rangle & \Rightarrow \langle A,2\rangle\langle B,1\rangle\langle C,0\rangle \\ & \Rightarrow a\langle A,0\rangle\langle B,2\rangle\langle C,1\rangle \Rightarrow a\langle A,1\rangle b\langle B,0\rangle\langle C,2\rangle \Rightarrow a\langle A,2\rangle b\langle B,1\rangle c\langle C,0\rangle \\ & \vdots \\ & \Rightarrow a^{n-1}\langle A,2\rangle b^{n-1}\langle B,1\rangle c^{n-1}\langle C,0\rangle \\ & \Rightarrow a^n b^{n-1}\langle B,3\rangle c^{n-1}\langle C,3\rangle \Rightarrow a^n b^n c^{n-1}\langle C,4\rangle \Rightarrow a^n b^n c^n \end{array}$$

and thus generates the language $\{a^n b^n c^n \mid n \geq 1\}$. □

As a final example, illustrating again the complexity of derivation in string transformer grammars even when the rules are simple and so are the string transformers, consider a variant of Example 12.

Example 14 (Another simple grammar for a complex language) Consider the grammar $G = \langle\{S, A, B\}, \{a, b\}, P, S\rangle$ where the production rule set P is

$$\begin{array}{l} S \to AB \\ A \to aA \\ A \to a \\ B \to bB \\ B \to b \end{array}$$

This generates the regular language $\{a^n b^m \mid n, m > 0\}$. Now associate with each rule the same string transformer defined by the function interchanging non-terminal symbol symbols $\{A \mapsto B, B \mapsto A\}$.

What language is now generated?

For leftmost derivations, we obtain the following prefixes of derivation paths.

$$\begin{array}{l} S \Rightarrow BA \Rightarrow bB\ldots \\ S \Rightarrow BA \Rightarrow bAB \Rightarrow baA\ldots \\ S \Rightarrow BA \Rightarrow bAB \Rightarrow baBA\ldots \end{array}$$

Given that B generates the language $(ba)^*b \cup (ba)^+$ and A generates the language $(ab)^*a \cup (ab)^+$, the leftmost derivation language will be

$$(ba)^*(b((ba)^*b \cup (ba)^+) \cup ba((ab)^*a \cup (ab)^+)).$$

For rightmost derivations, we obtain the prefixes

$$\begin{array}{l} S \Rightarrow BA \Rightarrow Aa\ldots \\ S \Rightarrow BA \Rightarrow AaB \Rightarrow Bab\ldots \\ S \Rightarrow BA \Rightarrow AaB \Rightarrow BabA\ldots \end{array}$$

Because of the third sequence, it is slightly trickier to derive the closed form. We have the recursion

$$\begin{array}{l} Z(\mu) \Rightarrow A\mu a \\ Z(\mu) \Rightarrow B\mu ab \\ Z(\mu) \Rightarrow Z(\mu ab) \end{array}$$

where $S \Rightarrow Z(\epsilon)$. The general form for μ is therefore $(ab)^*$ and hence the rightmost derivations generate the language

$$((ab)^*a \cup (ab)^+)(ab)^*a \cup ((ba)^*b \cup (ba)^+)(ab)^+$$

or more simply

$$(ab)^*a(ab)^*a \cup (ab)^+a \cup (ba)^+b \cup (ba)^+(ab)^+.$$

Now consider the case of unrestricted rule application. Let us again give some of the prefixes of derivation paths.

$$
\begin{array}{ll}
1 & S \Rightarrow BA \Rightarrow bB \ldots \\
2 & S \Rightarrow BA \Rightarrow bAB \ldots \\
3 & S \Rightarrow BA \Rightarrow AaB \ldots \\
4 & S \Rightarrow BA \Rightarrow Aa \ldots
\end{array}
$$

Branches 1 and 4 are easy to close off. The first will yield words $b(ba)^*b \cup b(ba)^+$. Similarly, the fourth generates finite words $(ab)^*aa \cup (ab)^+a$.

What happens with the second and third branches? Let us expand the second branch.

$$
\begin{array}{ll}
2.1 & bAB \Rightarrow baA \ldots \\
2.2 & bAB \Rightarrow baBA \ldots \\
2.3 & bAB \Rightarrow bBbA \ldots \\
2.4 & bAB \Rightarrow bBb \ldots
\end{array}
$$

Again, we know what will happen with the first and fourth branches — as we had above apart from being prefixed by the letter b. The branch (2.2) now recurses having generated prefix ba. Thus it will generate a set of finite words that are prefixed by $(ba)^*$. But what follows the prefix? More interestingly, consider branch 2.3. There is recursion again, however, the letter b is both prefixed and placed between the non-terminal symbol B and A — if one iterated just using this branch, then words of the form b^nBb^nA are derived. Let us expand the third branch (3) of the derivation.

$$
\begin{array}{ll}
3.1 & AaB \Rightarrow aaA \ldots \\
3.2 & AaB \Rightarrow aBaA \ldots \\
3.3 & AaB \Rightarrow BabA \ldots \\
3.4 & AaB \Rightarrow Bab \ldots
\end{array}
$$

The first and fourth branches are straightforward. The second and third branches maintain the presence of non-terminal symbols A and B but switched in order, as in the case above.

It is instructive to see what happens with this string transformer grammar if we re-present it as a standard grammar using parameterised production rules. The new grammar will be directly constructed from the derivation tree, the beginnings of which we've written out above.

$$
\begin{array}{l}
S \to Z(\epsilon) \\
\\
Z(\mu) \to b\mu B \\
Z(\mu) \to ba\mu A \\
Z(\mu) \to baZ(\mu) \\
Z(\mu) \to bZ(\mu b) \\
Z(\mu) \to bB\mu b \\
Z(\mu) \to a\mu aA \\
Z(\mu) \to aZ(\mu a) \\
Z(\mu) \to Z(\mu ab) \\
Z(\mu) \to B\mu ab \\
Z(\mu) \to A\mu a \\
\\
B \to b \\
B \to bA \\
A \to a \\
A \to aB
\end{array}
$$

The non-terminal symbol Z is parameterised by a terminal string μ; $Z(\mu)$ represents derived words of the form $B\mu A$. The initial production S fires Z with an empty string, i.e. denoting the non-terminal pairing BA, the result of the first production application (after transformation) in the original grammar.

First of all, note the non-terminal symbol B yields the set of words $(ba)^*b \cup (ba)^+$. Similarly, A yields the set of words $(ab)^+ \cup (ab)^*a$. Thus, we can easily compute the words generated from the non-recursive rules for $Z(\mu)$. The other recursive cases are more interesting! A general pattern for words after a number of iterations from $Z(\mu)$ is as follows

$$((ba)^*b^{n_p}a^{m_p})^pZ(\mu\ ((ab)^*b^{n_p}a^{m_p})^p)$$

Note that we have the same indices occurring in the prefix substring as in the μ extension. We can thus substitute in the words generated through no further recursion of Z, obtaining the set of words

$$\{((ba)^*b^{n_p}a^{m_p})^p\ b\ ((ab)^*b^{n_p}a^{m_p})^p\ ((ba)^*b \cup (ba)^+) \mid p, n_1, \ldots, n_p, m_1, \ldots, m_p \in \mathbb{N}\} \cup$$
$$\{((ba)^*b^{n_p}a^{m_p})^p\ ba\ ((ab)^*b^{n_p}a^{m_p})^p\ ((ab)^+ \cup (ab)^*a) \mid p, n_1, \ldots, n_p, m_1, \ldots, m_p \in \mathbb{N}\} \cup$$
$$\{((ba)^*b^{n_p}a^{m_p})^p\ b\ ((ba)^*b \cup (ba)^+)\ ((ab)^*b^{n_p}a^{m_p})^p\ b \mid p, n_1, \ldots, n_p, m_1, \ldots, m_p \in \mathbb{N}\} \cup$$
$$\{((ba)^*b^{n_p}a^{m_p})^p\ a\ ((ab)^*b^{n_p}a^{m_p})^p\ a\ ((ab)^+ \cup (ab)^*a) \mid p, n_1, \ldots, n_p, m_1, \ldots, m_p \in \mathbb{N}\} \cup$$
$$\{((ba)^*b^{n_p}a^{m_p})^p\ ((ba)^*b \cup (ba)^+)\ ((ab)^*b^{n_p}a^{m_p})^p\ ab \mid p, n_1, \ldots, n_p, m_1, \ldots, m_p \in \mathbb{N}\} \cup$$
$$\{((ba)^*b^{n_p}a^{m_p})^p\ ((ab)^+ \cup (ab)^*a)\ ((ab)^*b^{n_p}a^{m_p})^p\ a \mid p, n_1, \ldots, n_p, m_1, \ldots, m_p \in \mathbb{N}\}$$

Thus we see that a simple context-free grammar with a simple string transformer has given rise to a very complex language. □

We have already shown that context-free string transformer grammars may generate non-context-free languages. Thus we are led to ask: what is the expressivity of context-free string transformer grammars?

Theorem 8 *A deterministic Turing machine (TM) can be simulated by a context-free string transformer grammar.*

Proof The essence of the simulation is to represent the tape of the TM by the currently derived word and the finite control of the TM is represented by the context-free grammar. Tape head movements are simulated by the string transformers. The non-terminal symbols of the regular grammar are triples formed from the state of the TM with the content of a tape cell and a marker 0 or 1 denoting whether the cell is under the head. The string transformers representing tape head movement may change the state component of a non-terminal symbol that is to come under the tape head following a left or right move.

The form of the string transformers here is, in fact, replacement of non-terminal symbols with non-terminal symbols but is not defined in terms of a function on the set of non-terminal symbols, as the determination of which symbol will replace a given one depends on its context in the string. This is the mechanism by which tape head movement is encoded.

In more detail, given a Turing Machine $TM = \langle Q, \Sigma, V, R, q_0, F\rangle$ where Q are the states of the finite control, Σ is the input alphabet, V is an alphabet, $R \subseteq (Q \times ((\Sigma \cup V) \times (\Sigma \cup V) \times \{L, R\}) \times Q)$ is the transition relation, and $F \in Q$ is the distinguished halt state, and q_0 is an starting state of the finite control. We assume an initial finitely populated tape $\sigma = \#_b\sigma_1 \ldots \sigma_n\#_e$ where $\#_b$ and $\#_e$ are beginning and end markers on the tape and $\sigma_i \in \Sigma$. We further assume that the tape head is positioned under σ_n.

We construct a context-free string transformer grammar with labelled non-terminal symbols

$$G = \langle N_{(\Sigma \cup V \cup \{\#_b, \#_e\}) \cup \{S\}, Q \times \{0,1\}}, \Sigma, T(\{L, R\} \times Q), P, S\rangle$$

where $N(\Sigma \cup V \cup \{\#_b, \#_e\})$ is a set of non-terminal symbols corresponding to the tape symbols of the Turing machine, i.e. the set $\{N_x \mid x \in (\Sigma \cup V \cup \{\#_b, \#_e\})\}$; S is a unique initial non-terminal symbol; the non-terminal labelling is drawn from $Q \times \{0, 1\}$, i.e. the states of the Turing machine paired with a 0 or 1 marker to denote whether the tape symbol corresponding to the non-terminal symbol is under the tape head; $T_{(\{L,R\} \times Q)}$ is a set of string transformers defined in terms of functions on labels, each one corresponding to either a left or right tape head movement into state $q \in Q$. Finally P is a set of context-free production rules related to the transitions of the Turing machine and is defined as follows.

1. The initial production for S generates the initial tape, i.e.

   $$S \to \langle \sigma_1, q_0, 0\rangle \ldots \langle \sigma_n, q_0, 1\rangle$$

   For ease of notation, in the labelled non-terminal symbol, we let σ_1 stand for non-terminal symbol N_{σ_1}.

2. For each TM rule of the form $q \overset{x,y,L}{\to} q'$ for $q \neq F$ and $x \neq \#_b$, we have the production rule

   $$\langle x, q, 1\rangle \overset{t_{Lq'}}{\to} \langle y, q, 1\rangle$$

3. For each TM rule of the form $q \overset{x,y,R}{\rightarrow} q'$ for $q \neq F$ and $x \neq \#_e$, we have the production rule

$$\langle x, q, 1\rangle \overset{t_{Rq'}}{\rightarrow} \langle y, q, 1\rangle$$

4. For each TM rule of the form $q \overset{\#_b,y,L}{\rightarrow} q'$ for $q \neq F$, we have the production rule

$$\langle \#_b, q, 1\rangle \overset{t_{Lq'}}{\rightarrow} \langle \#_b, q, 0\rangle\langle y, q, 1\rangle$$

5. For each TM rule of the form $q \overset{\#_e,y,R}{\rightarrow} q'$ for $q \neq F$, we have the production rule

$$\langle \#_e, q, 1\rangle \overset{t_{Rq'}}{\rightarrow} \langle y, q, 1\rangle\langle \#_e, q, 0\rangle$$

6. Three termination rules that rewrite the non-terminal symbols to their corresponding terminal letter (tape symbol).

$$\begin{array}{ll} \langle x, F, p\rangle \rightarrow x & \text{for } x \neq \#_b, \#_e \\ \langle \#_b, F, p\rangle \rightarrow \epsilon & \\ \langle \#_e, F, p\rangle \rightarrow \epsilon & \end{array}$$

The label sequence transformation functions are defined as below.

$$\begin{array}{ll} t_{Lq} : (Q \times \{0,1\})^* \rightarrow (Q \times \{0,1\})^* & t_{Rq} : (Q \times \{0,1\})^* \rightarrow (Q \times \{0,1\})^* \\ t^1_{Lq} : (Q \times \{0,1\})^+ \rightarrow (Q \times \{0,1\})^+ & t^1_{Rq} : (Q \times \{0,1\})^+ \rightarrow (Q \times \{0,1\})^+ \\ & \\ t_{Lq}(\langle\rangle) = \langle\rangle & t_{Rq}(\langle\rangle) = \langle\rangle \\ t_{Lq}(\sigma :: \langle s, 0\rangle) = t_{Lq}(\sigma) :: \langle q, 0\rangle & t_{Rq}(\langle s, 0\rangle :: \sigma) = \langle q, 0\rangle :: t_{Rq}(\sigma) \\ t_{Lq}(\sigma :: \langle s, 1\rangle) = t^1_{Lq}(\sigma) :: \langle q, 0\rangle & t_{Rq}(\langle s, 1\rangle :: \sigma) = \langle q, 0\rangle :: t^1_{Rq}(\sigma) \\ t^1_{Lq}(\langle\langle s, 0\rangle\rangle) = \langle\langle q, 1\rangle\rangle & t^1_{Rq}(\langle\langle s, 0\rangle\rangle) = \langle\langle q, 1\rangle\rangle \\ t^1_{Lq}(\sigma :: \langle s, 0\rangle) = t_{Lq}(\sigma) :: \langle q, 1\rangle & t^1_{Rq}(\langle s, 0\rangle :: \sigma) = \langle q, 1\rangle :: t_{Rq}(\sigma) \end{array}$$

We now claim that the string transformer grammar generates a word w if and only the Turing machine halts with tape contents $\#_b w \#_e$. The proof can be given by showing a bisimulation between the Turing machine and the string transformer grammar. We leave the details as an exercise for the interested reader. □

6 Reactivity and Derivation Strategies

We note from some of the previous examples that we may use the labelling and associated transformations of string transformer grammars to mimic derivation strategies. Can all derivation strategies be encoded this way? We answer this positively by first defining a suitable general notion of derivation strategy, then encoding through labels and relabelling functions a general scheme for constructing string transformer grammars.

Given a context-free grammar, let us uniquely name each of its production rules by, say, R_i, for $i \in \mathbb{N}$. For a given derivation sequence

$$S \Rightarrow \alpha_1 \Rightarrow \ldots \alpha_i \Rightarrow \ldots s$$

each derivation step can be labelled by the rule name applied at that step and the non-terminal position (amongst the non-terminal symbols of the currently derived string) at which the application took place. For example, given $G = \langle\{S, A, B\}, \{a, b\}, P, S\rangle$ with P the set of named production rules:

$$\begin{array}{ll} R_0 : & S \rightarrow AB \\ R_1 : & A \rightarrow a \\ R_2 : & A \rightarrow aA \\ R_3 : & B \rightarrow b \\ R_4 : & B \rightarrow bB \end{array}$$

This generates $\{a^m b^n \mid m, n \geq 1\}$. Here is an example of a (labelled) derivation sequence:

$$S \overset{R_0,1}{\Rightarrow} AB \overset{R_1,1}{\Rightarrow} aB \overset{R_3,1}{\Rightarrow} ab.$$

A leftmost derivation sequence is clearly a derivation sequence for which the non-terminal position is always 1, as above. Indeed, we can define a leftmost derivation strategy as one which only allows derivation sequences where the non-terminal position is always 1.

Of course, there are many other derivation strategies of interest. To move towards the more general situation let us consider a strategy for generating words of the form $a^{2n}b^n$ for $n \geq 1$ using the above production rules. An example of such a strategy is alternating the generation of 2 a's with 1 b, and so on. We can write this strategy explicitly in terms of derivation sequences, using only the last two rules used (at most). We can write this strategy as a function σ which takes a current derivation sequence and yields the set of rules which may be used next together with the position of the occurrence in the current string of the non-terminal symbol at which the rule is used. In the following definition of the function σ, let δ be an arbitrary derivation sequence, a dot denotes the concatenation of an item onto a derivation sequence, and ϵ is the empty sequence.

$$\begin{array}{ll}
\sigma(\epsilon) = \{(R_0, 1)\} & \\
\sigma((R_0, 1)) = \{(R_2, 2)\} & \\
\sigma(\delta.(R_2, 1).(R_2, 1)) = \{(R_4, 2)\} & \\
\sigma(\delta.(R, j).(R_2, 1)) = \{(R_1, 1), (R_2, 1)\} & \text{for } R \neq R_2 \\
\sigma(\delta.(R_1, 1)) = \{(R_3, 2)\} & \\
\sigma(\delta.(R_4, 2)) = \{(R_2, 1)\} &
\end{array}$$

For all other sequences the result is the empty set.

We are now in a position to introduce the general notion of a *derivation strategy* in a derivation tree of a context-free grammar.

Definition 6 *Consider a context-free grammar G with uniquely named production rules (with Rule the set of rule names in G). A* derivation strategy *(or simply a* strategy*) of G is a function*

$$\sigma : (Rule \times \mathbb{N}_1)^* \to \mathcal{P}_f(Rule \times \mathbb{N}_1).$$

Here $\mathcal{P}_f$ is the finite powerset function and $\mathbb{N}_1$ the positive integers. The argument to the strategy σ is a derivation sequence — the history of the derivation so far – recording the (possibly empty) sequence of named rules and the non-terminal positions in the strings at which the rule is applied. Given a derivation sequence, a strategy yields a (possibly empty) finite set of pairs (R, i) consisting of a rule name R and a non-terminal position i in the current string. For such a pair (R, i), a possible next step in the derivation is to apply the rule named R to the non-terminal at position i.

A derivation sequence δ is said to be a derivation in strategy σ if, at any position j in δ, $(0 \leq j \leq n)$, we have $\delta_j \in \sigma(\delta|_{j-1})$, where δ_j is the j-th step in δ and $\delta|_{j-1}$ is the initial segment of δ up to the $(j-1)$-th entry.

A strategy σ of grammar G is said to generate a word w if there is a derivation of w in σ. The language generated by a strategy is the set of all its generated words.

As in games, strategies may depend on only certain aspects of the history of a derivation (e.g. on the number of rule applications, as in the examples above) or, in fact, may be *history-free*, in which case they do not depend on the history of a derivation at all but only on the current string. In these cases, the strategy, as a function, is constant over certain collections of inputs.

Example 15 () A *leftmost* derivation strategy σ for a context-free grammar G (with uniquely named rules) is history-free, and $(R, i) \in \sigma(s)$ for derived string s iff $i = 1$ and R is the name of a rule of G for rewriting the non-terminal at position 1 in s.

A *leftish* derivation strategy σ for a context-free grammar G (with uniquely named rules) is history-free, and $(R, i) \in \sigma(s)$ for derived string s iff for all j, $j < i$, there are no rules in G for rewriting the non-terminal at position j, and R is the name of a rule for rewriting the i-th non-terminal symbol. □

This notion of a derivation strategy is *general* in the following sense.

Theorem 9 *For a context-free grammar, any set Δ of (finite or infinite) derivation sequences defines a strategy σ such that $\delta \in \Delta$ iff δ is a derivation sequence in σ.*

We now turn to the main result of this section, showing that any strategy for deriving words in a context-free grammar corresponds exactly to a string transformer grammar with unrestricted use of production rules. This result both shows the expressivity of string transformer grammars and also provides a general construction of such grammars for various languages.

Theorem 10 *Let G be a context-free grammar with a finite set of uniquely named production rules and σ a derivation strategy of G. Then there is a context-free string transformer grammar (possibly with an infinite set of rules) whose unrestricted rule application generates the language of σ. In fact, we may construct the context-free string transformer grammar so as to exactly simulate the strategy σ (in the sense of derivation steps being in 1-1 correspondence).*

Proof The key to the proof is the construction of a context-free string transformer grammar G' using labelled non-terminal symbols, where the labels code the current history together with a non-terminal position. We then ensure that G' has a rule for a particular labelled non-terminal just when the strategy σ allows that rule to be applied. This, together with suitable label transformations, ensures that the derivations of G' correspond exactly to those of the strategy σ of grammar G.

Thus, the non-terminal symbols of G' consist of the non-terminal symbols of G labelled with a pair (δ, i) where δ is a derivation sequence of σ and i a position of a non-terminal in the string derived by δ. This may produce a finite or an infinite set of labelled non-terminal symbols depending whether there is a finite or infinite number of derivation sequences of σ.

For the rules of G', consider a derivation sequence δ in strategy σ. For each $(R, i) \in \sigma(\delta)$ with rule $X \to s$ in G named by R, we create the rule of G',

$$\langle X, (\delta, i)\rangle \xrightarrow{t} \tilde{s}$$

where $\tilde{s}$ consists of the string s with each non-terminal labelled by the derivation δ and its non-terminal position in s. The transformation t acts on labels by 'advancing the history' and adding the current position of the non-terminal, that is, for the non-terminal at position j in the current string,

$$t(\delta, \_) = (\delta.(R, i), j).$$

The initial symbol of G' is $\langle S, (\epsilon, 1)\rangle$ where S is the initial symbol of G and ϵ is the empty derivation sequence.

It is straightforward to verify that, for each word w, there is a bijection between derivation sequences in strategy σ of word w and unrestricted derivations in G' of word w. □

Whilst, in general, this construction of a string transformer grammar may yield a grammar with an infinite set of rules, we shall see that for many natural strategies for generating languages, not only is the resulting grammar finite, but it is a succinct and useful presentation of the language. In other cases, the infinite set of rules is presented as a finite set of rule schema (i.e. parameterised rules).

We give examples of both these cases, constructing string transformer grammars from strategies in context-free grammars.

Example 16 (An alternation strategy as a string transformer grammar) We revisit the strategy described above for the language $\{a^{2n}b^n \mid n \geq 1\}$. The strategy is one of strict alternation between generating pairs of a's and single b's. This strategy depends only on the last two (at most) rules used, and so we can label the non-terminal symbols in the constructed string transformer grammar with the necessary one or two previous rules:

$$\begin{aligned}
&\langle S, (\epsilon, 1)\rangle \xrightarrow{t_0} \langle A, (\epsilon, 1)\rangle\langle B, (\epsilon, 2)\rangle \\
&\langle A, ((R_0, 1), 1)\rangle \xrightarrow{t_1} a\langle A, ((R_0, 1), 1)\rangle \\
&\langle B, ((R_2, 1).(R_2, 1), 2)\rangle \xrightarrow{t_2} b\langle B, ((R_2, 1).(R_2, 1), 2)\rangle \\
&\langle A, ((R, j).(R_2, 1), 1)\rangle \xrightarrow{t_3} a \qquad R \neq R_2 \\
&\langle A, ((R, j).(R_2, 1), 1)\rangle \xrightarrow{t_4} a\langle A, ((R, j).(R_2, 1), 1)\rangle \qquad R \neq R_2 \\
&\langle B, ((R_1, 1), 2)\rangle \xrightarrow{t_5} b \\
&\langle A, ((R_4, 2), 1)\rangle \xrightarrow{t_6} a\langle A, ((R_4, 2), 1)\rangle
\end{aligned}$$

with label transformations:

$$
\begin{array}{l}
t_0(\epsilon, \_) = ((R_0, 1), j) \\
t_1((R_0, 1), \_) = ((R_0, 1).(R_2, 1), j) \\
t_2((R_2, 1).(R_2, 1), \_) = ((R_4, 2), j) \\
t_3((R, j)(R_2, 1), \_) = ((R_1, 1), j) \\
t_4((R, j)(R_2, 1), \_) = ((R_2, 1).(R_2, 1), j) \\
t_5((R_1, 1), \_) = ((R_3, 2), j) \\
t_6((R_4, 2), \_) = ((R_4, 2).(R_2, 1), j)
\end{array}
$$

with j =1 for non-terminal symbol A and $j = 2$ for B.

Simplifying the labels reduces this grammar to the set of rules:

$$
\begin{array}{l}
S \to \langle A, 0\rangle\langle B, 0\rangle \\
\langle A, 0\rangle \xrightarrow{t_0} a\langle A, 0\rangle \\
\langle B, 22\rangle \xrightarrow{t_1} b\langle B, 22\rangle \\
\langle A, 2\rangle \xrightarrow{t_2} a \\
\langle A, 2\rangle \xrightarrow{t_3} a\langle A, 2\rangle \\
\langle B, 1\rangle \xrightarrow{t_4} b \\
\langle A, 4\rangle \xrightarrow{t_5} a\langle A, 4\rangle
\end{array}
$$

with label transformations: $t_0(0) = 2$, $t_1(22) = 4$, $t_2(2) = 1$, $t_3(2) = 22$, $t_4(1) = 3$, $t_5(4) = 2$. □

What about other strategies for the same language $\{a^{2n}b^n \mid n \geq 1\}$ using the same grammar?

Example 17 (Another string transformer construction) As another strategy for the language $\{a^{2n}b^n \mid n \geq 1\}$, we could allow arbitrary interleaved generation of a's and b's but allow termination only when the correct number of a'a and b's are generated. This again provides a derivation strategy which, using the above construction, produces the following equivalent string transformer grammar.

$$
\begin{array}{lll}
\langle S, ([0,0,0,0,0,0], 1)\rangle \xrightarrow{t_0} \langle A, ([0,0,0,0,0,0], 1)\rangle\langle B, ([0,0,0,0,0,0], 1)\rangle & & \\
\langle A, ([1,0,m,0,n], 1)\rangle \xrightarrow{t_1} a\langle A, ([1,0,m,0,n], 1)\rangle & & m, n \geq 0 \\
\langle B, ([1,0,m,0,n], 2)\rangle \xrightarrow{t_2} b\langle B, ([1,0,m,0,n], 2)\rangle & & m, n \geq 0 \\
\langle A, ([1,0,2n-1,0,n-1], 1)\rangle \xrightarrow{t_3} a & & n \geq 1 \\
\langle B, ([1,0,2n-1,0,n-1], 2)\rangle \xrightarrow{t_4} b & & n \geq 1 \\
\langle B, ([1,1,m,0,n], 1)\rangle \xrightarrow{t_5} b & & \\
\langle A, ([1,0,m,1,n], 1)\rangle \xrightarrow{t_6} a & &
\end{array}
$$

Here the arrays encode information about the derivation so far: The $i-th$ entry records the number of application of rule R_i of the grammar at the beginning of this section. The transformations of labels extend the count according to which rule has just been used:

$$
\begin{array}{l}
t_0([0,0,0,0,0], \_) = ([1,0,0,0,0], j) \\
t_1([1,0,m,0,n], \_) = ([1,0,m+1,0,n], j) \\
t_2([1,0,m,0,n], \_) = ([1,0,m,0,n+1], j) \\
t_3([1,0,2n-1,0,n-1], \_) = ([1,1,2n-1,0,n-1], j) \\
t_4([1,0,2n-1,0,n-1], \_) = ([1,0,2n-1,1,n-1], j) \\
t_5([1,1,2n-1,0,n-1], \_) = ([1,1,2n-1,1,n-1], j) \\
t_6([1,0,2n-1,1,n-1], \_) = ([1,1,2n-1,1,n-1], j)
\end{array}
$$

with j =1 for non-terminal symbol A and $j = 2$ for B.

Note that this is a context-free string transformer grammar with a set of *parameterised* rules, i.e. a finite schema for an infinite set of rules, one for each valid m and n. □

We now turn to another example – the running example of a non-context-free language: $\{a^n b^n c^n \mid n \geq 1\}$

Example 18 ($a^n b^n c^n$ yet again) Consider the context-free grammar G with named rules:

$$\begin{aligned}
&R_0 : S \to AC \\
&R_1 : A \to aAb \\
&R_2 : A \to ab \\
&R_3 : C \to cC \\
&R_4 : C \to c
\end{aligned}$$

A strategy for generating the language $\{a^n b^n c^n \mid n \geq 1\}$ is to alternate use of rules R_1 and R_3. This derivation strategy σ of G is defined as follows. For any derivation sequence δ:

$$\begin{aligned}
&\sigma(\epsilon) = \{(R_0, 1)\} \\
&\sigma((R_0, 1)) = \{(R_1, 1), (R_2, 1)\} \\
&\sigma(\delta.(R_1, 1)) = \{(R_3, 2)\} \\
&\sigma(\delta.(R_2, 1)) = \{(R_4, 2)\} \\
&\sigma(\delta.(R_3, 2)) = \{(R_1, 1), (R_2, 1)\}
\end{aligned}$$

This strategy generates the language $\{a^n b^n c^n \mid n > 0\}$.

Notice that this strategy depends only on the previous rule applied, thus we may simplify the labelling of non-terminal symbols using the previously applied rule in place of the entire derivation history. Thus labels consist of a rule name and the non-terminal position of its application, together with the current position of the non-terminal symbol.

Following the construction in the proof above, we construct a string transformer grammar G' with labelled non-terminal symbols which mimics this strategy to generate the same language. The rules of G' are:

$$\begin{aligned}
&\langle S, (\epsilon, 1)\rangle \xrightarrow{t_0} \langle A, (\epsilon, 1)\rangle\langle C, (\epsilon, 1)\rangle \\
&\langle A, ((R_0, 1), 1)\rangle \xrightarrow{t_1} a\langle A, ((R_0, 1), 1)\rangle b \\
&\langle A, ((R_0, 1), 1)\rangle \xrightarrow{t_2} ab \\
&\langle C, ((R_1, 1), 2)\rangle \xrightarrow{t_3} c\langle C, ((R_1, 1), 1)\rangle \\
&\langle C, ((R_2, 1), 2)\rangle \xrightarrow{t_4} c \\
&\langle A, ((R_3, 2), 1)\rangle \xrightarrow{t_5} a\langle A, ((R_3, 2), 1)\rangle b \\
&\langle A, ((R_3, 2), 1)\rangle \xrightarrow{t_6} ab
\end{aligned}$$

The string transformations t_m act on the labels of non-terminal symbols as follows, for the j-th non-terminal symbol in a string:

$$\begin{aligned}
&t_0(\epsilon, _) = ((R_0, 1), j) \\
&t_1((R_0, 1), _) = ((R_1, 1), j) \\
&t_2((R_0, 1), _) = ((R_2, 1), j) \\
&t_3((R_1, 1), _) = ((R_3, 2), j) \\
&t_4((R_1, 1), _) = ((R_4, 2), j) \\
&t_5((R_3, 2), _) = ((R_1, 1), j) \\
&t_6((R_3, 2), _) = ((R_2, 1), j)
\end{aligned}$$

By observing the pattern of the non-terminal labels in the rules of this grammar, we may simplify it considerably, using rule numbers as labels:

$$\begin{aligned}
&S \to \langle A, 0\rangle\langle C, 0\rangle \\
&\langle A, 0\rangle \xrightarrow{t_1} a\langle A, 0\rangle b \\
&\langle A, 0\rangle \xrightarrow{t_2} ab \\
&\langle C, 1\rangle \xrightarrow{t_3} c\langle C, 1\rangle \\
&\langle C, 2\rangle \xrightarrow{t_4} c \\
&\langle A, 3\rangle \xrightarrow{t_5} a\langle A, 3\rangle b \\
&\langle A, 3\rangle \xrightarrow{t_6} ab
\end{aligned}$$

with transformations $t_1(0) = 1, t_2(0) = 2, t_3(1) = 3, t_4(1) = 4, t_5(3) = 1, t_6(3) = 2$. □

Example 19 (A context-sensitive grammar) In [16], Rosenkrantz uses the following context-sensitive language to demonstrate a programmed context-free grammar with both success and failure continuation rule sets:

$$\{\text{bin}(n)ha^n \mid n \in \mathbb{N}\}$$

where bin(n) is a binary representation (with, say, leftmost digit as the most significant) of the positive integer n. Let us first give an unrestricted grammar for this language. Take $G_u = \langle\{S, T, T', D, Z, F\}, \{1, 0, h, a\}, P, S\rangle$ with P the set of rules

$$\begin{array}{l} S \to TZ \\ T \to 1T'aD \\ T \to 0T'D \\ T' \to 1T'aD \\ T' \to 0T'D \\ T' \to hF \\ Da \to aaD \\ DZ \to Z \\ Fa \to aF \\ FZ \to \epsilon \end{array}$$

G_u thus has derivations such as:

$$\begin{array}{ll} S & \Rightarrow TZ \Rightarrow 1T'aDZ \Rightarrow 11T'aDaDZ \Rightarrow 11T'aaaDDZ \Rightarrow 11hFaaaDDZ \Rightarrow 11haFaaDDZ \\ & \Rightarrow 11haaFaDDZ \Rightarrow 11haaaFDDZ \Rightarrow 11haaaFDZ \Rightarrow 11haaaFZ \Rightarrow 11haaa \end{array}$$

Consider now the following context-free grammar with named rules $G_{cf} = \langle\{S, A\}, \{1, 0, h, a\}, \{r_0, r_1, r_2, r_3, r_4\}, P, S\rangle$ with rules in P as

$$\begin{array}{ll} r_0: & S \to 1SA \\ r_1: & S \to 0S \\ r_2: & S \to h \\ r_3: & A \to AA \\ r_4: & A \to a \end{array}$$

The language generated by this context-free grammar clearly contains the language of G_u. So by defining an appropriate strategy for G_{cf}, we can obtain the language of G_u. □

6.1 Switching grammars and string transformers

We now consider the relationship between switching grammars, introduced earlier in the paper, and string transformer grammars.

Theorem 11 *Every context-free switching grammar can be exactly simulated by a context-free string transformer grammar i.e. there is a context-free string transformer grammar whose derivation steps correspond exactly to those of the switching grammar. In particular, the class of languages generated by context-free switching grammars is contained in that of context-free string transformer grammars.*

Proof Let $G = \langle N, \Sigma, I, P, S, i_0\rangle$ be a context-free switching grammar. For each $i \in I$, we form a set of string transformer rules $\tilde{P}_i$ as follows: For each rule $X \to \gamma, j$ in P_i form the rule $\langle X, i\rangle \xrightarrow{t} \tilde{\gamma}$ in $\tilde{P}_i$ where $\tilde{\gamma}$ is γ with each non-terminal symbol Y replaced by $\langle Y, i\rangle$, and the string transformer t is defined by the function on non-terminal symbols $t(\langle Y, k\rangle) = \langle Y, j\rangle$.

Now form the set of production rules of a string transformer grammar $\tilde{P} = \bigcup_{i\in I} \tilde{P}_i$. It is easy to verify that this string transformer grammar exactly simulates the original switching grammar. □

We have shown that any derivation strategy may be represented (by an exact simulation) by a context-free string transformer grammar with unrestricted rule application. Context-free switching grammars have the same degree of expressivity for arbitrary derivation strategies but the encoding of strategies is different and the encoding in string transformer grammars does not factor through the construction for switching grammars. A difficulty in both cases is to identify the n-th non-terminal symbol (which is used in the definition of strategies) in a string after a substitution has taken place. With string transformers, we relabel the non-terminal symbols after a rule application, for switching grammars this

option is not available and we have to code, within the labels, information about relative positions of the non-terminal symbols. Switching grammars as representations of strategies tend to be less succinct than string transformer grammars, partly because of this more elaborate labelling of non-terminal symbols.

Theorem 12 *Let G be a context-free grammar with uniquely named production rules and σ a derivation strategy of G. Then there is a context-free switching grammar with unrestricted rule application which exactly simulates σ.*

Proof Consider a valid derivation δ of string s in strategy σ with non-empty $\sigma(\delta)$. We proceed by induction on the length of δ. Assume that there is a switching grammar with rule sets indexed by initial substrings of δ (not including δ) which exactly simulates the steps of δ to produce a string $\tilde{s}$. Form a set of rules P_δ as follows. For each $\langle l, n\rangle \in \sigma(\delta)$, let $X \to \gamma$ be the rule labelled l. Form switching rule $\langle X, \lambda\rangle \to \tilde{\gamma}, \delta'$ in P_δ where λ is the label of the n-th non-terminal symbol in $\tilde{s}$, $\tilde{\gamma}$ is γ with non-terminal symbol Y at the j position in γ replaced by $\langle Y, \lambda.j\rangle$ and $\delta' = \delta.\langle l, n\rangle$.

Then for all valid derivation sequences δ of σ, the indexed set of rule sets P_δ forms, together with initial symbol $\langle S, 0\rangle$ where S is initial symbol of G, and initial set of rules $\sigma([])$, forms a switching grammar that exactly simulates the strategy σ of G. □

Example 20 (The language $\{a^n b^n c^n \mid n \geq 1\}$) Consider the context-free grammar G with rules

$$\begin{array}{ll} 1: & S \to AX \\ 2: & X \to bXc \\ 3: & X \to bc \\ 4: & A \to AA \\ 5: & A \to a \end{array}$$

As a strategy σ for $a^n b^n c^n$, for $n \geq 1$, consider alternating between extending the A's and generating X's, then, when there are no X's, finally reducing A's to a's in an arbitrary order. Thus:

$\sigma([]) = \{\langle 1, 1\rangle\}$
$\sigma([\langle 1, 1\rangle]) = \{\langle 4, 1\rangle, \langle 5, 1\rangle\}$
$\sigma(\delta.\langle 4, 1\rangle) = \{\langle 5, |\delta| + 1\rangle\}$
$\sigma(\delta.\langle 5, i\rangle) = \{\langle 4, 1\rangle, \langle 3, i\rangle\}$
$\sigma(\delta.\langle 3, i\rangle) = \{\langle 5, j\rangle \mid 1 \leq j \leq |\delta|\}$
$\sigma(\delta.\langle 5, i\rangle) = \{\langle 5, j\rangle \mid 1 \leq j \leq k\}$
where k is the number of A symbols in the string generated by $\delta.\langle 5, i\rangle$

We consider the first few steps of the construction using the derivation sequence $\delta = S \Rightarrow AX \Rightarrow AAX$. The initial set is

$$P_{[]} = \{\langle S, 0\rangle \to \langle A, 0.1\rangle\langle X, 0.2\rangle, [\langle 1, 1\rangle]\}.$$

The corresponding step in the switching grammar for $S \Rightarrow AX$ is therefore $\langle S, 0\rangle \Rightarrow \langle A, 0.1\rangle\langle X, 0.2\rangle$. For the next step, the strategy gives two rules, $A \to AA$ or $A \to a$. Thus the ruleset for the switching grammar is

$$P_{[\langle 1,1\rangle]} = \{\langle A, 0.1\rangle \to \langle A, 0.1.1\rangle\langle A, 0.1.2\rangle, [\langle 1, 1\rangle, \langle 4, 1\rangle],\ \ \langle A, 0.1\rangle \to a, [\langle 1, 1\rangle, \langle 5, 1\rangle]\}.$$

Then the derivation δ, represented in the switching grammar, yields the string $\langle A, 0.1.1\rangle\langle A, 0.1.2\rangle\langle X, 0.2\rangle$. Notice how the labels are used to code the tree of substitution positions so that the correct rules may be applied according to the form of the labelled non-terminal symbol. □

7 Embedded reactivity

We now turn to examples of reactive grammars where the reactivity, instead of being associated with rules of the grammar, is determined by the generated string or parts of the string. The presentation and formalisation of such grammars is not straightforward because what is meant by 'parts of a string' depends, in general, on the parsing of the string, i.e. on the derivation sequence used to generate it. Moreover, the effect of reactivity will depend again on the derivation sequence and how subtrees in a parse tree are related to each other.

Let us begin by looking at some examples of embedded reactivity.

Example 21 (Correction grammars) Consider the following example text:

> '*Tom likes playing basketball, sorry I mean football.*'

Here, the phrase '*sorry I mean X*' is a reactive component of the text, that is, it describes a transformation to be performed on the text. In this case, the transformation is to replace the word '*basketball*' with '*football*' and remove the reactive component, resulting in the string:

> '*Tom likes playing football.*'

Let us look at the process in more detail. How is the '*X*' in '*sorry I mean X*' matched to part of the remaining text? Clearly, it ought to have the same 'phrase type' which in this case is a noun phrase (indeed it is a noun). But there are two noun phrases in the text '*Tom likes playing basketball.*' Which to choose? Both semantics and proximity may have a role in determining the action to be associated with a reactive component.

Reactive components may interact. Consider the example text:

> '*Jack loves Mary. Mary rejected Jack, sorry I mean the man in blue, sorry I mean red, sorry I mean accepted.*'

Of course, as a natural language text, this is rather contrived, but as an example of an embedded reactivity, it illustrates the complex phenomena that may be involved. One possible resolution of this (i.e. determination of the actions associated with the reactive components) is the text:

> '*Jack loves Mary. Mary accepted the man in red.*'

Other resolutions are possible, for example:

> '*The man in red loves Mary. Mary accepted the man in red.*'

When a string has multiple derivations, i.e. ambiguity in parsing, then the reactivity may depend upon the derivation. For example:

> '*British left waffles on Falkland Islands, sorry I mean toast!*'

□

We are not proposing embedded reactive grammars as a technique for analysing such natural language phrases as in the example above. Rather, we are using the example to illustrate embedded reactivity and its analysis.

Example 22 (Spelling reform) A widely quoted text, purporting to be an incremental spelling reform, is:

> For example, in Year 1 that useless letter 'c' would be dropped to be replased either by 'k' or 's,' and likewise 'x' would no longer be part of the alphabet. The only kase in which 'c' would be retained would be the 'ch' formation, which will be dealt with later. Year 2 might reform 'w' spelling, so that 'which' and 'one' would take the same konsonant, wile Year 3 might well abolish 'y' replasing it with 'i' and Iear 4 might fiks the 'g/j' anomali wonse and for all. Jenerally, then, the improvement would kontinue iear bai iear with Iear 5 doing awai with useless double konsonants, and Iears 6-12 or so modifaiing vowlz and the rimeining voist and unvoist konsonants. Bai Iear 15 or sou, it wud fainali bi posibl tu meik ius ov thi ridandant letez 'c,' 'y' and 'x"–bai now jast a memori in the maindz ov ould doderez–tu riplais 'ch,' 'sh,' and 'th' rispektivli. Fainali, xen, aafte sam 20 iers ov orxogrefkl riform, wi wud hev a lojikl, kohirnt speling in ius xrewawt xe Ingliy-spiking werld.

The reactivity here is the replacement of letters, or letter combinations. Notice how the action associated with a reactive component may depend upon the phonetics as well as on letters and letter combinations. Notice also that, unlike the previous example where the action was retrospective and we are given the phrase before the action, here it is applied to subsequent text and we are given the result of the application. Again, reactive components are acted on by other reactive components. □

Example 23 (Grammar and word order) Consider the example text:

Tom loves Mary. From now on, subjects switch with objects. Pizza likes Jack. From now on, verbs switch with objects. Cat Mary loves.

Here again there are reactive components, now acting on word order according to phrase classes occupied by words. Like the previous example, this is forward action. Notice also that the reactive component '*verbs switch with objects*' is subject to the component '*subjects switch with objects*' and so the actual action is '*objects switch with verbs*', which, in this case, is the same action but, in general, would be different. Again, this brings out the complexity of embedded reactivity, where the reactivity can act on other reactive components, or indeed allows the possibility of two reactive components acting on each other. □

7.1 Formalising grammars with embedded reactivity

We begin the process of formalising a general notion of an *embedded reactive grammar*. To do so, we first formalise parse trees in terms of derivations.

We introduce a labelling of non-terminals in a derivation sequence using lists of numbers. For example, consider a derivation step

$$aAcB \rightarrow aaCAcB$$

which is the result of replacing A using a rule

$$r : A \rightarrow aCA.$$

If the original non-terminal symbol A is labelled p then the C in the result string is labelled $p.1$ and the A in the resultant string is labelled $p.2$; the label of B remains unchanged. The initial symbol is labelled with the empty sequence. Each derivation step is denoted by a pair (p, r) consisting of a non-terminal label p (the label of the non-terminal to be replaced) and a rule name r (the rule used).

Definition 7 *For derivation steps $\pi = (p, r)$ and $\pi' = (p', r')$, we say that (p', r')* extends *(p, r) iff sequence p is a prefix of p'.*

For a derivation sequence, $\delta = \pi_0, \ldots, \pi_n$ a subsequence $\pi_{i_0}, \ldots \pi_{i_k}$ is compatible *iff (1) $\pi_{i_{j+1}}$ extends π_{i_j} for $0 \leq j \leq k-1$, and (2) no intermediate derivations extend any of the elements of the subsequence.*

In a derivation sequence δ of string s, the collection of compatible subsequences which are left closed (i.e. may not be extended leftwards) forms a tree (i.e. is prefix closed). This is the parse tree $\mathsf{parsetree}(\delta, s)$ *determined by the derivation sequence. A* path *p in a parse tree (as a list of numbers) determines a subtree $\mathsf{subtree}(\delta, s, p)$. The set of paths in a parse tree, we denote by $\mathsf{paths}(\delta, s)$.*

Definition 8 (Embedded reactive grammar) *An* embedded reactive grammar *consists of a set of non-terminal symbols N, an initial non-terminal $S \in N$, a set of terminal symbols Σ, a set of (production) rules P as in Definition 1, together with for each derivation sequence δ of string s a set of reactive components $R(\delta, s) \subseteq \mathsf{paths}(\delta, s)$. In addition, there is a transformation of strings T, which associates with each reactive component $p \in R(\delta, s)$ a transformed string $s' = T(p, \delta, s)$* together with *a derivation sequence δ' of s'.*

The definition is fairly general. It allows reactive components to contain non-terminals (i.e. allows some forms of 'reactive patterns'). The transformations associated with a reactive component are unrestricted – any string may be the result of such a transformation. Note that the result of a transformation is not simply a string but a derivation sequence of a string. Thus transformations associated with reactive components map derivation sequences to derivation sequences. However, the transformation is functional – a unique string and its derivation is determined by a reactive component in a derivation of a string. The recognition of reactive components is, in general, dependent upon the context, i.e. on the string in which it is embedded. For example, in the case of text correction, a phrase '*sorry I mean football*' may be considered not to be a reactive component if, for example, it appears in a direct quote.

In defining a grammar, it is usual to define what is meant by a derivation step. However, this is no longer a simple matter because embedded reactivities move us between derivation sequences. Several options are available which correspond to strategies for applying the actions associated with reactive components. One is to first generate a terminal string using the underlying grammar, then apply the actions of any reactive components (in any order, each time producing a new derivation sequence), to end possibly with a word with no reactive components (this will not always occur - it depends on

the transformations). Another possibility is to interleave applications of production rules with reactive transformations. In either case, we need a new term to denote the combined processes of applying production rules and reactive transformations.

Definition 9 *A string s' with derivation δ' is a* single step result *from string s with derivation δ, written $(\delta, s) \rightarrow (\delta', s')$ in an embedded reactive grammar iff either (1) this is a single step derivation in the underlying grammar, or (2) (δ', s') is the result of a transformation associated with a reactive component of (δ, s).*

Chains of such single steps arise from various strategies for applying grammar productions and reactive transformations.

Let us now consider how one of the examples above is an embedded reactive grammar in this sense.

Example 24 (Correction grammars revisited) We revisit the example of correction grammars above (Example 21) and show how it is an example of an embedded reactive grammar.

The reactive components are of the form $\rho =$ '*sorry I mean* w' where w is a terminal word generated from a non-terminal X. In the tree $t = \mathsf{parsetree}(\Delta, s)$, the action of a subtree determining the reactive component ρ is the replacement of a suitably chosen subtree of t and the removal of subtree ρ from t. The resultant tree must itself be a parse tree in the underlying grammar i.e. have an associated derivation Δ'. Which subtree is replaced is a choice determined by our understanding of the notion of text correction.

This is one form of an embedded reactive grammar that describes the correction process. Notice that we only replace with terminal words. This is not always necessary (depending upon how trees are matched in the correction phase) and an alternative formulation would allow reactive components to be of the form '*sorry I mean* s' where $s \in (N \cup \Sigma)^*$. □

Thus in the case of correction grammars, we present the string before any actions of reactive components are undertaken, and then consider the result of these actions. For the other two examples above, the text presented is the result of applying the actions. An analysis of the text requires us to find a chain of single steps (consisting of applying grammar rules and reactive transformations) which result in the given text. Notice also that, in these cases, the reactive transformation leaves in place the reactive component (i.e. the instructions for modifying the text) instead of deleting it.

8 Conclusions

Reaction to movement is a pretty natural concept. If one considers the dynamic operation of an automaton, or state machine, there is movement as the automaton transits from one state to another as a symbol is accepted (or generated). Thus it is natural to explore how automata, themselves, might be extended to react to such movement. Guided by prior work applying ideas of reactivity in automata and Kripke structures, we have made an initial exploration into how different forms of reactivity can be applied in the context of grammars. Given the considerable research that has been undertaken in the area of grammars over the past 40 years, it comes as little surprise that some of the reactivities we've considered coincide with various, well-known, extensions to grammar systems. Our interests, however, have been driven primarily by an investigation of principles, rather than finding ways to extend the expressiveness of particular forms of grammar rules. Our work in this area is preliminary and, although results are few, we believe that the range of reactivities explored is itself of considerable interest. One particular area that we have not explored in this paper and which we believe warrants further work is the use of reactive grammars in parsing, with the intention of generating efficient parsing algorithms for fairly complex languages which can be expressed using simple reactive grammars.

Acknowledgments

The authors would like to thank Yanliang Jiang for his Masters work on investigating and implementing some forms of embedded reactive grammars.

References

[1] A.V. Aho. Indexed Grammars — An Extension of Context-Free Grammars. *Journal of the ACM*, 15(4): 647–671 (1968).

[2] N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2: 137–167 (1959).

[3] H. Barringer, M. Fisher, D. Gabbay, R. Owens and M. Reynolds. *The Imperative Future: Principles of Executable Temporal Logic*. Research Studies Press. 1996.

[4] H. Barringer, A. Goldberg, K. Havelund and K. Sen. Rule-Based Runtime Verification. Proceedings of the *VMCAI'04, 5th International Conference on Verification, Model Checking and Abstract interpretation*, Venice. Volume 2937, Lecture Notes in Computer Science, Springer-Verlag, 2004. 2004.

[5] H. Barringer, K. Havelund and D. Rydeheard. Rule systems for run-time monitoring: from EAGLE to RULER. *Lecture Notes in Computer Science* 4839, Springer (2007) 111–125.

[6] J. Dassow and G. Paun. *Regulated Rewriting in Formal Language Theory*. EATCS Monographs on Theoretical Computer Science 18. Springer-Verlag. 1989.

[7] H. Fernau. Regulated grammars with leftmost derivations. Proceedings of *SOFSEM'98*, Lecture Notes in Computer Science, Volume 1521, 322–331, Springer-Verlag, 1998.

[8] D.M. Gabbay. Introducing Reactive Kripke Semantics and Arc Accessibility. *Lecture Notes in Computer Science* 4800: 292–341. Springer 2008.

[9] S. Ginsburg and E.H. Spanier. Control Sets on Grammars. *Mathematical Systems Theory*, 2(2): 159–177 (1968).

[10] O.H. Ibarra. Simple matrix languages. *Information and Control*, 17: 359–394 (1970).

[11] T. Kasai. An Hierarchy between Context-Free and Context-Sensitive Languages. *Journal of Computer and System Sciences*, 4: 492–508 (1970).

[12] A. Lindenmayer Mathematical models for cellular interaction in development, I and II. *Journal of Theoretical Biology*, 18: 280–315 (1968).

[13] R. Meersman and G. Rozenberg *Cooperating grammar systems*. Springer LNCS 64 (1978).

[14] E. Moriya. Some remarks on State Grammars and Matrix Grammars. *Information and Control*, 23: 48–57 (1973).

[15] G. Paun *The Generative Mechanisms of Some Economic Processes*. Techn. Publ. House. Bucharest (1980).

[16] D.J. Rosenkrantz. Programmed grammars and classes of formal languages. *Journal of the ACM*, 16(1): 107–131 (1969).