A USEFUL LEMMA FOR CONTEXT-FREE PROGRAMMED GRAMMARS

JAN VAN LEEUWEN

RUU-CS-78-1

January 1978

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6
Utrecht 2506
Telefoon 030-53 1454

A USEFUL LEMMA FOR CONTEXT-FREE PROGRAMMED GRAMMARS

JAN VAN LEEUWEN

Technical Report RUU-CS-78-1

January 1978

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht, the Netherlands

proposed running head:

CONTEXT-FREE PROGRAMMED GRAMMARS

all correspondence to:

Dr. Jan van Leeuwen
Dept. of Computer Science
University of Utrecht
Budapestlaan 6, P.O.Box 80.012
3508 TA Utrecht
the Netherlands

A USEFUL LEMMA FOR CONTEXT-FREE PROGRAMMED GRAMMARS

Jan van Leeuwen
Department of Computer Science
University of Utrecht
3508 TA Utrecht, the Netherlands

Abstract. We show that all quasi-realtime one-way multi-counter languages can be generated by a context-free ε-free programmed grammar (even under the free interpretation). The result can be used to obtain a new and almost trivial proof of the fundamental theorem that arbitrary context-free programmed grammars can generate all recursively enumerable languages. The proof of our result also yields the following, interesting characterization: the quasi-realtime one-way multi-counter languages are precisely the ε-limited homomorphic images of (free) context-free programmed production languages. It follows that the (free) derivation languages of context-free or even context-free programmed grammars, which were known to be context-sensitive, are in fact contained in the family of context-free ε-free programmed languages.

1. INTRODUCTION

In the past ten years numerous generalizations of the context-free grammar-formalism have been proposed (see e.g. Salomaa [10]), but only a few have survived. Context-free programmed grammars (Rosenkrantz [9]) have proved their value in formal language theory for some time, as the application of context-free productions according to some "program" appeared to be both natural and sufficient in many instances where the more powerful context-sensitive or phrase structure grammars were used before. The idea of using grammars as programs was developed further by van Leeuwen [11], who showed that even "weak" control-structures are sufficient to gain the full power of context-free programmed grammars.

Whereas context-free ε-free programmed languages are strictly contained in the context-sensitive languages (Rosenkrantz [9], p. 116) and in NP (van Leeuwen [12]), the fundamental theorem for programmed grammars asserts that arbitrary context-free programmed grammars can generate precisely all recursively enumerable languages. The theorem was originally shown by Rosenkrantz ([9], p. 119) for context-free programmed grammars operating under the leftmost mode of derivation, but later proofs of Salomaa ([10], thm 5.1) and van Leeuwen [11] removed

this condition. All proofs have in common that they are long and tedious, requiring the explicit programming of a fair number of "arithmetical subroutines". We shall attempt to simplify this considerably.

The main result of this paper is

Lemma A. The family of quasi-realtime one-way multi-counter languages is (strictly) included in the family of context-free ε-free programmed languages.

We note that in the lemma and from now on we shall always assume the free mode during derivations. Observe how the lemma, which is in itself essentially quite easy to prove, yields an almost trivial proof of the fundamental theorem for programmed grammars, which should be convincing even without knowing the precise definition of these grammars (see section 2).

Theorem. The family of (arbitrary) context-free programmed languages is precisely the family of all recursively enumerable languages.

Proof

It follows as usual that context-free programmed languages are re.

Let L be an arbitrary re language, $L \subseteq \Sigma^*$ for some alphabet Σ. It is wellknown that L can be accepted by a one-way multi-counter machine M (see Minsky [8], Fischer [2], or Hopcroft & Ullman [5]). Choose $¢ \notin \Sigma$. Make M into a realtime machine M' by having M read a "¢" each time it would do an ε-move. The language L' recognized by M' can be generated by some context-free ε-free programmed grammar G, according to Lemma A. Erasing the terminal ¢ in the righthand-side of all productions in G yields a context-free programmed grammar for L.

□

The proof of Lemma A will yield another interesting result, which can be used to obtain a complete "grammatical" characterization of all delay-bounded one-way multi-counter languages.

Lemma B. The family of quasi-realtime one-way multi-counter languages is precisely the family of ε-limited homomorphic images of production languages of context-free programmed grammars.

The Lemma and its generalization for arbitrary delay-bounded one-way multi-counter machines will be shown in section 3. One application of Lemma B immediately shows that derivation languages of context-free programmed grammars are (in fact, deterministically) log-space recognizable (see also Igarashi [6]). More interesting for us here is the following conclusion.

Theorem. The derivation languages of context-free programmed grammars are contained in the family of context-free ε-free programmed languages.

Proof

Combine Lemma A and Lemma B.

□

Earlier, the best result known for these derivation languages merely stated inclusion in the family of context-sensitive languages (as one can observe from Salomaa [19], note 6.2). One may rephrase the result as saying that "context-free ε-free programmed grammars are sufficiently powerful to generate their own derivation languages", a result known to be false for ordinary context-free grammars. In particular we can now claim a much tighter bound on the extra grammar-strength needed to generate the Szilard (or: derivation) languages of arbitrary context-free grammars (see Salomaa [10], p. 185 bottom): context-free ε-free programmed grammars will do.

Section 2 will contain relevant definitions and the proof of Lemma A. Section 3 will continue the constructions to prove Lemma B and a further generalization.

2. CONSTRUCTION FOR LEMMA A

For all preliminaries from automata and formal language theory which are not explicitly introduced here we refer to standard texts like Hopcroft & Ullman [5] and Salomaa [10]. We use ε to denote the empty string over any alphabet.

Context-free programmed grammars (or cfpg's, Rosenkrantz [9]) can be described as context-free grammars in which all productions carry a unique label and each production includes a clause for the selection of a next production after the former was "applied". Formally, a cfpg is a tuple $G = \langle V, \Sigma, L, P, S\rangle$ with V, Σ and S as usual, L a set of rule-labels, and P a finite set of productions of the form

$$(r)\quad A \to w \quad S(\alpha)\ F(\beta)$$

with $r \in L$ (unique for this rule), $A \in V - \Sigma$, $w \in V^*$, $\alpha \subseteq L$ (the success-field), and $\beta \subseteq L$ (the failure-field). Observe that α, β could be empty. If $w \neq \varepsilon$ for each production, then we call G a context-free ε-free programmed grammar (or $\text{cfpg}^{-\varepsilon}$).

Let pairs (x,r) in derivations with cfpg's indicate that "production r must be applied to intermediate string x". We write $(x,r) \Rightarrow (y,s)$ if one of two cases holds.

(i) A occurs in x (which means that r applies "successfully"), y is obtained from x by replacing some arbitrary occurrence of A by w, and $s \in \alpha$.

(ii) A does not occur in x (which means that r "fails"), $y = x$, and $s \in \beta$.

We write $(x,r) \Rightarrow (y,\emptyset)$ in the special event that the "next" production (s) must be chosen from an empty S- or F-field. Obviously, a derivation cannot proceed after reaching a pair $(y,\emptyset)$. Let $\overset{*}{\Rightarrow}$ be the reflexive, transitive closure of $\Rightarrow$.

Definition. L is a context-free programmed language (or context-free ε-free programmed language) if and only if a cfpg (or $\text{cfpg}^{-\varepsilon}$) G exists such that

$$L - \{\varepsilon\} = \{w \in \Sigma^* \mid (S, r_o) \overset{*}{\Rightarrow} (w, r_f) \text{ for some } r_o \text{ which labels a rule for } S \text{ and some } r_f \in \{\emptyset\} \cup L\}.$$

Let the families of context-free programmed and context-free ε-free programmed languages be denoted as CFPL and $\text{CFPL}^{-\varepsilon}$, respectively. From Rosenkrantz [9] and van Leeuwen [11] we conclude

Proposition 2.1 $\text{CFPL}^{-\varepsilon}$ is an AFL (which is strictly contained in the family of context-sensitive languages).

We assume the reader to be familiar with the notion of multi-tape machines, so our next description can be brief. A counter-tape is a pushdown-tape with a working alphabet of just a single letter. The scanned tape-symbol will be "a" as long as the counter is not empty, and "ε" if it is. In the latter case popping is prohibited, and the machine will have special instructions for handling counters when they're empty. The "δ-function" of a multi-counter machine will prescribe for each source-configuration a finite number of admissible transitions, each transition including a move-directive for the input head and a pop, leave, or push instruction for each counter. A computation begins with the machine in a specified initial state, its input head on the leftmost square of the input-tape, and its counters containing "ε". An input is acceptable if the machine can reach a designated final state, with the input head running off the righthand end of the input-tape.

In one-way machines the input-head can only move in left-to-right direction. If input is required for a transition, the machine will consume the "σ" in the current input-square and move its head right. If no input is required, the machine will consume an "ε" and leave the position of its head unaltered. Typically, after the input-head has moved, several ε-moves may follow before the contents of the next square (if any) is actually scanned.

Let $D : \mathbb{N} \to \mathbb{N}$ be an arbitrary function.

Definition. A one-way machine is said to be D(n)-delay bounded if and only if for all acceptable inputs of length n it has an (accepting) computation in which no more than D(n) consecutive ε-moves occur before or after any input-consuming move.

A o-delay bounded machine is traditionally called realtime, and any machine which is c-delay bounded (for some constant c) is called quasi-realtime. The terminology obviously applies to one-way multi-counter machines. The structure of quasi-realtime languages in general was investigated e.g. in Book & Greibach [1].

We can now prove

Lemma A. The quasi-realtime one-way multi-counter languages are (strictly) included in $CFPL^{-\varepsilon}$.

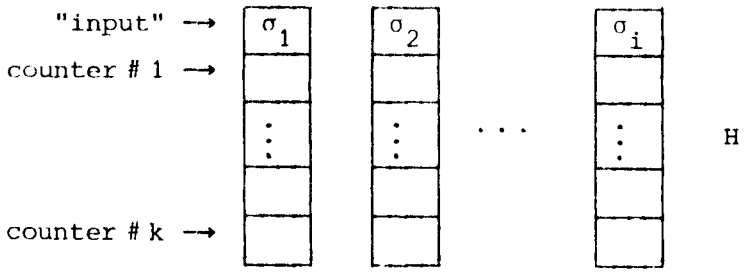
Proof

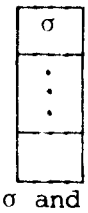
Let M be an arbitrary quasi-realtime one-way multi-counter machine, and assume that M is in fact c-delay bounded (some c). We may assume without loss of generality that M can make no ε-moves in its initial state

(thus, it always begins a computation with an input-consuming move)! A crucial observation is that there is a constant d such that

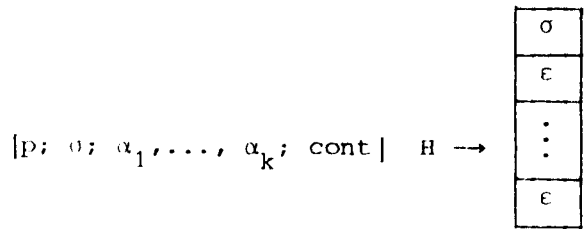
(*) for each acceptable input $\sigma_1 \ldots \sigma_n$ of length n, machine M must admit an accepting computation in which after consuming the i^{th} input-symbol ($1 \leq i \leq n$) none of its counters can grow larger than i.d before the next input-consuming move or halting.

One can take d = c+1, but any larger value for d will do for the construction below also. We shall design a $\text{cfpg}^{-\varepsilon}$ G which can perform a direct simulation of the computations of M, maintaining intermediate strings of the form



The monstrous symbols  contain k+1 track-squares, one holding an "actual" input-symbol σ and the remaining k capable of holding up to d a's (thus any one of $\varepsilon, a, \ldots, a^d$). The contents of the j^{th} such track-square should be understood to contribute to the contents of the j^{th} counter in some computation of M, on an input as appearing in the top-track squares from left to right. Notice that each counter-track can "store" up to i.d a's when i "inputs" have been generated, but we know from (*) that such a capacity is sufficient to simulate at least one accepting computation for each acceptable input.

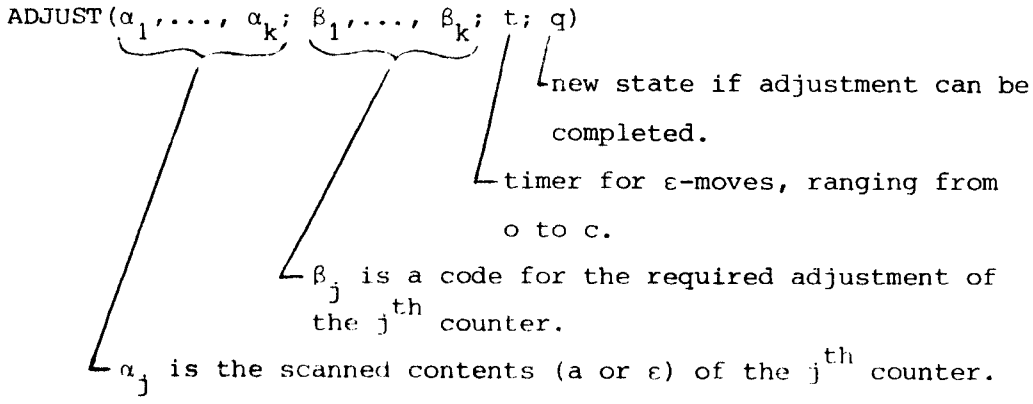
The idea is to "program" the application of productions



which indicate that M is to continue with the processing of "some" next input-symbol σ after it has proceeded to state p and while scanning $\alpha_1, \ldots, \alpha_k$ on the counters (as a result of the computation on the preceding

inputs). Each time such a "next" input has been generated, the grammar must "pause" and call a macro ADJUST(...) to update the counters as required by the recent input-consuming move and up to c subsequent ε-moves before it can "continue" with another $[p; \sigma; \alpha_1, \ldots, \alpha_k; \text{cont}]$-rule (with p and $\alpha_1 \ldots \alpha_k$ consistent with the new state of the computation and arbitrary σ). We describe the macro ADJUST first, using informal terms only as the reader can easily supply the tedious details of the actual productions needed.

The macro ADJUST is called as



, which the grammar generates when it chooses to simulate M by using $(q, \beta_1, \ldots, \beta_k) \in \delta(.., .., \alpha_1, \ldots, \alpha_k)$, knowing the current state and input for the move but guessing the scanned contents $\alpha_1, \ldots, \alpha_k$ of the current counters. It follows that the macro must lead through the following chain of checks and actions, easily achieved with a $\text{cfpg}^{-\varepsilon}$ block:

(i) for j from 1 to k,

if $\alpha_j = \varepsilon$ then

- verify that the j^{th} counter is empty, by checking that there is no symbol whose j^{th} counter-track square has contents $\in \{a, \ldots, a^d\}$

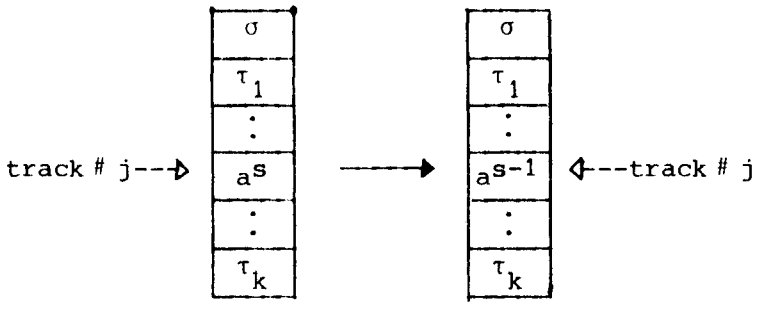
if $\alpha_j = a$ then

- verify that the j^{th} counter is not empty, by checking that there is at least one symbol whose j^{th} counter-track square has contents $\in \{a, \ldots, a^d\}$.

(ii) for j from 1 to k,

if β_j = pop then

- range through all possible productions of the form



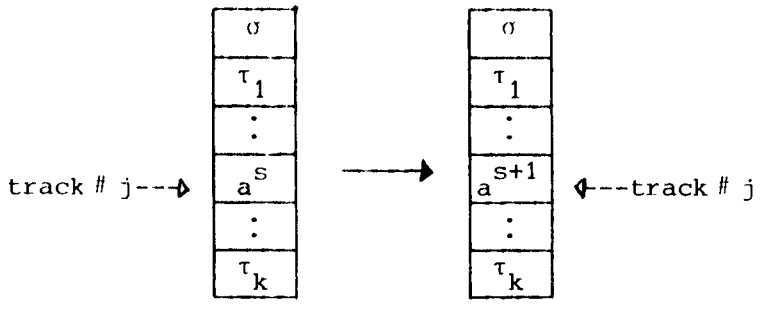
(for some enumeration of the choices of the fixed, remaining squares and $1 \leq s \leq d$), until you hit the first successful application.
Note that one must get to such a production, because a "pop" can only be given when the corresponding counter was non-empty.

if β_j = leave then

- do nothing, and just transfer control.

if β_j = push then

- range through all possible productions of the form



(for some enumeration of the choices of the fixed, remaining squares and this time $0 \leq s \leq d-1$), until you hit the first successful application.
Note that it may now happen that we find no such hit at all, precisely when the j^{th} track contains the maximum counter-value it can presently store (which is i.d). However, we know from (*) that for all acceptable inputs there must be an accepting computation of M which will lead us through here successfully.

(iii) if any of the required checks or actions in (i) and (ii) cannot be completed successfully, then control must be interrupted and transferred to some production like

[sink] H⟶H S(sink) F(sink)

effectively causing the grammar to get "hung up" on an intermediate string of ⋮-symbols (and perhaps H) which from here on cannot ever become "terminal".

(iv) if the checks and actions called on in (i) and (ii) all run to a successful ending, then we can prepare for the simulation of a next move of M (if any) and transfer control to any one of the following productions or blocks:

⋮

ADJUST(α_1,..., α_k; β_1,..., β_k; t+1; r)

⋮

⟵ for any α_1,..., α_k and for all r, β_1,..., β_k such that (r, β_1,..., β_k) ∈ δ(q, ε, α_1,..., α_k), provided that t+1 ≤ c
(which adjusts for another ε-move)

⋮

|q; σ; α_1,..., α_k; cont] or [q; σ; α_1,..., α_k; stop]

⋮

⟵ for any σ and any α_1,..., α_k
(which prepares for an input-consuming move on σ, with the option "stop" explicitly added if no more input-consuming moves are anticipated hereafter)

⋮

HALT

⟵ which is to be included here only when q is a final state of M
(HALT is another macro which we explain later but which essentially "reads out" the contents of the top-track of the current intermediate string and stops the derivation)

With the macro ADJUST at hand we can give the complete specification of G:

(i) the production |start|

S⟶H S(...) F(Ø)

⟵ with the success-field containing

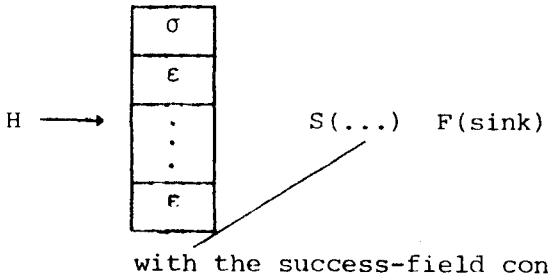
⋮

|q_o; σ; ε,..., ε; cont] or [q_o; σ; ε,..., ε; stop]

⋮

(σ ranging over all possible input-symbols)

(ii) the productions $[p;\ \sigma;\ \alpha_1, \ldots, \alpha_k;\ \text{stop}]$



with the success-field containing all calls of the form

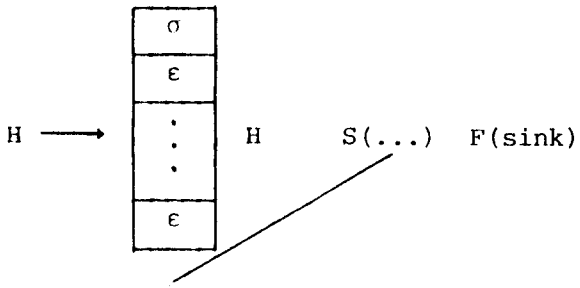
$$\text{ADJUST}(\alpha_1, \ldots, \alpha_k;\ \beta_1, \ldots, \beta_k;\ o;\ q)$$

for which

$$(q, \beta_1, \ldots, \beta_k) \in \delta(p, \sigma, \alpha_1, \ldots, \alpha_k)$$

Thus, we drop the H once a choice for the last "real" input for M is generated (as it would otherwise stand in the way to obtain the top-track as a terminal string, provided it is accepted by M). Observe that the failure-field is "sink", and any attempt to "stop again" after an ADJUST will kill the derivation on a non-terminal product.

(iii) the productions $[p;\ \sigma;\ \alpha_1, \ldots, \alpha_k;\ \text{cont}]$



with the success-field (as above) containing all calls of the form

$$\text{ADJUST}(\alpha_1, \ldots, \alpha_k;\ \beta_1, \ldots, \beta_k;\ o;\ q)$$

for which

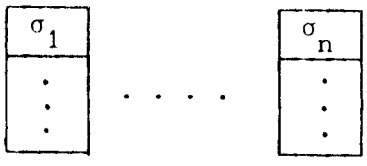
$$(q, \beta_1, \ldots, \beta_k) \in \delta(p, \sigma, \alpha_1, \ldots, \alpha_k)$$

Observe that the failure-field is again "sink", so any attempt to "continue" after an ADJUST while we had prepared to "stop" by dropping the H will kill the derivation on a non-terminal string.

Putting it all together one can see that G faithfully simulates all and only those computations of M in which the counters remain bounded as expressed in (*). Each time G has completed the simulation of a move (in the last stage of an ADJUST) and finds M in an accepting state, the intermediate string must be of the form

(a)

or

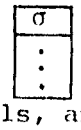


(b)

with the top-track containing an acceptable string. Referring to the options of control-transfer proposed in the last phase of any ADJUST, we now see that it gives G the following correct options:

(i) G can continue to simulate a move of M
(which attempts to "try again" on ε-moves in case a, but anticipates an extension to some other acceptable string in case b)

(ii) G can "halt" and call the macro HALT for producing the terminal product $\sigma_1 \ldots \sigma_n$
(it should do it in case a, but detect case b as improper)

The macro HALT simply converts all 

-symbols into σ (cycling through an enumeration of all possible symbols, and passing to the next one once you "fail" to find any more copies of some), and ends with a production

[finish] $H \longrightarrow H \quad S(\emptyset) \quad F(\emptyset)$

It is straightforward to see that G can generate all and only those strings $\sigma_1 \ldots \sigma_n$ which can be accepted by M, i.e. G is a $\text{cfpg}^{-\varepsilon}$ generating precisely the language accepted by M.

□

The given argument could be simplified even further, using that each quasi-realtime one-way multi-counter machine can be reduced to an equivalent machine of this type which actually operates in realtime (a suggestion due to Paul Vitanyi). As it stands, the proof shows that cfpg's can "realize" quasi-realtime machines directly without the need for much technical complication, and it has the advantage that we can see clearly what to do if we attempt to simulate more general delay-bounded machines.

There are various ramifications of the technique used in the proof. For instance, it is not hard to show by a related argument that the family of unary context-free ε-free programmed languages is precisely the family of unary multi-head finite automaton languages (which are the unary NLOG

languages!). With this theorem, the traditional result that the context-free ε-free programmed languages are strictly contained in CS becomes very easy to prove also: just apply the tape-hierarchy theorem for unary languages (Hartmanis & Berman [4]). We note that unary languages and diagonalization were previously used to simplify the proof of strict inclusion by Jeanrond [7], although he didn't have a complete characterization of the unary class.

3. CONSTRUCTION FOR LEMMA B

We need some more, technical concepts. The general idea of derivation-languages for grammars should be familiar (see e.g. Salomaa [10], note 6.2). The derivation language of a context-free programmed grammar G consists of all strings which can be read as the sequence of (labels of) applied productions in the derivation of some terminal string.

Definition. The derivation language of a cfpg$^{-\varepsilon}$ or cfpg G is the set

$$D(G) = \{r_1 \ldots r_t \mid r_1 \text{ labels a rule for } S, \text{ and there is some } w \in \Sigma^* \text{ such that } (S,r_1) \xRightarrow{r_1 \ldots r_t} (w,r) \text{ for some } r \in \{\emptyset\} \cup L\}$$

Note that for the derivation language it is immaterial whether G contains ε-rules or not.

We also need another kind of derivation language, in which we collect the control-words which lead derivations to a designated label (regardless of being able to complete the derivations to a terminal string).

Definition. The production language of a cfpg$^{-\varepsilon}$ or cfpg G with respect to label r is the set

$$D(G,r) = \{r_1 \ldots r_t \mid r_1 \text{ labels a rule for } S \text{ and } (S,r_1) \xRightarrow{r_1 \ldots r_t} (x,r) \text{ for some } x\}$$

Again it is immaterial for production languages whether G contains ε-rules or not. Note in the definition that x need not be terminal, and the derivation language of G may indeed be completely distinct from any of its associated production languages.

Before we can state a relationship between the derivation- and production-languages of context-free programmed grammars, we need one more auxiliary concept (see e.g. Ginsburg [3], p. 33).

Definition. For an arbitrary language L, the homomorphism h is said to be ε-limited on L if and only if there is a constant d such that for all $w \in L$

$$w = xyz \;\&\; h(y) = \varepsilon \;\Rightarrow\; |y| \leq d$$

Proposition 3.1 Each cfpg G can be effectively modified to an equivalent cfpg G' (which is ε-free whenever G was) such that D(G) is an ε-limited homomorphic image of some D(G',r).

Proof

Let the non-terminal alphabet of G be $\{A_1, \ldots, A_d\}$. Let \$ be a label not occurring in G, and add a production

$$|\$| \quad A_1 \longrightarrow A_1 \quad S(\emptyset) \quad F(\emptyset)$$

We shall modify G further to obtain a G' such that D(G) is an ε-limited

homomorphic image of D(G',$).

The idea is to add productions to G which let you exit for $ when it is detected that a terminal string has been produced. To achieve it, we replace each production

$$[r] \quad A_i \rightarrow w_j \quad S(\alpha) \quad F(\beta)$$

in G by

$$[r] \quad A_i \rightarrow w_j \quad S(\alpha) \quad F(\mathrm{CHECK}(r,\beta))$$

where CHECK is the following macro, specified for symbolic parameters r and β:

$$\begin{array}{lll} [r/1] & A_1 \rightarrow A_1 & S(\beta) \quad F(r/2) \\ \vdots & \vdots & \\ [r/d-1] & A_{d-1} \rightarrow A_{d-1} & S(\beta) \quad F(r/d) \\ [r/d] & A_d \rightarrow A_d & S(\beta) \quad F(\$) \end{array}$$

It is easily verified that G' "realizes" the same flow of control as G, as long as non-terminals remain in the intermediate string.

Words in D(G',$) and D(G) correspond, under the stipulation that in the former each failing G-production is followed by up to d rule-applications in the CHECK-macro before control transfers normally to the "next" G-production (or to the $-rule to exit). The homomorphism h erasing [../i] labels ($1 \leq i \leq d$) must be ε-limited on D(G',$) and obviously maps it onto D(G). □

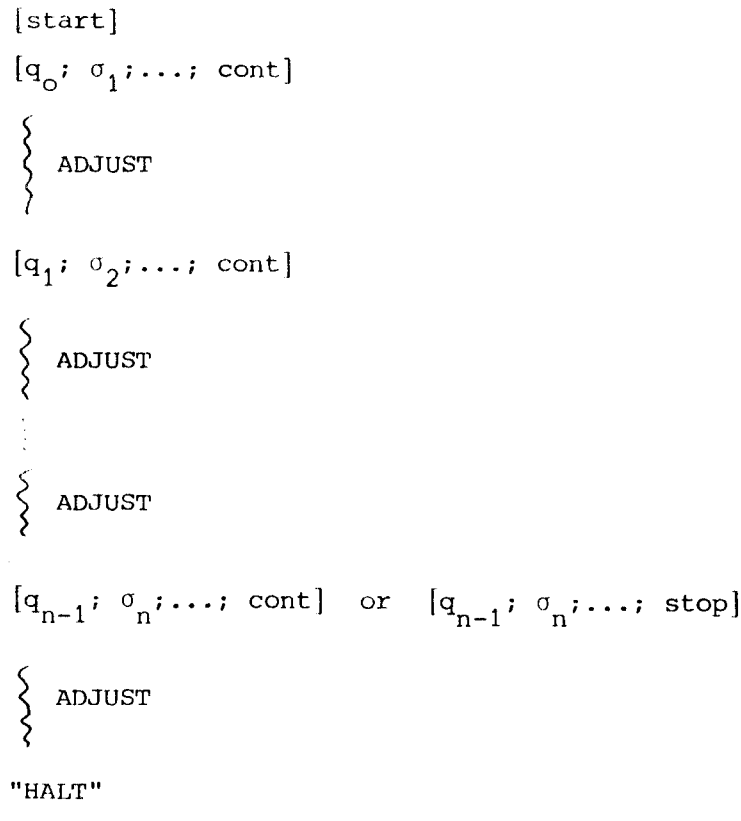
Inspired by lemma A, we can now prove lemma B and obtain an interesting characterization of the quasi-realtime one-way multi-counter languages in terms of context-free programmed grammars.

Lemma B. The family of quasi-realtime one-way multi-counter languages coincides with the family of ε-limited homomorphic images of production languages of context-free programmed grammars.

Proof

(a) To show that each quasi-realtime one-way multi-counter language can be obtained as the ε-limited homomorphic image of a context-free programmed production language we must go back to the cfpg G constructed in the proof of lemma A. Recall that we took an arbitrary quasi-realtime one-way multi-counter machine M, and designed G as an almost perfect simulator of M's (accepting) computations.

G reaches a call to macro HALT (which it may or may not choose to follow) if and only if it just completed the required adjustments for another move and finds M in an accepting state. Inspecting the structure of G one may easily verify that it can reach this stage only by going through a chain of productions in a sequence of the following form



with $\sigma_1 \ldots \sigma_n$ appearing in the top-track of the intermediate string and the simulation reaching an accepting state. It follows that $\sigma_1 \ldots \sigma_n$ is an accepted input, and each accepted input for M can be so obtained.

Thus, M's language may be obtained from D(G,HALT) by applying a homomorphism h which rewrites each [.; σ; ...] to $\sigma (\sigma \in \Sigma)$ but erases any symbol not of this form. Observe that the number of steps used to adjust the counters through the ADJUST-macro in the simulation of a single, admissible move is bounded by some constant e. Now recall that M was c-delay bounded for some c, and see that G never does more than c further calls on ADJUST (to handle ε-moves) before it simulates another input-consuming move or calls HALT. It follows that in words of D(G,HALT) any portion for adjustments after each [.; σ; ...] symbol must be bounded by some constant like (c+1).e, and h is indeed ε-limited on D(G,HALT).

(b) To show the reverse inclusion we simplify the problem slightly, by using that the family of quasi-realtime one-way multi-counter languages is itself closed under ε-limited homomorphisms. Thus, it is sufficient to prove that each production language of a context-free programmed grammar G can be accepted by some quasi-realtime one-way multi-counter machine M. (It will appear that M can even be deterministic here.) The construction is related to an argument developed in Igarashi [6].

The idea is that in simulating the production-sequences of G it is immaterial to know what terminal strings they lead to and, using the context-

free nature of the replacements, we may just as well "fold" the intermediate strings and keep track of their Parikh-vectors only.

Let the non-terminal alphabet of G be $\{A_1,\ldots, A_d\}$, and \$ be a designated label in G. We shall design a machine M to perform a step-by-step simulation of G's derivations, according to control-words in D(G,\$). M will use d counters $\#A_1,\ldots,\#A_d$ and have the labels of G for states. The action of M on an arbitrary input $r_1\ldots r_t$ is probably described easiest by means of the following pseudo-ALGOL program:

```
comment "initialize"
#S ← #S+1;
p ← input;
verify that p is rule for S (reject otherwise);
put M in state p;
while M has not run off the right end of the tape do
    begin
        decode p as [p]  A→w  S(α)  F(β)
        if #A = ∅ then
                    comment "failing application"
                    set M to a state in β (reject of ∅)
                else
                    comment "successful application"
                    #A ← #A-1;
                    #A1 ← #A1 + #A1(w);
                    .
                    .
                    .
                    #Ad ← #Ad + #Ad(w);
                    set M to a state in α (reject if ∅)
        fi;
        p ← input
    end
od;
if M has now reached state $ then accept else reject;
```

Clearly M can perform the test and update its counters (if needed) for each input p in bounded time, and the machine is quasi-realtime. As it updates its counters consistent with the Parikh-vector $(\#_{A_1}(x),\ldots,\#_{A_d}(x))$ of the intermediate strings x, the machine accepts precisely the control-words which send G to \$ as required.

□

The construction for lemma B is not specifically restricted to quasi-realtime machines only. With a suitable concept of "D(n)-limited homomorphisms" (the obvious generalization of ε-limited homomorphisms), one can generalize the result and prove in very much the same way that the family of c.D(n)-delay bounded one-way multi-counter languages (for arbitrary c's) is precisely the family of c'.D(n)-limited homomorphic images of context-free programmed production languages. Whereas this result may not be very appealing in its full generality, it does indicate the close connection between the computations of one-way multi-counter machines and derivations with context-free programmed grammars. This may be another argument explaining why context-free programmed grammars have proved to be a powerful vehicle for language generation.

Combining lemmas A and B gives some further interesting results for the general study of derivation-languages. First, derivation languages of context-free programmed grammars are quasi-realtime one-way (deterministic) multi-counter languages (by 3.1 and lemma B), and therefore contained in DLOG. This result was also indicated in Igarashi [6]. A second observation appears to be of more importance.

Proposition 3.2 The derivation languages of context-free programmed grammars are (strictly) contained in the family of context-free ε-free programmed languages.

Proof

By 3.1, lemma B, and lemma A (applied in this order).

□

This result is a substantial improvement of the previously known theorem (see Salomaa [10], p. 185) that merely asserts inclusion in the context-sensitive languages. Informally, it implies that the derivation-languages of all traditional generalizations of context-free grammars which admit a "stepwise" simulation by means of some context-free programmed grammar are contained in $CFPG^{-\varepsilon}$ also. In particular, it applies to the (free) derivation languages or Szilard languages of ordinary context-free grammars.

4. REFERENCES

[1] Book, R.V., and S. Greibach, Quasi-realtime languages, Math. Syst. Th. 4 (1970) 97-111.

[2] Fischer, P.C., Turing machines with restricted memory access, Inf. & Control 9 (1966) 364-379.

[3] Ginsburg, S., Formal languages: algebraic and automata-theoretic properties, Fund. St. in Computer Sci. #2, North-Holland / Amer. Elsevier, Amsterdam (1975).

[4] Hartmanis, J., and L. Berman, A note on tape-bounds for SLA language processing, Conf. record 16th Ann. Symp. Foundations of Computer Science, Berkeley, Oct. 13-15 (1975), pp. 65-70.

[5] Hopcroft, J.E., and J.D. Ullman, Formal languages and their relation to automata, Addison-Wesley, Reading, Mass. (1969).

[6] Igarashi, Y., The tape-complexity of some classes of Szilard Languages, SIAM J. Computing 6 (1977) 460-466.

[7] Jeanrond, H-J., Some remarks on programmed grammars, Diplomarbeit, Fachbereich f. Angewandte Mathematik u. Informatik, Universität des Saarlandes, Saarbrücken (1975).

[8] Minsky, Computation: finite and infinite machines, Prentice-Hall, Englewood Cliffs, NJ (1967).

[9] Rosenkrantz, D.J., Programmed grammars and classes of formal languages, JACM 16 (1969) 107-131.

[10] Salomaa, A., Formal languages, Acad. Press, New York, NY (1973).

[11] van Leeuwen, J., Rule-labeled programs: a study of a generalization of context-free grammars and some classes of formal languages, Ph. D. Thesis, Dept. of Mathematics, University of Utrecht, Utrecht (1972).

[12] van Leeuwen, J., Extremal properties of non-deterministic time-complexity classes, in: E. Gelenbe & D. Potier (ed.), International Computing Symposium 1975, North-Holland / American Elsevier, Amsterdam (1975), pp. 61-64.