

# Eliminating null rules in linear time\*

M. A. Harrison and A. Yehudai†

Computer Science Division, University of California at Berkeley

We present a linear time algorithm for eliminating null rules in context free grammars. Until recently all algorithms given in the literature for this problem required exponential time.  
(Received November 1978; revised January 1980)

Null rules (i.e. productions like  $A \rightarrow A$  where  $A$  is the empty string) are often undesirable in a context free grammar either for a theoretical reason (it may be easier to prove properties of grammars with no such rules) or for practical reasons (some parsing techniques may fail to work in the presence of null rules).

It is well known (Bar-Hillel, Perles and Shamir, 1962) that for every context free grammar  $G$  one can construct a context free grammar  $G'$  with no null rules that is equivalent to  $G$  (i.e. generates the same language). This process is called a transformation (or, more specifically a null rule eliminating transformation).

The algorithm from Bar-Hillel, Perles and Shamir (1962), as well as some other algorithms found in the literature, turn out to be quite inefficient. They are intended mainly as constructive proofs for a Normal Form theorem. Also, their inefficiency surfaces only on certain specially designed grammars. Still, it is important to investigate the complexity of null rule elimination. We prove that this can actually be done in linear time.

While the paper is concerned with a problem interesting to compiler writers, it can also be viewed as an example of algorithmic improvement. In particular the development of the programs for the new algorithm are done in a careful systematic fashion, as details of the implementation (i.e. the choice of data structures) significantly affect the programs' efficiency.

The remainder of the Introduction provides some basic definitions and notation. In Section 1 we analyse the classical algorithm and show that it is exponential. We also discuss some other algorithms that are exponential. Then we show how one can improve the performance of the algorithm. Section 2 considers the computation of all non-terminals of a grammar that can generate the empty string. This is needed as a subroutine to the main algorithm. Finally, in Section 3, we obtain our main results by combining some of the previous results. We use fairly standard notation and repeat only some of the elementary definitions; for details see Harrison (1978) and Hopcroft and Ullman (1969). By a grammar we always mean a context free grammar.

We would like to present our algorithms in a readable way without hiding the main complexity issues. We choose to write algorithms in Pidgin ALGOL (cf. Aho, Hopcroft and Ullman, 1974). This representation enables us to specify as much or as little of the actual implementation of the algorithm as we find necessary. We then analyse the time complexity assuming the algorithm is executed on a reasonable model of a computer. We are interested in the asymptotic behaviour of the worst case complexity. Since the complexity is computed as a function of the input size, and since the input to the algorithms is (an encoding of) a grammar we need to discuss the size of such an encoding. A reasonable encoding consists mostly of a list of the productions in the grammar. (The size of any additional information such as the list of nonterminals and terminals, as well as delimiters signifying end of production, etc. will be

smaller.) There are two principal ways to measure the size of the encoding (i.e. of the production list of a grammar).

## Definition 1

Let  $G = (V, \Sigma, P, S)$  be a context free grammar. Define

$$|G| = \sum_{\substack{A \rightarrow a \\ \text{in } P}} |A\alpha|$$

and

$$\|G\| = |G| \cdot \log_2 |V|.$$

$|G|$  is simply the number of symbols involved in productions.

$\|G\|$  is a more realistic measure because it takes into account the number of bits needed to encode each symbol in  $V$  (assuming a fixed alphabet). Unless otherwise specified  $n$  will denote the size of the input using either measure.

Whenever  $\|G\|$  is assumed as the size measure, we need to estimate  $|G|$  and  $|V|$  in order to compute the complexity. The following lemma establishes some relationship between these quantities.

## Lemma 1

For any context free grammar  $G = (V, \Sigma, P, S)$ , if  $L(G) \neq \emptyset$ ,  $L(G) \neq \{A\}$ , and if every letter in  $V$  occurs in at least one production, then

$$(a) \ 2 \leq |V| \leq |G|$$

$$(b) \ |V| \leq \frac{2n}{\log n} \text{ where } n = \|G\| = |G| \log |V|. \text{ (All logs are to the base 2.)}$$

## Proof

Since  $S \in N$ ,  $|N| \geq 1$ . Since  $\Sigma \neq \emptyset$ , we have  $|V| = |N| + |\Sigma| \geq 1 + 1 = 2$ . The upper bound of (a) follows from the assumption that each symbol of  $V$  appears in at least one production.

From (a), we have

$$n = |G| \log |V| \geq |V| \log |V| \geq 2 \log 2.$$

Consider the function

$$f(x) = 2\sqrt{x} - \log x.$$

It can be seen that for  $x \geq 2$   $f(x) > 0$ . It follows that, for all  $n \geq 2$

$$2\sqrt{n} > \log n.$$

So

$$\frac{2n}{\log n} > \sqrt{n}.$$

Taking logs and multiplying by  $\frac{2n}{\log n} > 0$  we obtain

$$\frac{2n}{\log n} \log \left( \frac{2n}{\log n} \right) > n.$$

\*Research supported by National Science Foundation Grants GJ-43332 and MCS74-07636-A01.

†Division of Computer Science, Tel-Aviv University, Ramat-Aviv, Tel-Aviv, ISRAEL.

Hence

$$\frac{2n}{\log n} \log \left( \frac{2n}{\log n} \right) > |V| \log |V|$$

and it follows that

$$\frac{2n}{\log n} > |V|$$

□

When discussing the complexity of algorithms we will often make two evaluations according to the size measure we use. The use of a specific measure for the size of the input will imply that if the algorithm has a grammar as its output then it is assumed to be written out in the same 'format'.

An interesting property of many of the algorithms that perform grammatical transformation is that their time complexity is dominated by the size of the output grammar. In such cases we need only evaluate the size of the output grammar (using either size measure), and show that the computation itself is of the same order of magnitude as the size, in order to obtain the algorithm's complexity.

Next we introduce some 'Normal Forms' of grammars.

#### Definition 2

A grammar  $G = (V, \Sigma, P, S)$  is said to be

1. *Reduced* if  $P = \emptyset$  or for every  $A \in V$ ,  $S \Rightarrow \alpha A \beta \Rightarrow w$  for some  $\alpha, \beta \in V^*$ ,  $w \in \Sigma^*$ .
2.  *$\Lambda$ -free* if  $P \subseteq N \times V^+ \cup \{S \rightarrow \Lambda\}$  and if  $S \rightarrow \Lambda$  in  $P$  implies that  $S$  does not appear in a right hand side of any production.
3. *chain-free* if  $P \cap N \times N = \emptyset$ .
4. in *2-Normal-Form* (2NF) if  $P \subseteq N \times V^2$ . ( $V_\Lambda$  denotes the set  $V \cup \{\Lambda\}$ ) Hunt, Szymanski and Ullman (1975)
5. in *Chomsky-Normal-Form* (CNF) if it is  $\Lambda$ -free and  $P \subseteq N \times (N^2 \cup \Sigma) \cup \{S \rightarrow \Lambda\}$ .

It is well-known that every language has a reduced grammar. In Yehudai (1977) it is shown that reduction can be done in linear time. The definition of a  $\Lambda$ -free grammar allows  $S \rightarrow \Lambda$  to be used only in generating  $\Lambda$ . 2NF only limits the length of the right hand side of a production, while CNF allows only three types of productions:  $A \rightarrow BC$ ,  $A \rightarrow a$  or  $S \rightarrow \Lambda$  where  $A, B, C \in N$  and  $a \in \Sigma$ .

#### 1. Eliminating null rules

We begin this section by presenting the classical algorithm, due to Bar-Hillel, Perles and Shamir (1962) for null rule elimination. The construction is very simple, but the grammar can grow exponentially.

#### Algorithm 1

Input:  $G = (V, \Sigma, P, S)$  a reduced grammar

Output: grammar  $G'$  such that  $L(G') = L(G)$  and  $G'$  is  $\Lambda$ -free

begin

NULL :=  $\{A \in N \mid A \Rightarrow \cdot\}$ ;

$N' := N$ ;

$P' := \emptyset$ ;

for all  $A \rightarrow \alpha \in P$  do comment  $\alpha = \alpha_0 B_1 \dots B_n \alpha_n$ ,  $n \geq 0$ ,  
 $B_i \in \text{NULL}$ ,  $\alpha_i \in (V - \text{NULL})^*$ ;

for all  $(X_1, X_2, \dots, X_n) \in \{B_1, \Lambda\} \times \{B_2, \Lambda\} \times \dots \times \{B_n, \Lambda\}$  such that  $\alpha_0 X_1 \alpha_1 \dots X_n \alpha_n \neq \Lambda$  do  $P' :=$   
 $P' \cup \{A \rightarrow \alpha_0 X_1 \alpha_1 \dots X_n \alpha_n\}$ ;

if  $S \in \text{NULL}$  then begin  $N' := N' \cup \{S\}$ ;  $P' := P' \cup$   
 $\{S' \rightarrow S, S' \rightarrow \Lambda\}$  end

else  $S' := S$ ;

$G' := (N' \cup \Sigma, \Sigma, P', S')$

end.

This algorithm should be followed by reduction, since some nonterminals may become useless.

The computation of NULL needs to be specified. It turns out, however, that the above algorithm has so large a time complexity that the way NULL is computed is irrelevant.

#### Lemma 2

Algorithm 1 performs null rule elimination in exponential time.

#### Proof

The correctness of this algorithm is proved in Harrison (1978). We will now present a grammar  $G$ , for which Algorithm 1 produces a  $\Lambda$ -free grammar  $G'$  whose size is exponentially larger than that of  $G$ . This will be sufficient to prove the result since the size of the output is clearly a lower bound on the time complexity.

More precisely we will consider an infinite family of grammars  $G_1, G_2, \dots, G_k, \dots$  where  $G_k = (V_k, \Sigma_k, P_k, A)$ ,  $N_k = \{A, B_1, B_2, \dots, B_k\}$ ,  $\Sigma_k = \{a_1, a_2, \dots, a_k\}$  and  $P_k = \{A \rightarrow B_1 B_2 \dots B_k\} \cup \{B_i \rightarrow a_i, B_i \rightarrow \Lambda \mid 1 \leq i \leq k\}$ . (In subsequent discussions the subscript  $k$  will be omitted whenever no confusion may arise and we will talk about  $G, N, \Sigma, V, P$ , etc.) We can see that  $\text{NULL} = N$  since  $B_i \Rightarrow \Lambda$  for each  $i$  and

$A \Rightarrow B_1 B_2 \dots B_k \Rightarrow \Lambda$ . The production  $A \rightarrow B_1 B_2 \dots B_k$  in  $P$  will yield  $2^k - 1$  productions in  $P'$ , namely  $A \rightarrow \beta$  for every non-null subword  $\beta$  of  $B_1 B_2 \dots B_k$ . The result of the transformation (again omitting subscripts) is  $G' = (V \cup \{S\}, \Sigma, P', A')$  where  $P' = \{A' \rightarrow A, A' \rightarrow \Lambda\} \cup \{A \rightarrow X_1 X_2 \dots X_k \mid X_i \in \{B_i, \Lambda\}, X_1 X_2 \dots X_k \neq \Lambda\} \cup \{B_i \rightarrow a_i \mid 1 \leq i \leq k\}$ .

We can compute the sizes of the grammars involved:

$|V| = 2k + 1$ ,  $|G| = 4k + 1$ ,  $|V'| = 2k + 2$  and  $|G'| = (k + 2)2^{k-1} + 2k + 2$ .  $|G'|$  is exponentially larger than  $|G|$ , and the same is true for  $\|G'\|$  as a function of  $\|G\|$ . □

The proof indicates a stronger result than the one stated in the lemma.

#### Corollary

Any algorithm for null rule elimination which produces the same output grammar as Algorithm 1 takes at least exponential time.

Graham (1974) gives an algorithm to eliminate null rules without destroying the  $(m, n)$  BRC property. While the latter requirement calls for a more complicated construction than Algorithm 1, it does resemble it. In particular, when that algorithm is applied to the grammar  $G$  in the proof of Lemma 2 (which is clearly  $(k, k)$  BRC), the resulting grammar is essentially  $G'$ , which is exponentially larger.

Rosenkrantz and Stearns (1970) present a null rule elimination algorithm for LL grammars which guarantees an  $\text{LL}(k + 1)$  grammar as a result if the original grammar is  $\text{LL}(k)$ . This algorithm cannot be used for arbitrary grammars since the construction is shown to produce a finite number of non-terminals only for unambiguous grammars.

The question to ask at this point is: why does this algorithm produce such large grammars, and is there any better way to do it? Clearly, the exponential growth is the result of a 'subset construction' reminiscent of the transformation from non-deterministic to deterministic finite automata (Harrison, 1978; Hopcroft and Ullman, 1969). The following observation proves useful in the realisation that unlike the finite automaton case, null rule elimination may be done without possible exponential explosion.

#### Lemma 3

Let  $G = (V, \Sigma, P, S)$  be a reduced grammar and let  $l \geq 0$  such that for all  $A \rightarrow \alpha$  in  $P$ ,  $|\alpha| \leq l$ .

Algorithm 1, when applied to  $G$ , yields a grammar  $G'$  whose size depends upon that of  $G$  as follows

$$\begin{aligned} |G'| &\leq 2^l |G| \\ |V'| &= |V| \\ \|G'\| &\leq 2^l \|G\| \end{aligned}$$

*Proof*

Algorithm 1 replaces each production  $A \rightarrow \alpha_0 B_1 \dots B_n \alpha_n$  by at most  $2^n$  productions each of no greater length than the original one. But  $n \leq |\alpha_0 B_1 \dots B_n \alpha_n| \leq l$  hence  $|G'| \leq 2^l |G|$ . The number of nonterminals is unchanged hence  $|V'| = |V|$ . The bound for  $\|G\|$  follows by definition  $\square$

Since  $l$  may be forced to be small by an appropriate transformation (which, as we shall see, is quite efficient) there is a good prospect that  $\Lambda$ -rules can be removed without huge increases in size. While it is possible to obtain an algorithm that directly eliminates  $\Lambda$ -rules (cf. Yehudai, 1977), it is quite complicated. We therefore use a two step approach.

The following is a simple algorithm to convert any grammar to 2NF. This is done by factoring 'long' right hand sides and introducing new nonterminals where necessary. It is essentially the classical algorithm for Chomsky-Normal-Form but we use it in a broader context by not requiring the input grammar to be  $\Lambda$ -free or chain free.

**Algorithm 2**

Input:  $G = (V, \Sigma, P, S)$  a reduced grammar

Output: a grammar  $G'$  in 2NF such that  $L(G') = L(G)$

**begin**

$N' := N$ ;

$P' := \emptyset$ ;

**for all**  $A \rightarrow \alpha \in P$  **do**

**begin**

**if**  $|\alpha| \leq 2$  **then**  $P' := P' \cup \{A \rightarrow \alpha\}$

**else begin**

**comment**  $\alpha = X_1 X_2 \dots X_r, r \geq 3, X_i \in V$ ;

**for**  $i := 1$  **to**  $r - 2$  **do**

**begin**

$N' := N' \cup \{C_i(A \rightarrow \alpha)\}$ ;

**comment** abbreviate  $C_i$ , let  $C_0 = A$ ;

$P' := P' \cup \{C_{i-1} \rightarrow X_i C_i\}$

**end**;

$P' := P' \cup \{C_{r-2} \rightarrow X_{r-1} X_r\}$

**end**

**end**;

$G' := (N' \cup \Sigma, \Sigma, P', S)$

**end.**

**Lemma 4**

Algorithm 2, when applied to  $G = (V, \Sigma, P, S)$  correctly produces an equivalent grammar  $G'$  in 2NF. Moreover if  $G$  is  $\Lambda$ -free (chain free) then so is  $G'$ .

*Proof*

It is easy to see that  $P'$  does not contain any production with a right hand side longer than 2. That  $L(G') = L(G)$  can be seen as a result of the following claims.

**Claims**

For all  $A \in N, \alpha, \beta \in V^*$ ,

1. If  $A \rightarrow \alpha \in P$  then  $A \xrightarrow{G'} \alpha$

2. if  $A \xrightarrow{G'} \beta$  then  $A \xrightarrow{G'} \beta$

3. Let  $A \rightarrow \alpha$  be in  $P, \alpha = X_1 X_2 \dots X_r$  for some  $X_1, X_2, \dots, X_r \in V$  and let  $C_i = C_i(A \rightarrow \alpha)$  be in  $N'$  for some  $i, 0 \leq i \leq$

$r - 2$ . If  $C_i \xrightarrow{G'} \beta$  then this derivation can be factored

$$C_i \xrightarrow{G'} X_{i+1} \dots X_r \xrightarrow{G'} \beta.$$

4. If  $A \xrightarrow{G'} \beta$  then  $A \xrightarrow{G} \beta$

Claim 1 can be proved by induction on  $|\alpha|$ , Claim 2 by induction on the length of the derivation, Claim 3 by induction on  $r - i = |\alpha| - i$  (with basis  $r - i = 2$ ), and Claim 4 by induction on the length of the derivation.

Finally, productions with right hand side 0 or 1 are included in  $P'$  only if they are in  $P$  so that the algorithm does preserve  $\Lambda$ -freeness and chain-freeness.  $\square$

We note in passing that if  $G$  is  $\Lambda$ -free and chain-free then only a minor modification is required to put the output grammar  $G'$  in Chomsky-Normal-Form.

**Lemma 5**

Algorithm 2 yields a grammar  $G'$  whose size depends upon that of  $G$  as follows:

$$\begin{aligned} |G'| &\leq 3 |G| \\ |N'| &\leq |N| + |G| \\ \|G'\| &= O(\|G\| \log \|G\|). \end{aligned}$$

The time complexity of the algorithm is dominated by the size of the output.

*Proof*

For every  $A \rightarrow \alpha \in P$ , where  $|\alpha| = r$  (and the production contributes  $r + 1$  to  $|G|$ ), either  $A \rightarrow \alpha \in P'$  (if  $r \leq 2$ ) or else we get  $r - 1$  productions  $(C_i \rightarrow X_{i+1} C_{i+1}, 0 \leq i < r - 2, C_{r-2} \rightarrow X_{r-1} X_r)$  in  $P'$ . In this latter case we also added  $r - 2$  new nonterminals to  $N' - N$ .

Hence

$$|G'| \leq \sum_{\substack{A \rightarrow \alpha \in P \\ r = |\alpha| \leq 2}} |A\alpha| + \sum_{\substack{A \rightarrow \alpha \in P \\ r = |\alpha| > 2}} 3(|A\alpha| - 2)$$

so that

$$|G'| \leq \sum_{A \rightarrow \alpha \in P} 3 |A\alpha| = 3 |G|$$

also

$$|N'| = |N| + \sum_{\substack{A \rightarrow \alpha \in P \\ |\alpha| > 2}} (|\alpha| - 3) \leq |N| + |G|$$

we have  $|G'| = O(|G|)$  and  $|V'| = O(|G|)$

$$\|G'\| = |G'| \log |V'| = O(|G| \log |G|)$$

and since  $|G| \leq \|G\|$

$$\|G'\| = O(\|G\| \log \|G\|)$$

The statement about time complexity is obvious as there is virtually no computation done.  $\square$

We can now perform null rule elimination in the following way.

**Algorithm 3**

Input:  $G = (V, \Sigma, P, S)$  a reduced grammar

Output:  $G'$  a  $\Lambda$ -free grammar such that  $L(G') = L(G)$

**begin**

apply algorithm 2 to  $G$ , obtaining  $G_1$  in 2NF

apply algorithm 1 to  $G_1$ , obtaining  $G'$  a  $\Lambda$ -free grammar in 2NF

**end.**

The following result relates to algorithm 3.

**Lemma 6**

Algorithm 3 correctly computes a  $\Lambda$ -free grammar  $G'$  such that  $L(G') = L(G)$  and the size of  $G$  depends upon that of  $G$  as follows.

$$\begin{aligned}
|G'| &\leq 12|G| \\
|V'| &\leq 2|G| \\
\text{and } \|G'\| &= O(\|G\| \log \|G\|)
\end{aligned}$$

**Proof**

The correctness of the algorithm is self evident. To compute the sizes we note that Lemma 5 yields

$$\begin{aligned}
|G_1| &\leq 3|G| \text{ and} \\
|V_1| &= |N_1| + |\Sigma| \leq |N| + |G| + |\Sigma| \\
&= |V| + |G| \leq 2|G|
\end{aligned}$$

and by Lemma 3

$$|G'| \leq 4|G_1| \text{ and } |V'| = |V_1|.$$

Hence

$$\begin{aligned}
|G'| &\leq 12|G| \\
|V'| &\leq 2|G|
\end{aligned}$$

and as in Lemma 5

$$\|G'\| = O(\|G\| \log \|G\|).$$

□

Lemma 6 falls short of stating the time complexity of the algorithm. Before we can do this we must discuss the computation of NULL.

## 2. Computation of NULL

The algorithms in the literature use a nested set construction to compute NULL:

Let  $W_0 = \emptyset$   
and for  $i \geq 0$ ,  $W_{i+1} = W_i \cup \{A \in N \mid A \rightarrow \alpha \text{ is in } P \text{ for some } \alpha \in W_i^*\}$ . We can then let  $\text{NULL} = W_{|N|}$ . cf. Harrison (1978) for a proof of the correctness of this construction.

But the nested set construction is inefficient. It may require up to  $|N|$  passes over the grammar (in Yehudai (1977) we show that this bound is achieved). So its time complexity is  $(|N| \cdot n)$  where  $n$  is the size of the grammar. If we use  $n = |G|$  as size measure then, since  $|N| \leq |V| \leq |G|$  this means  $O(n^2)$  steps. If we take  $n = \|G\|$  as measure then, using Lemma 1 we obtain

$$|N| \cdot \|G\| \leq \frac{2 \cdot \|G\|}{\log \|G\|} \cdot \|G\|$$

$$\text{and the time complexity is } O\left(\frac{n^2}{\log n}\right).$$

A question raised by the above analysis is whether or not we can do better. A close examination of the nested set construction shows that while each computation of  $W_{i+1}$  involves rescanning the entire grammar, only a small fraction of it is pertinent. Moreover, each production can only yield information about the symbols that appear in it. It appears that if we organise the information provided by the grammar in some meaningful way, scanning the grammar many times will not be required. Hunt, Szymanski and Ullman (1974) suggest the possibility of computing NULL in linear time. We now present such an algorithm. First we discuss the data structures used by the algorithm in some detail.  $W$  and  $U$  are both sets.  $W$  is used to collect elements known to be in NULL (elements are added but never removed from  $W$ ). The use of  $U$  will become clear later. When the grammar is read, a symbol  $A$  is entered in both  $W$  and  $U$  whenever a production  $A \rightarrow \alpha$  is encountered (and provided  $A$  is not yet in NULL). For each  $X \in V$ ,  $\text{POS}(X)$  is a multiset (i.e. analogous to a set but elements may appear more than once.)

Elements of  $\text{POS}(X)$  are productions in  $P$ . In particular when a production  $A \rightarrow \alpha$  is read in, it is entered in  $\text{POS}(X)$   $l$  times if  $B$  appears in  $\alpha$   $l$  times. This is done for all  $X \in V$ . The information in  $\text{POS}(X)$  is later consulted in the process of 'updating'.

For each production  $A \rightarrow \alpha$  in  $P$  the integer  $\text{NONNULL}(A \rightarrow \alpha)$  denotes the number of occurrences in  $\alpha$  of symbols not yet known to generate  $A^*$ . This number is constantly updated and if and when it reaches 0, we may conclude that  $A$  can generate  $A$ . If that is not already known (i.e. if  $A$  is not yet in  $W$ ) then  $A$  is entered in  $W$  and in  $U$ .

The process of 'updating' is as follows. A symbol  $B$  is removed from  $U$ .  $B$  is now known to be in NULL (i.e. to generate  $A$ ). Therefore for each  $A \rightarrow \alpha$  in  $P$  we decrement  $\text{NONNULL}(A \rightarrow \alpha)$  by one for each occurrence of  $B$  in  $\alpha$ . To do this only  $\text{POS}(B)$  need be inspected (rather than the entire grammar): for each occurrence of a production  $A \rightarrow \alpha$  in  $\text{POS}(B)$ ,  $\text{NONNULL}(A \rightarrow \alpha)$  is decremented by one. As mentioned above we add  $A$  to  $W$  and  $U$  whenever, in the course of decrementing  $\text{NONNULL}(A \rightarrow \alpha)$ , it reaches 0 and if  $A$  is not in  $W$ . Note that  $U$  is always a subset of  $W$  containing those elements for which 'updating' was not yet done. When  $U$  becomes empty the algorithm terminates.

NULL appears as a variable in the algorithm. Just before termination it is assigned the value of  $W$ .

Next we present the algorithm.

### Algorithm 4

Input:  $G = (V, \Sigma, P, S)$

Output:  $\text{NULL} = \{A \in N \mid A \Rightarrow^* A\}$

begin

L1:  $W := \emptyset;$

$U := \emptyset;$

for all  $X \in V$  do  $\text{POS}(X) := \emptyset;$

L2: for all  $A \rightarrow \alpha$  in  $P$  do

begin

if  $\alpha = A$  then begin

if  $A \notin W$  then begin

$W := W \cup \{A\};$

$U := U \cup \{A\}$

end

end

else begin

comment  $\alpha = X_1 X_2 \dots X_k, k \geq 1,$

$X_i \in V;$

$\text{NONNULL}(A \rightarrow \alpha) := k;$

for all  $1 \leq i \leq k$  do

$\text{POS}(X_i) := \text{POS}(X_i) \cup \{A \rightarrow \alpha\}$

end

end;

L3: while  $U$  is not empty do

begin

choose  $B \in U;$

$U := U - \{B\};$

for all  $A \rightarrow \alpha$  in  $\text{POS}(B)$  do

begin

$\text{NONNULL}(A \rightarrow \alpha) := \text{NONNULL}(A \rightarrow \alpha) - 1;$

if  $\text{NONNULL}(A \rightarrow \alpha) = 0$  and  $A \notin W$

then begin

$W := W \cup \{A\};$

$U := U \cup \{A\}$

end

end

end;

L4:  $\text{NULL} := W$

end.

\*In fact  $\text{NONNULL}(A \rightarrow \alpha)$  is defined only for  $\alpha \neq A$ , but one can assume that the value is zero for  $\alpha = A$  since this value is never consulted anyway.

The statement labels L1, L2, L3 and L4 used in the algorithm designate the start of four phases in the algorithm: Initialisation (of  $W$ ,  $U$  and  $POS$ ), reading the grammar (the **for** loop), 'updating' (the **while** loop), and outputting the result.

The following example illustrates the behaviour of the algorithm.

#### Example

Let  $G = (N \cup \{a\}, \{a\}, P, A_1)$ , where  $N = \{A_1, A_2, \dots, A_n\}$  and  $P = \{A_i \rightarrow A_{i+1} \mid 1 \leq i < n\} \cup \{A_n \rightarrow A\}$ . Apply Algorithm 4.

When L2 is reached for the first time  $W = \emptyset$ ,  $U = \emptyset$  and  $POS(X)$  is empty for all  $x \in N$ . After the **for** loop at L2 is executed once (with the production  $A_1 \rightarrow A_2$ ), we obtain  $NONNULL(A_1 \rightarrow A_2) = 1$  and  $POS(A_2)$  contains the single element  $A_1 \rightarrow A_2$  (once). The **for** loop is then executed with the productions  $A_2 \rightarrow A_3, \dots, A_{n-1} \rightarrow A_n$  and finally  $A_n \rightarrow A$ .

When L3 is reached for the first time  $POS(A_i)$  contains the single element  $A_{i-1} \rightarrow A_i$  (once) for all  $i$ ,  $2 \leq i \leq n$ . Both  $W$  and  $U$  contain only  $A_n$ , and for all  $1 \leq i \leq n$   $NONNULL(A_i \rightarrow A_{i+1}) = 1$ .

The **while** loop is executed  $n$  times, and after the last time  $U = \emptyset$ ,  $W = \{A_n, A_{n-1}, \dots, A_1\}$  and  $NONNULL(A_i \rightarrow A_{i+1}) = 0$  for all  $1 \leq i < n$ .

It should be noted here that if the productions in the grammar were ordered differently, then the **while** loop may have been executed fewer times. However even in this worst ordering the computation is efficient because we only look at the 'right points' in the grammar rather than make a full scan every time.

A completely formal proof of correctness of Algorithm 4 (using the techniques of Hoare (1969)) is lengthy and rather technical. We will give a less formal argument.

We denote, for any set  $M \subseteq N$  and any string  $\alpha \in V^*$ ,  $OC(M, \alpha)$  to be the number of occurrences of symbols from  $M$  in  $\alpha$ . To avoid confusion we will use  $D$  and  $C \rightarrow \beta$  as bound elements from  $M$  and  $P$  respectively. The next lemma establishes invariant conditions for the **while** loop at L3.

#### Lemma 7

The following conditions are invariants to the **while** loop at L3 (i.e. if conditions 1-5 (below) hold at L3, and if the **while** loop is then executed once then conditions 1-5 hold after that execution).

1. For each  $D \in N$  and each  $C \rightarrow \beta$  in  $P$ ,  $OC(\{D\}, \beta) =$  the number of times  $C \rightarrow \beta$  appears in  $POS(D)$ .
2.  $U \subseteq W$
3. For each  $C \rightarrow \beta$  in  $P$ ,  $NONNULL(C \rightarrow \beta) = OC(V, \beta) - OC(W - U, \beta)$ .
4.  $W = \{C \in N \mid \exists \beta \in V^* \text{ such that } C \rightarrow \beta \text{ is in } P \text{ and } NONNULL(C \rightarrow \beta) = 0\}$
5.  $W \subseteq \{C \in N \mid C \Rightarrow A\}$

#### Proof

Assume 1-5 hold when the **while** condition is about to be executed. Also, suppose  $U \neq \emptyset$  and  $B \in U$ . Then the **while** loop will be executed. Condition 1 holds after execution of the loop since  $POS(D)$  remains unchanged for all  $D \in N$ .

Execution of the loop removes  $B$  from  $U$  and then adds zero or more elements on to both  $W$  and  $U$ . Thus  $U \subseteq W$  must remain true. Moreover the only change in  $W - U$  is the addition to it of  $B$  (before execution of the loop  $B \in U$  and  $U \subseteq W$ ,  $B$  is removed from  $U$  but not from  $W$ ). For all  $C \rightarrow \beta$  in  $P$ , execution of the loop decrements  $NONNULL(C \rightarrow \beta)$  by the number of occurrences of  $C \rightarrow \beta$  in  $POS(B)$ . By condition 1 that quantity is  $OC(\{B\}, \beta)$ . So, for all  $C \rightarrow \beta$  in  $P$  both sides of equation 3

are decremented by the same number so that condition 3 remains true.

From condition 3 and the fact that  $W - U \subseteq V$  it is clear that  $NONNULL(C \rightarrow \beta) \geq 0$  is satisfied for all  $C \rightarrow \beta$  in  $P$ . Therefore, since the loop never increments  $NONNULL(C \rightarrow \beta)$  for any  $C \rightarrow \beta$  in  $P$ , no element may leave the right hand side of equation 4 during execution of the loop. The same is true of  $W$ , the left hand side of that equation. An element  $C$  may enter the right hand side if  $NONNULL(C \rightarrow \beta)$  is decremented to zero for some  $C \rightarrow \beta$  in  $P$  so that  $C$  was not yet in the set. But whenever that happens, the if condition is satisfied and the element is placed in  $W$  (and in  $U$ ). Hence condition 4 is preserved.

Now suppose condition 5 holds just before execution of the **while** loop. Let  $C \in N$  be any element that would be placed in  $W$  during execution of the loop. Since condition 4 would hold after execution of the loop it follows that for some production  $C \rightarrow \beta$  in  $P$ ,  $NONNULL(C \rightarrow \beta)$  would be zero after execution of the loop. From the proof of 3 it follows that for that particular production  $NONNULL(C \rightarrow \beta) = OC(\{B\}, \beta)$  just before execution of the loop. By condition 3 that means  $\beta \in ((W - U) \cup \{B\}) \subseteq W^*$ . We can write  $\beta = B_1 \dots B_n$  for some  $n \geq 0$ ,  $B_i \in W$  for all  $1 \leq i \leq n$ . By condition 5 (which

holds just before execution of the loop),  $B_i \Rightarrow A$  for all  $i$ ,  $1 \leq i \leq n$ . Therefore  $C \Rightarrow \beta = B_1 \dots B_n \Rightarrow A$  so that  $C$  belongs to the right hand side of equation 5. Since  $C$  was an arbitrary element which is added to  $W$  during execution of the loop we conclude that 5 is satisfied after execution of the loop.

#### Lemma 8

Algorithm 4 correctly computes in linear time.

#### Proof

First we consider 'partial correctness' (cf. Manna, 1974).

We want to show that if the algorithm terminates then  $NULL = \{C \in N \mid C \Rightarrow A\}$ . This will follow directly from Lemma 7 and the next two claims, which deal with the parts of the algorithm before and after the **while** loop, respectively.

#### Claim 1

When L3 is reached for the first time (after execution of the initialisation and reading phases) conditions 1-5 of Lemma 7 are satisfied.

#### Proof

It is quite easy to verify that when L3 is reached for the first time condition 1 is satisfied. Also

6. For all  $C \rightarrow \beta$  in  $P$ ,  $NONNULL(C \rightarrow \beta) = OC(V, \beta)$
7.  $W = \{C \in N \mid C \rightarrow A \text{ is in } P\}$  and
8.  $U = W$ .

Then 2 follows directly from 8, 3 follows from 6 and the fact that  $W - U = \emptyset$ . From condition 6 we also obtain that  $NONNULL(C \rightarrow \beta) = 0$  if and only if  $\beta = A$ , hence using 7 we get  $W = \{C \in N \mid C \rightarrow A \text{ is in } P\} = \{C \in N \mid \text{there exist } C \rightarrow \beta \text{ in } P, NONNULL(C \rightarrow \beta) = 0\}$  and 4 follows. Condition 5 clearly holds since for all  $C \in W$ ,  $C \Rightarrow A$ .

#### Claim 2

Suppose 1-5 hold at L4, and assume  $U = \emptyset$ . Then after this line has been executed  $NULL = \{C \in N \mid C \Rightarrow A\}$ .

#### Proof

For this condition to be satisfied after execution of this line, we must have  $W = \{C \in N \mid C \Rightarrow A\}$  at L4. This will be shown to

follow from 1-5 and  $U = \emptyset$ . In particular 3 and  $U = \emptyset$  implies that for each  $C \rightarrow \beta$  in  $P$ ,  $\text{NONNULL}(C \rightarrow \beta) = \text{OC}(V, \beta) - \text{OC}(W, \beta)$ . So that  $\text{NONNULL}(C \rightarrow \beta) = 0$  if and only if  $\beta \in W^*$ . Therefore, using 4,  $W = \{C \in N \mid \exists \beta \in W^* \text{ such that } C \rightarrow \beta \text{ is in } P\}$ .

We now prove that  $W = \{C \in N \mid C \Rightarrow A\}$  by contradiction. Since 5 directly yields containment in one direction we assume, for the sake of contradiction, that  $\{C \in N \mid C \Rightarrow A\} \subsetneq W$ , and choose  $A \in \{C \in N \mid C \Rightarrow A\} - W$  such that  $A$  has a shortest derivation of  $A, A \Rightarrow A$  among all elements in this set difference. Since  $i > 0$  we can write  $A \Rightarrow B_1 B_2 \dots B_m \Rightarrow A$ . Then for each  $j, 1 \leq j \leq m, B_j \Rightarrow A$  in a derivation of length less than  $i$ . By the minimality of  $i$  none of the  $B_j$ 's can be in  $\{C \in N \mid C \Rightarrow A\} - W$ . But all the nonterminals among the  $B_j$ 's must belong to  $\{C \in N \mid C \Rightarrow A\}$  and therefore also to  $W$ . Then  $B_1 \dots B_m \in W^*$  and  $A \in \{C \in N \mid \exists \beta \in W^* \text{ such that } C \rightarrow \beta \text{ is in } P\} = W$ . This contradiction completes the proof of the claim.

We can observe that every element of  $N$  can be removed from  $U$  at most once (since an element is entered in  $W$  and  $U$  only when it is not already there, and nothing is ever removed from  $W$ ). Therefore the while loop is executed at most  $|N|$  times. This immediately proves termination and hence total correctness (cf. Manna, 1974).

Before we can compute the time complexity of the algorithm we must specify the implementation of some data objects. We use an array of bits to implement  $W$  so that membership may be checked in constant time.\*  $U$  is implemented as a stack so choosing an element takes constant time. For each  $D \in N$

\*When a uniform cost criterion is used, array indexing takes constant time (Cf. Aho, Hopcroft and Ullman, 1974).

## References

- AHO, A. V., HOPCROFT, J. E. and ULLMAN, J. D. (1974). *The Design and Analysis of Computer Algorithms*, Reading, Mass: Addison-Wesley.
- BAR-HILLEL, Y., PERLES, M. and SHAMIR, E. (1962). On Formal Properties of Simple Phrase Structure Grammars, *Zeitschrift für Phonetik, Sprachwissenschaft, und Kommunikationsforschung*, Vol. 14, pp. 143-172.
- GRAHAM, S. L. (1974). On Bounded Right Context Languages and Grammars, *SIAM Journal of Computing*, Vol. 3, pp. 224-254.
- HARRISON, M. A. (1978). *Introduction to Formal Language Theory*, Reading, Mass: Addison-Wesley.
- HOARE, C. A. R. (1969). An Axiomatic Basis for Computer Programming, *CACM*, Vol. 12, pp. 576-581.
- HOPCROFT, J. E. and ULLMAN, J. D. (1969). *Formal Languages and Their Relation to Automata*, Reading, Mass: Addison-Wesley.
- HUNT, H. B. III, ROSENKRANTZ, D. J. and SZYMANSKI, T. G. (1976). On The Equivalence, Containment and Covering Problems for the Regular and Context Free Languages, *Journal of Computer and System Sciences*, Vol. 12, pp. 222-268.
- HUNT, H. B. III, SZYMANSKI, T. G. and ULLMAN, J. D. (1974). Operations on Sparse Relations and Efficient Algorithms for Grammar Problems, Conference Record of IEEE 15th Annual Symposium on Switching and Automata Theory, New Orleans, Louisiana.
- HUNT, H. B. III, SZYMANSKI, T. G. and ULLMAN, J. D. (1975). On the Complexity of LR(k) Testing, *ACM*, Vol. 18, pp. 707-716.
- MANNA, Z. (1974). *Mathematical Theory of Computation*, New York: McGraw-Hill.
- ROSENKRANTZ, D. J. and STEARNS, R. E. (1970). Properties of Deterministic Top-down Grammars, *Information and Control*, Vol. 17, pp. 226-255.
- YEHUDAI, A. (1977). On the Complexity of Grammar and Language Problems, Ph.D. Thesis, University of California, Berkeley.

$\text{POS}(D)$  is stored as a list, so that adding an element takes constant time and scanning the entire list requires a constant time per element.

Initialisation consists of  $|N| + 2$  operations (of assignment to  $\emptyset$ ). For every  $A \rightarrow \alpha \in P$ ,  $\text{NONNULL}(A \rightarrow \alpha)$  is once set to a value  $n = |\alpha|$  and then decremented at most  $|\alpha|$  times, and compared to 0 that many times. The number of operations involving  $\text{NONNULL}$  is therefore proportional to the size of  $G$ . The same is true for operations on  $\text{POS}$ , since every position of a nonterminal is recorded once and consulted at most once. As noted above we can have at most  $|N|$  operations of each of the following types: adding an element to  $W$ , adding an element to  $U$ , choosing and removing an element from  $U$ . Checking for membership in  $W$  can be performed at most  $2|P|$  times. In the reading phase, a check may be done for each production and one check per production can occur in the 'updating' phase. In fact one can show that only  $|P|$  operations are required. Each operation discussed takes a constant amount of time. We conclude that the time required by the algorithm is  $O(|G|)$  if we consider reading of a symbol a constant-time operation and  $O(\|G\|)$  otherwise.  $\square$

Using algorithm 4 to compute  $\text{NULL}$  we can characterise the time complexity of algorithm 3.

## Theorem

There is an algorithm that performs null rules elimination on any grammar  $G = (V, \Sigma, P, S)$  in time  $O(n \log n)$  ( $O(n)$ ) if the size measure is  $\|G\|$  ( $|G|$ ) respectively.

## Proof

Follows from lemmas 6 and 8 (since all other computations done by algorithm 3 are easy).  $\square$

A polynomial time algorithm for eliminating null rules has been independently obtained by Hunt, Rosenkrantz and Szymanski (1976). Their algorithm runs in times  $O(n^2 \log n)$  or  $O(n^2)$  depending on the size measure.