The theory of left factored languages: Part 2*

D. Wood†

† *Courant Institute, 251 Mercer Street, New York, NY 10012, U.S.A.*

In the first part of this paper‡ left-factored grammars and languages were introduced and their relevance to syntax-directed top-down analysers discussed. A number of results concerning these languages were proved and a number of problems were posed. In this second part further results are proved, further problems are posed and conclusions are reached.

(Received February 1969)

5. E languages

As Knuth (1968) has pointed out, each left-factored language is deterministic (Ginsburg and Greibach, 1966). One question that arises is what relationship do LF languages have with E languages (which we now define).

A language L, is an *E language* if $L = T(M)$ for some ϵ-free deterministic pushdown acceptor M. A *pushdown acceptor* (pda) is a 7-tuple

$$M = (Q, T, I, g, S_0, q_0, F)$$

where

(i) Q is a non-empty finite set of *states*;
(ii) T is a non-empty finite set of *input symbols*;
(iii) I is a non-empty finite set of *intermediate* (or *auxiliary*) *symbols*;
(iv) g is a mapping from $Q \times T' \times I$ to the finite subsets of $Q \times I^*(T' = T \cup \{\epsilon\}$ as before);
(v) $S_0 \in I$, is the *initial symbol*;
(vi) $q_0 \in Q$, the *start state*;
(vii) F is a subset of Q, the set of *final states*.

Initially the pda has S_0 on its pushdown tape and is in state q_0. A *move* of a pda is defined by:

$$(p, Aw, xX) \rightarrow (q, w, xy)$$

if $g(p, A, X)$ contains (q, y), where $A \in T'$, $X \in I$, $x, y \in I^*$, $p, q \in Q$, $w \in T^*$. The equivalent of a derivation in a grammar is an *acceptance* in a pda. (q, w, y) is an *acceptance* of $(p, A_1 \ldots A_k w, x)$ written $(p, A_1 \ldots A_k w, x) \Rightarrow (q, w, y)$, if there exists an acceptance sequence, $w_1, w_2, \ldots, w_{k+1}$ where

$$w_i = (p_i, A_i \ldots A_k w, x_i), w_i \rightarrow w_{i+1}$$

is a valid move, and $p_1 = p$, $x_1 = x$, $p_{k+1} = q$ and $y = x_{k+1}$. Write $(p, u, x) \overset{*}{\Rightarrow} (q, w, y)$ if either $(p, u, x) = (q, w, y)$ or $(p, u, x) \Rightarrow (q, w, y)$. A word w, is *accepted* by a pda M if

$$(q_0, w, S_0) \overset{*}{\Rightarrow} (q, \epsilon, x) \qquad \text{for some } q \in F, x \in I^*.$$

The set of all words accepted by M is denoted by $T(M)$.

An *ϵ-free deterministic pushdown acceptor* (edpda) is a pda with the following restrictions on the mapping g.

1. $g(q, \epsilon, X) = \phi$ for all $q \in Q$, $X \in I$.
2. For each $q \in Q, X \in I$, $g(q, A, X)$ contains one element for each $A \in T$.
3. $g(q, A, S_0) = \{(p, S_0 x)\}$ for each $p, q \in Q$ and $x \in I^*$.

Condition 1 says that at each move we must accept a symbol from the input tape. Condition 2 says that given a $q \in Q$, $X \in I$ and the present symbol we have only one move (i.e. deterministic). Condition 3 implies that there is always a non-empty word on the pushdown tape, so that the next move is always possible.

Lemma 5

Every S-language is an E language.

Proof:

Let G be an arbitrary S-grammar, then we can define an edpda $M = (Q, T, J, g, S_0, q, F)$ where

$$Q = \{q, d\}, F = \{d\}, J = I \cup \{X' : X \in I\}$$

and g is defined as:

(i) $g(q, A, S_0) = (q, S_0 X_n' \ldots X_1)$ for all rules $S \rightarrow AX_1 \ldots X_n$,
(ii) $g(q, A, S_0) = (d, S_0)$ for all rules $S \rightarrow A$,
(iii) $g(q, A, X) = (q, X_n \ldots X_1)$ where $X \rightarrow AX_1 \ldots X_n$,
(iv) $g(q, A, X) = (q, \epsilon)$ where $X \rightarrow A$,
(v) $g(q, A, X') = (q, X_n' X_{n-1} \ldots X_1)$ where $X \rightarrow AX_1 \ldots X_n$,
(vi) $g(q, A, X') = (d, \epsilon)$ where $X \rightarrow A$.

We only move into a final state when the present symbol A, can terminate a word, we are in state q and

* *This work was carried out in part at the University of Leeds and in part at the Courant Institute with the National Science Foundation, Grant NSF-GJ-95*

‡ *The Computer Journal, Vol.* 12, *No.* 4, *p.* 349.

the top of the pushdown tape is either S_0 or $X \in J - I$ where $S \to A$ or $X \to A$ belongs to the grammar G, S is assumed to be non-cyclic.

Let $\mathcal{S}$ be the set of S-languages, $\mathcal{L}$ the set of LF languages and $\mathcal{E}$ the set of E languages. Then we have:

Lemma 6

(i) $\mathcal{S} \subset \mathcal{E}$, (ii) $\mathcal{S} \subset \mathcal{L}$, (iii) $\mathcal{E} \not\subset \mathcal{L}$, and (iv) $\mathcal{L} \subset \mathcal{E}$.

Proof:

(i) By Lemma 5 we have shown every S-language is an E language, we give an example of an E language which is not an S-language (due to Korenjak and Hopcroft (1966)).

$$L = \{a^iba^ib: i \geqslant 1\} \cup \{a^ica^ic: i \geqslant 1\}.$$

This language does not have the power property (see Section 6) and therefore is not left factored (and thus not an S-language). The following edpda accepts it, however:

$$\begin{aligned} g(q, a, S) &= (q, SA) \\ g(q, a, A) &= (q, AA) \\ g(q, b, A) &= (p, A) \\ g(q, c, A) &= (r, A) \\ g(p, a, A) &= (p, \epsilon) \\ g(p, b, S) &= (s, S) \\ g(r, a, A) &= (r, \epsilon) \\ g(r, c, S) &= (s, S) \quad \text{where} \quad q_0 = q,\ S_0 = S \end{aligned}$$

and $F = \{s\}$.

(ii) Let $L = \{a^i: i \geqslant 0\}$, then $\epsilon \in L$ and L is not an S-language but it is an LF language.

(iii) The language given in part (i) is an E language which is not an LF language.

Rosenkrantz and Stearns (1969) give a construction for an edpda for general LL (k) languages (LF = LL (l)) (see Knuth (1968)).

Thus we obtain:

Open Problem 3:

For every LF (k) language L, $(k \geqslant 1)$ (see Appendix 2) is $L \not\#^k$ an E-language?

6. Miscellaneous results

An intermediate symbol X in a grammar G has the *prefix property* if $X \Rightarrow x = yz (z \neq \epsilon)$ implies that $X \not\Rightarrow y$, $x, y, z \in T^*$. A language has the prefix property if its sentence symbol has the prefix property.

Lemma 7

Every intermediate symbol in an S-grammar has the prefix property.

This has been proved in Korenjak and Hopcroft (1966), that the lemma does not generalise to LF grammars can be seen by the following example:

$S \to aS|\epsilon$ generates the language $\{a^i: i \geqslant 0\}$. This is obviously LF but for any string $a^n, n \geqslant 2$ there exist strings a^k, $0 \leqslant k < n$ such that $S \Rightarrow a^k$ and $S \Rightarrow a^n$. The prefix property is used in the theory of S-languages to show that particular languages are not S-languages. We define a property for LF languages which is useful in the same way.

A language L, has the *power property*, if it does not contain any infinite subsets Q and R defined by:

(i) $Q = \{uv^iw_1x_1^jz_1 = q : q \in L, u, w_1, z_1 \in T^*, v, x_1 \in T^* - \{\epsilon\}, i, j > 0\}$,

$R = \{uv^kw_2x_2^lz_2 = r : r \in L, u, w_2, x_2, z_2 \in T^*, v \in T^* - \{\epsilon\}, k, l > 0\}$, and

(ii) $Q \cap R = \phi$ where the following condition holds: for $q \in Q$, $j = M(i)$, where M is a single-valued function, e.g. $j = i$, similarly for $r \in R$, $l = N(k)$, where N is a single-valued function, and for any string $q \in Q$ there exists one string $r \in R$ such that $k = i$ and conversely.

Examples of languages which do not have the power property are:

(i) $L = \{a^iba^ib, a^ica^ic: i > 0\}$;
(ii) $L = \{a^ica^i, a^{2i}: i > 0\}$;
(iii) $L = \{a^ibc^i, a^i: i > 0\}$. Note that each of these languages is not LF. This leads to the following lemma.

Lemma 8

Every LF language has the power property.

Proof:

Assume the contrary, then there would be subsets Q and R satisfying conditions above. Since the language is LF, the left derivations for $q \in Q$ and the corresponding $r \in R$ must proceed in the same manner. Now $Q \cap R = \phi$, therefore we need to construct an LF grammar which generates *only* the strings $v^iw_1x_1^jz$, and $v^iw_2x_2^lz_2$, $i, j, l > 0$; this, however, is not possible with an LF grammar. Therefore the lemma is true.

Example (ii) above is not an S-language; it contains both $a^{2i}ca^{2i}$ and a^{2i}; or looking at it another way, both a^ica^i and $a^i\epsilon a^i$ and is therefore not LF either.

The *reverse* of a word w, written w^R is $W_n \ldots W_1$, where $w = W_1 \ldots W_n$. The *reverse* of a set of words $W = \{w\}$, W^R is $\{w^R\}$. Given two sets of words X, Y then the *product* of X and Y, written XY, is the set $\{xy: x \in X, y \in Y\}$.

Lemma 9

The set $\mathcal{L}$ of LF languages is not closed under (i) union, (ii) intersection, (iii) reversal, (iv) product, (v) complement.

Proof:

(i) Let $L_1 = \{a^ib^jcb^ja^i: i, j \geqslant 1\}$ and $L_2 = \{a^ib^ja^i: i, j \geqslant 1\}$. L_1 and L_2 are both left factored.

$L_1 \cup L_2$ is, however, not left factored; it does not have the power property.

(ii) Let $L_3 = \{a^ib^ia^j: i, j \geqslant 1\}$ and $L_4 = \{a^ib^ja^j: i, j \geqslant 1\}$. L_3 and L_4 are both LF languages, but $L_3 \cap L_4 = \{a^ib^ia^i\}$ is not even a context-free language.

(iii) Let $L_5 = \{ca^ib^ja^i: i, j \geqslant 1\}$, then $L_1 \cup L_5$ is an LF language, but $(L_1 \cup L_5)^R$ is not LF.

(iv) Let $L_6 = \{a^i b^j a^i : i, j \geqslant 0\}$, then $L_6 L_1$ can be written as

$$L_1 \cup L_2 L_1 \cup \{a^{2i} : i \geqslant 1\} L_1 \cup \{b^i : i \geqslant 1\} L_1.$$

This language does not have the power property because by (i) above $L_1 \cup L_2$ does not have the power property, therefore $L_6 L_1$ is not LF.

(v) Let $L = \{a^i b^j : 1 < i < j\}$, then L can be generated by the following LF grammar.

$$G = (\{Q, R, S\}, \{a, b\}, S, \{S \rightarrow aQbR, \; Q \rightarrow aQb|b, \; R \rightarrow bR|\epsilon\}).$$

(This is the counter example given in Rosenkrantz and Stearns (1969), however our proof that $T^* - L$ is not LF does not depend on automata theory.) Consider $L_1 = T^* - L$, then it can be considered to be the union of the following languages

$$L_1 = \{\epsilon\} \cup \{bw : w \in T^*\} \cup \{a^i b^j a w : i, j \geqslant 1, \, w \in T^*\} \cup \{a^i b^j : 1 \leqslant j \leqslant i\} \cup \{a^i : i \geqslant 1\}.$$

Now $L_1 - L_2$, where $L_2 = \{a^i b^j : 1 \leqslant j \leqslant i\}$, can easily be given an LF grammar. Let us consider L_2 in more detail. Its grammar must have a starting production $S \rightarrow XaQ$ or $S \rightarrow aQ$ since the number of a's in any terminal string are not less than the number of b's. Q must be recursive, in order that at least an equal number of a and b's are generated, therefore $Q \rightarrow aQb$. Also Q must stop (and be LF), therefore we obtain $S \rightarrow aQ, Q \rightarrow aQb|aR, R \rightarrow aR|b$. However Q is not LF. With the original production

$$S \rightarrow XaQ \text{ we obtain } Q \rightarrow aQb|b, \; X \rightarrow aX|\epsilon$$

and in this case X is not LF.

Therefore L_2 is not an LF language.

Returning to L_1 we can see that the difficulty of generating L_1 with an LF grammar is greater than that of generating L_2 by an LF grammar, since we must also generate $\{a^i : i \geqslant 1\}$ and $\{a^i b^j a w : i, j \geqslant 1, \, w \in T^*\}$, both of which begin with the symbol a. Therefore L_1 is not LF. Therefore $T^* - L$ where L is LF is not necessarily LF.

7. Discussion

We have presented a theory of left factored grammars and indicated their inherent advantages when used as a basis for top-down syntax analysers (Appendix 1). It remains, however, to relate this paper with others that have appeared recently. Korenjak and Hopcroft (1966) have discussed S-languages in some detail, these form a proper subset of the LF languages. Lewis II and Stearns (1968) have introduced LL (k) languages which are a generalisation of LL (l) languages. In Appendix 2 a generalisation of LF languages is given, these are called LF (k) languages. With an LF grammar the associated analyser needs at most one symbol to decide which of several rule alternatives to choose. Similarly with an LL (k) grammar the associated analyser needs at most k symbols to decide which of several rule alternatives to choose. Thus an LL (l) grammar is LF and vice versa. It is shown in Wood (1969b) that LL (l) grammars are the same as LF (l) grammars and that LF (k) grammars (k > 1) are always LL (k) grammars but not necessarily vice versa. Usually an analyser that is based directly on a grammar is used in conjunction with a compiler-compiler scheme (Brooker and Morris, 1962; Wood, 1968). This implies that semantic analysis also takes place; in Lewis II and Stearns (1968) it can be seen that semantic insertions into a recogniser are more natural if that recogniser is top-down. This is because, as Knuth (1968, p. 57) points out, we know what production is being used before we process its components.

Having defined LF languages we can similarly define RF (right factored) languages, it follows that if $\mathscr{L}$ = the set of LF languages, $\mathscr{R}$ = the set of RF languages, that $\mathscr{L} = \mathscr{R}^R$. Because of the left-to-right property of deterministic languages, there exist RF languages which are not deterministic. For example,

$$L = \{a^j b^j a^i c^2, \, a^j b^i a^i c : i, j \geqslant 1\}$$

is RF but not deterministic (see Ginsburg and Greibach, 1966, p. 641).

Acknowledgements

I would like to express my thanks to my former supervisor, Dr. M. Wells for his encouragement during the part of this work carried out at Leeds University. I also thank Dr. Karel Culik II who pointed out a serious error in an earlier draft of Appendix 2.

Appendix 1

A top-down analyser derived directly from a grammar

The method we describe is not claimed to be either original (in many ways it resembles Knuth's Parsing Machine (Knuth, 1968)), or unique. We think that it is simple both to construct an analyser from a given grammar and to understand its operation.

The syntax analyser is made up of a series of procedure declarations which can possibly call each other recursively; the program forming the analyser consists of these declarations together with the call of one of them. The called procedure will correspond to the sentence symbol (for example, ⟨program⟩ in ALGOL 60). Each procedure attempts to match a particular syntactic type in the input, and it returns with a value of **true** or **false** depending on whether it was successful or not. Let us take the partial syntax of an ALGOL 60 program given in Section 1 and transform it into a partial top-down analyser. We will use pseudo-ALGOL in writing down the analyser. There are a number of conditions which a grammar must meet to be amenable to the top-down analyser transformation.

1. The grammar should contain no left cycles. For example: ⟨blockhead⟩ as defined in the ALGOL 60 report is left cyclic,

⟨blockhead⟩ ::= **begin** ⟨declaration⟩|⟨blockhead⟩; ⟨declaration⟩.

The corresponding procedure would be in a position to go into an infinite recursive loop. Left cycles can be removed (Wood, 1969a).

2. The order in which the alternatives are tried is important. Consider the ALGOL 60 definition:

```
⟨unsigned number⟩ ::= ⟨decimal number⟩|
                      ⟨exponent part⟩|
                      ⟨decimal number⟩
                      ⟨exponent part⟩
```

If we did the transformation using this ordering of alternatives, the analyser would never successfully recognise a decimal number followed by an exponent part. We have to reorder the alternatives such that ⟨decimal number⟩ is textually later than ⟨decimal number⟩⟨exponent part⟩; for example

```
⟨unsigned number⟩ ::= ⟨exponent part⟩|
                      ⟨decimal number⟩
                      ⟨exponent part⟩|
                      ⟨decimal number⟩
```

A similar situation occurs with the use of the null symbol (⟨empty⟩ in the ALGOL 60 report). Consider

```
⟨actual parameter part⟩ ::= ⟨empty⟩|
                            (⟨actual parameter list⟩).
```

Now the analyser will always return the value **true** with the ⟨empty⟩ symbol, because it can always find nothing. Therefore if there were an actual parameter list it would never find it. We rewrite the definition as:

```
⟨actual parameter part⟩ ::= (⟨actual parameter list⟩)|
                            ⟨empty⟩.
```

This generalises into the rule: always place ⟨empty⟩ as the last of a number of alternatives.

```
begin comment the analyser;
boolean procedure program;
begin stack (source pointer);
  program := if block then true else
             if compound statement then true else goto
               ERROR;
  erase;
end;
boolean procedure block;
begin stack (source pointer);
  block := if unlabelled block then true else
           if label then
             if terminal (':') then
               if block then true else goto ERROR
                 else goto ERROR
             else false;
  erase;
end;
comment
  We have taken certain liberties with ALGOL 60, these
  are:
  1. an expression can contain a jump;
  2. we allow nested conditional expressions between then
     and else;
comment
  There are four system procedures, stack, erase, false and
  terminal.
  stack: preserves its actual parameter on a pushdown list.
  erase: deletes the top item in the same pushdown list.
  false: is a boolean procedure whose value is always false,
    which reinstates the value of the source pointer to be the
    top value in the pushdown list.
  terminal: is presumably a code-bodied boolean procedure
    (or it makes use of one) which examines the present
    source symbol (defined by source pointer) and com-
    pares it with the symbol in the string parameter. If
    they match then the value of terminal is true and the
    source pointer is advanced to the next source symbol,
    otherwise the value of terminal is false;
boolean procedure compound statement;
begin stack (source pointer);
  compound statement := if unlabelled compound then
                          true
    else if label then
      if terminal(':') then
        if compound statement then true else goto
          ERROR
          else goto ERROR
      else false;
  erase;
end;
comment
  We can continue with remainder of the ALGOL 60
  syntax and produce a recogniser in this way (noting, how-
  ever that some left recursion (or left cycles) would have to
  be removed, ordering of alternatives changed and syntactic
  ambiguities removed). Consider instead the consequence
  of transforming the ALGOL 60 syntax into left factored
  form: there would be no necessity to preserve the source
  pointer, therefore the procedures stack, erase and false
  would be redundant: For example, using the left factored
  partial ALGOL 60 syntax from the introduction we
  obtain;
boolean procedure program;
  program := if unlabelled program then true else
             if label then
               if terminal(':') then
                 if program then true else goto
                                  ERROR
                               else goto ERROR
                   else goto ERROR;
boolean procedure unlabelled program;
  unlabelled program := if terminal ('begin') then
                          if program tail then true else
                                               goto
                                              ERROR
                                            else false;
boolean procedure program tail;
  program tail := if compound tail then true else
                    if declaration then
                      if terminal(':') then
                        if program tail then true else
                                goto ERROR
                              else goto ERROR
                            else false;
comment and so on;
```

We should point out that ALGOL 60 is not left factored because it does not have the power property. It contains the constructs:

if BE **then** $(^nBE)^n$ **else** BE
and if BE **then** $(^nAE)^n$ **else** AE
but not if BE **then** $(^nAE)^n$ **else** BE or
if BE **then** $(^nBE)^n$ **else** AE;. This can be overcome by allowing the last two constructs syntactically but discarding them during the semantic analysis.

Appendix 2

LF(k) grammars and languages

We define a generalisation of LF grammars which we call LF(k) grammars. Basically an LF(k) grammar requires a look-ahead of k terminal symbols to decide which alternative to use next (thus LF(l) is LF). We sketch the proof of a solution to the decision problem for LF(k) grammars.

Let w/k where w is a word, and k a positive integer represent either

$$w \quad \text{if} \quad |w| \leqslant k$$

or

$$u \quad \text{if} \quad |w| > k \quad \text{and}$$
$$w = uv, \quad |u| = k.$$

If L is a set of words $\{w\}$, then $L/k = \{w/k\}$.

The *partial k left terminal set* of a word w, $k > 0$, with respect to a grammar G is:

$$\{a/k \colon w \overset{L*}{\Rightarrow} a,\ w \in IT^*,\ a \in T^*\}.$$

This is denoted by $k\colon \langle X, u\underline{w}v\rangle\ G$ or $k\colon \langle Xu\underline{w}v\rangle$ if G is understood, where $X \rightarrow uwv$ or $w \in IT^*$. We write $k\colon \langle X, w\rangle$ or simply $k\colon\langle w\rangle$ if no difficulty of interpretation exists. Informally $k\colon\langle w\rangle$ is the set of terminal strings of length at most k that can begin the terminal derivation of w.

The *k left terminal set* of a rule $X \rightarrow w$, $k > 0$, with respect to a grammar G is:

$$\{a/k \colon S \overset{*}{\Rightarrow} uXv, \quad Xv \Rightarrow wv \overset{L*}{\Rightarrow} a, \quad u, v, w \in IT^*, a \in T^*\}$$

This is denoted by $k\colon [X, w]G$ or $k\colon [X, w]$ if G is understood.

An intermediate symbol X, is *LF(k)* if for all $v, w \in A_X$, $k\colon [X, v] \cap k\colon [X, w] = \phi$, where $v \neq w$. A grammar is an *LF(k) grammar* if each $X \in I$ is LF(k). An *LF(k) language* is a language generated by an LF(k) grammar. (Note that LF(l) is equivalent to LF.)

*Theorem A*1

If a grammar G contains a left cycle, then G is not LF(k) for any k.

Proof:

This proof is very similar to the proof of Theorem 1. Any left cyclic intermediate symbol with at least two rule alternatives, at least one being left cyclic must have some terminal derivations in common. Therefore this intermediate symbol is not LF(k) for any k (for example the grammar $S \rightarrow Sa|b$).

*Lemma A*1

If a grammar G contains an intermediate symbol which has at least two nonfalse rule alternatives, then G is not *LF(k)* for any k.

Proof:

This follows immediately by the definition of the *LF(k)* property.

As for LF grammars this gives two necessary conditions for a grammar to be *LF(k)* for any k.

*Theorem A*2

Every LF(k) grammar is unambiguous.

Proof:

This requires trivial changes in the proof of Theorem 2.

Similarly Lemma A2 and Theorem A3 follow for LF(k) grammars.

We now give the LF(k) versions of Algorithms 1, 2 and 3; however we do not prove that they are correct.

*Algorithm A*1. Calculate $k\colon\langle w\rangle$

The method is based upon that used in Algorithm 1 except that instead of terminating when $1\colon\langle w\rangle$ has been bound we repeat to find $2\colon\langle w\rangle, \ldots, k\colon\langle w\rangle$. This means that we have to keep terminating nodes during a pass through Algorithm 1 (i.e. the elements $(w, X) \in D_{i+1} - U_{i+1}$), call this set K_j, defined by $K_j = K_j \cup (D_{i+1} - U_{i+1})$. We also use a count j which goes from 1 to k. To simplify the algorithm replace w by $w \neq^k$, then on termination of the algorithm replace $\neq$ by ϵ wherever it appears in $k\colon\langle w \neq^k\rangle$.

step 1: Let $j = 1$, $w = w \neq^k$, $K_0 = \{(w, \phi)\}$ and

$$T = T \cup \{\neq\}.$$

step 2: Let $D_{j1} = K_{j-1}$.

Let $w = w_1 w_2 \ldots w_m$ where each $w_i \in IT'$, then
$$U_{j1} = \{(w, X) \colon w_j \in I, \quad (w, X) \in K_{j-1}\},$$
$$K_j = \{(w, X) \colon w_j \in T', \quad (w, X) \in K_{j-1}\}$$
and $i = 1$.

step 3: If $U_{ji} = \phi$ then *step* 4 otherwise calculate:

$$D_{ji+1} = \{(w, r(Y) \cup y_j : l(Y) = y = y_1 \ldots y_j \ldots y_m\}) \colon Y \in U_{ji}, y \overset{L}{\Rightarrow} w\},$$
$$U_{ji+1} = \{(w, X) \colon w_j \in I, w_j' \notin X, (w, X) \in D_{ji+1}, \text{ if } w_j \in X \text{ then } X = (X - \{w_j\}) \cup \{w_j'\}\},$$
$$K_j = K_j \cup (D_{ji+1} - U_{ji+1})$$
increase i by one and repeat *step* 3.

step 4: If $j = k$ then *step* 5 otherwise increase j by one and repeat *steps* 2 *and* 3.

step 5: $k\colon\langle w \neq^k\rangle = \{u/k \colon (u, X) \in K_k\}$.
Therefore $k\colon\langle w\rangle$ follows by replacing any appearance of $\neq$ in $k\colon\langle w \neq^k\rangle$ by ϵ.

*Algorithm A*2. Compute $k\colon [X, w]$

$k\colon [X, w]$ can be computed in a similar manner to $[X, w]$ with, however, one main difference. Whenever the calculation of a partial k left terminal set occurs we must keep track of any terminal derivations of length less than k. In this case we must compute the follow (X) (as Knuth (1968) calls it), as we did in Algorithm 2 when we had an empty derivation. Assume that a new sentence symbol S_0, is introduced together with a production $S_0 \rightarrow S \neq^k$. On termination of the algorithm replace any appearance of $\neq$ by ϵ in $k\colon [X, w]$.

step 1: Let $n = \text{Card}(I)$, $H_1 = \{(X, a) \colon X \rightarrow w, a \in k\colon\langle X, w\rangle\}$,

$W_1 = \{a \colon (X, a) \in H_1, |a| = k\}$ and $i = 1$.

step 2: $H_{i+1} = \{(X, ba) \colon (Y, b) \in H_i, |b| < k,$
$$X \rightarrow uYw, \quad u, w \in IT^*,$$
$$a \in p\colon\langle X, uY\underline{w}\rangle, p = k - |b|\},$$

if $H_{i+1} = \phi$ then *step* 3, otherwise

$$W_{i+1} = W_i \cup \{a \colon (X, a) \in H_{i+1}, |a| = k\},$$

increase i by one, if $i \neq n + 1$ then repeat *step* 2.

step 3: If $a \neq^{j} \in W_N$, $a \in T^*$, $j \leqslant k$ (N is the value of i on transfer to *step* 3) then replace it by a and stop.

This completes the *LF(k)* versions of Algorithms 1 and 2, Algorithm A3 follows similarly.

We are interested in the following problem. Are LF(k + 1) languages also LF(k) languages, k $\geqslant$ 1? It seems appropriate to make the following definition. An *L(k) grammar* (k $\geqslant$ 0) is either

an LF(k) grammar if k > 0,
or an S-grammar if k = 0.

We first consider ϵ-free L(k + 1) grammars (k $\geqslant$ 0). This gives the following theorem.

Theorem 7

An ϵ-free L(k + 1) grammar G, (k $\geqslant$ 0) can be reduced to an equivalent L(k) grammar G_1.

Proof:

Assume, without loss of generality, that G is in normal form (by Theorem A3). If k = 0 then trivially we have an L(1) grammar is an L(0) grammar by definition. Otherwise form a partition on I as follows:

$$H = \{X: X \in I, \quad X \text{ is L(k)}\} \text{ and}$$
$$J = I - H.$$

If $J = \phi$ then $G_1 = G$ trivially, otherwise examine each intermediate symbol in J in turn. Each $X \in J$ has associated rule alternatives of the form:

$$X \rightarrow A_1x_1|A_2x_2| \ldots |A_mx_m, \text{ where}$$
$$A_i \in T,\ x_i \in I^*,\ 1 \leqslant i \leqslant m.$$

We now left factor any alternatives that begin with the same terminal symbol. For example, suppose $A_p = A_q = A_r$, $1 \leqslant p < q < r \leqslant m$, and there exists no i such that $A_i = A_p$, where $i \neq p$, $i \neq q$ and $i \neq r$. We then write

$$X \rightarrow A_pZ_1$$
$$Z_1 \rightarrow x_p|x_q|x_r$$

replacing the alternatives $X \rightarrow A_px_p|A_qx_q|A_rx_r$. Now Z_1 is L(k) because $k + 1$: $[X, A_px_p]$, $k + 1$: $[X, A_qx_q]$ and $k + 1$: $[X, A_rx_r]$ differ at the k + 1th symbol at most, therefore k: $[Z_1, x_p]$, k: $[Z_1, x_q]$ and k: $[Z_1, x_r]$ differ at most at the kth symbol. We generalise this: define a partition on A_X as follows:

$$A_X = \bigcup_{i=1}^{p} R(A_i),\ 1 \leqslant p \leqslant m,\ A_i \in T \text{ and we number the}$$

rule alternatives such that if $i \neq j$ then $A_i \neq A_j$, $1 \leqslant i, j \leqslant p$. Where we have

$$R(A_i) = \{A_qx_q: X \rightarrow A_qx_q,\ A_q = A_i,\ 1 \leqslant q \leqslant m\}.$$

Automatically the partition is disjoint. For each member of the partition carry out the left factoring as above (this is of course only necessary if Card($R(A_i)$) > 1). This produces a number of new L(k) intermediate symbols Z_i. The effect upon X is also to reduce it to L(k) because similar starting rule alternatives have been collapsed into one rule alternative. After processing each member of J we have a grammar which is L(k).

We now look at general L(k) grammars, and examine the following conjecture.

Conjecture:

An L(k + 1) grammar G(k > 0), can be reduced to an equivalent L(k) grammar G_1.

As in Theorem 7 we only need examine ϵ-normal form L(k + 1) grammars. As before we can define a partition on I.

$$H = \{X: X \in I, X \text{ is L(k)}\} \text{ and}$$
$$J = I - H.$$

We define *follow*(X), where $X \in I$, as the set

$$\{A: A \in T', \quad S \overset{*}{\Rightarrow} uXv \overset{*}{\Rightarrow} uXAw, \quad u, v\ w \in T^*\}.$$

In finding $[X, \epsilon]$ we, in fact, calculate follow(X) (see Section 4, Algorithm 2).

As in Theorem 7 we examine each intermediate symbol in turn, let the corresponding rule alternatives be:

$$X \rightarrow A_1x_1|A_2x_2|\ldots|A_mx_m|E,$$

where $A_i \in T$, $x_i \in I^*$, $E = \underline{0}$ or ϵ ($\underline{0}$ is the semi-group zero, $X \rightarrow x|\underline{0}$ or $X \rightarrow \underline{0}|x$ is the same as $X \rightarrow x$, and $x\underline{0} = \underline{0}x = \underline{0}$). We have two cases.

(i) $E = \underline{0}$, i.e. $X \not\Rightarrow \epsilon$.

This intermediate symbol can be reduced to L(k) as in Theorem 7.

(ii) $E = \epsilon$.

This can be broken down into two sub-cases.

(*a*) $\{A_i: 1 \leqslant i \leqslant m\} \cap [X, \epsilon] = \phi$.

If the set $\{A_ix_i: 1 \leqslant i \leqslant m\}$ is L(k + 1) then this can be reduced to L(k) as in Theorem 7. Follow(X) is irrelevant, because it does not share any elements with $\{A_i\}$. This can be illustrated with the following example:

$S \rightarrow qQpr|rQpc$,
$Q \rightarrow aQ|bB|\epsilon|bc$
$B \rightarrow bB|\epsilon$, where S is the sentence symbol.

This grammar is L(2) because 2: $[Q, bB] = \{bb,bp\}$ and 2: $[Q, bc] = \{bc\}$, it can be reduced to L(l) by replacing the Q productions with:

$Q \rightarrow aQ|bC|\epsilon$, $C \rightarrow c|B$.

(*b*) $\{A_i: 1 \leqslant i \leqslant m\} \cap [X, \epsilon] \neq \phi$.

Using the following example, we illustrate a possible method for carrying out the left factoring process in this case. Again S is the sentence symbol,

$S \rightarrow qQpr|rQpc$,
$Q \rightarrow aQ|pbB|\epsilon$,
$B \rightarrow bB|\epsilon$.

This grammar is L(2) because 2: $[Q, pbB] = \{pb\}$ and 2: $[Q, \epsilon] = \{pr, pc\}$. To enable left factorisation to be carried out we must let the 'pr' and 'pc' bubble upwards into the Q productions. This will give us

$S \rightarrow qQ|rQ$ and
$Q \rightarrow aQ|pbBpr|pbBpc|pr|pc$,

left factoring gives

$Q \rightarrow aQ|pC,$
$C \rightarrow bBpD|D$ and $D \rightarrow r|c.$

The grammar is now L(1).

Notes

1. We did not put pC after the rule $Q \rightarrow aQ$. Because this rule is cyclic $Q \rightarrow aQpC$ is not equivalent to $Q \rightarrow aQ$ in the original grammar.
2. In general we only want to bubble those parts of $[X, \epsilon]$ which are relevant.
3. We do not want to disturb the use of an intermediate symbol where it does not contribute to the left factoring.
4. If X is directly right cyclic any transformation we carry out must deal with this case separately, see the example above.

This leaves us with an algorithm that bears some similarity to the calculation of $[X, \epsilon]$. Before stating the algorithm we will formalise our notation. Define the set $U = \{A_i\} \cap [X, \epsilon]$ as the *head set*. By *the rules of X*, $X \in I$ we mean each production $X \rightarrow w$, $w \in A_X$. Let a *use of X* be any appearance of X in a right side of a production belonging to P, for example in $Y \rightarrow uXv$. By *the tail of a use of X* (or simply a *tail of X*) we mean that part of the right side of a production following a use of X, for example, v, is a tail of X in $Y \rightarrow uXv$. The *left of a use of X* (or simply, *the left of X*) is Y, where $Y \rightarrow uXv$.

After many false starts, taking into account notes (1) to (4) above, we arrived at the renaming algorithm which is described below.

The renaming algorithm

Assume that all elements of J that fit the other cases have been removed; this can always be done. If $J = \phi$ we have finished, otherwise for each $X \in J$ we calculate the associated head set U. The algorithm works by uniquely renaming each relevant use of X, and then incorporating the tail of X in the rules of each renamed X. For example, if a use of X is:

$$Y \rightarrow uXv, \quad \text{where} \quad [Y, v] \cap U \neq \phi$$

and $$X \rightarrow A_1x_1|A_2x_2|\ldots|A_mx_m|\epsilon,$$

then we replace the production $Y \rightarrow uXv$ by

$$Y \rightarrow uX^1$$

and add the rules

$$X^1 \rightarrow A_1x_1Z_1|A_2x_2Z_1|\ldots|A_mx_mZ_1|B_1z_1|\ldots|B_rz_r|E$$

where $$Z_1 \rightarrow B_1z_1|B_2z_2|\ldots|B_rz_r|E,$$

where $v \overset{L*}{\Rightarrow} B_1z_1$, $v \overset{L*}{\Rightarrow} B_2z_2, \ldots, A_i, B_i \in T$, $x_i, z_i \in I^*$,

$E = \underline{0}$ or ϵ. However, if X is directly right cyclic then we do not append Z_1 to those rule alternatives with which X is directly right cyclic. Let the set of the indices of these alternatives be F, then for each $i \in F$ we write

$$A_ix_i = A_iu_iX,$$

as $$A_iu_iX^1.$$

In the rule $X \rightarrow A_ix_i$, $i \in F$, x_i cannot be equal to u_iXw_i, $w_i \Rightarrow \epsilon$. This would lead to an ambiguous

grammar which cannot therefore be L(k) for any $k > 0$ (see Example 2, below). Now if $X^1 \rightarrow \epsilon$ and $[X^1, \epsilon] \cap U \neq \phi$ we repeat the algorithm with Y; this creates renamed X^1's as well as renamed Y's. If we denote this second level by a second superscript we will have

$$Y^1 \rightarrow uX^{11}v_1, \; Y^2 \rightarrow uX^{12}v_2, \ldots,$$

the v_i are then absorbed in the associated X^{1i} to give

$$Y^1 \rightarrow uX^{11}, \; Y^2 \rightarrow uX^{12}, \ldots$$

giving

$$X^{1i} \rightarrow A_1x_1Z_{1i}|\ldots|A_mx_mZ_{1i}|B_1z_1v_i|\ldots|B_rz_rv_i| \quad B_{1i}z_{1i}|\ldots|B_{r_ii}z_{r_ii}|E,$$

where

$$Z_{1i} \rightarrow B_1z_1v_i|\ldots|B_rz_rv_i|B_{1i}z_{1i}|\ldots|B_{r_ii}z_{r_ii}|E.$$

If there still remains an X^{1i} such that $X^{1i} \rightarrow \epsilon$ and $[X^{1i}, \epsilon] \cap U \neq \phi$ then we repeat the algorithm for a third time, and so on. Eventually we have a number of renamed X's, Y's, . . .; reverting to single subscripts let these be $X_1, X_2, \ldots, Y_1, Y_2, \ldots$ and so on. Each of the renamed rules is then of the form

$$X_i \rightarrow A_1x_1|A_2x_2|\ldots|A_mx_m|\epsilon$$

where $$[X_i, \epsilon] \cap \{A_i: 1 \leqslant i \leqslant m\} = \phi.$$

This reduces the problem to subcase (*a*), that is, left factoring can be carried out. Let $J = J - \{X\}$ and repeat the algorithm. Now because we know that X is L(k + 1), this algorithm must eventually find all relevant members of follow(X). By the construction we can left factor each renamed X, which reduces it to L(k). Renaming does not affect any of the other productions; it can, however, introduce many new rules. The final grammar before left factorisation takes place is also in ϵ-normal form as each renamed X is constructed to be in ϵ-normal form, as is each intermediate symbol Z_i.

The only outstanding problem with the Renaming Algorithm is: does it terminate given any L(k + 1) grammar? We give three examples, the first two were used in trying to break the Renaming Algorithm. Example 3 provided by Kurki-Suonio (1969a) is an L(k + 1) language which can never be given an L(k) grammar.

Example 1. $S \rightarrow qQpr|rQpc$
$Q \rightarrow aQ|pbB|\epsilon$
$B \rightarrow bB|\epsilon,$

as we have seen, this is an L(2) grammar.

We obtain

$$S \rightarrow qQ^1|rQ^2$$
$$Q^1 \rightarrow aQ^1|pbBZ_1|pr$$
$$Z_1 \rightarrow pr$$
$$Q^2 \rightarrow aQ^2|pbBZ_2|pc$$
$$Z_2 \rightarrow pc$$

and unchanged $Q \rightarrow aQ|pbB|\epsilon$, $B \rightarrow bB|\epsilon$; because Q is not now used, we can delete it and the rules of Q. Note that because Q is directly right cyclic we have invoked the special transformation under the renaming algorithm. The above productions can now be left factored giving finally

$$\begin{aligned}S &\rightarrow qQ_1|rQ_2\\ Q_1 &\rightarrow aQ_1|pQ_3\\ Q_3 &\rightarrow bBpr|r\\ Q_2 &\rightarrow aQ_2|pQ_4\\ Q_4 &\rightarrow bBpc|c\\ B &\rightarrow bB|\epsilon.\end{aligned}$$

This example leads to the consideration of the following rules of Q:

Example 2. $Q \rightarrow aQB|pbB|\epsilon$, where S and B are as above. This grammar which does present some problems is, however, not L(2). Since

$$\begin{aligned}Q \overset{L}{\Rightarrow} aQB \overset{L}{\Rightarrow} &aaQBB\\ \Rightarrow &aaBB\\ \Rightarrow &aabB \text{ or } aaBb,\end{aligned}$$

both lead to identical terminal strings, therefore the grammar is ambiguous, and therefore not L(k) for any k, $k > 0$. The ambiguity arises because the rule $B \rightarrow bB|\epsilon$ is not L(k) for any $k > 0$ in the above grammar. The equivalent unambiguous grammar is, however, L(2),

$$Q \rightarrow aQ|BC,\ C \rightarrow pbB|\epsilon.$$

Note that the rules $Q \rightarrow aQC|pbB|\epsilon$, $C \rightarrow b|\epsilon$ would lead to ambiguity. This means that a direct right cycle must be of the form $X \rightarrow AuX$.

Example 3. $S \rightarrow \vdash A \dashv$
$A \rightarrow aAB|\epsilon$
$B \rightarrow a^k bA|c$

for any $k \geqslant 1$ is an L(k + 1) grammar which cannot be reduced to an L(k) grammar.

Theorem 8. *The Kurki-Suonio Theorem*

There are L(k + 1) languages which do not possess an L(k) grammar, $k \geqslant 0$.

The proof of this theorem can be found in Kurki-Suonio (1969b), he uses example 3 above.

Another example of an L(k + 1) language which is not an L(k) language is provided by Rosenkrantz and Stearns (1969).

This we now give:

Example 4. $L = \{a^n(b^{k+1}x)^n : n \geqslant 1,\ x = d, b \text{ or } cc\}$. This is given by the following L(k + 1) grammar

$$\begin{aligned}S &\rightarrow aSA|aA\\ A &\rightarrow cc|b|b^{k+1}d.\end{aligned}$$

We can sum up Theorem 7 and subcase (ii, a) above with the following theorem.

Theorem 9

If the following condition holds in an L(k + 1) grammar ($k > 0$) then it can be reduced to an L(k) grammar. Condition: each L(k + 1) intermediate symbol X is either ϵ-free or follow$(X) \cap [X, x] = \phi$ for all rules $X \rightarrow x$, $x \neq \epsilon$.

Theorem 10

If $\mathscr{L}(k)$ represents the set of all L(k) languages, then there exists an infinite hierarchy of L(k) languages, i.e.

$$\mathscr{L}(0) \subset \mathscr{L}(1) \subset \mathscr{L}(2) \subset \mathscr{L}(3) \ldots$$

Thus the conjecture on the extension of Theorem 7 to general L(k) languages is untrue. It remains to ask: how far can the renaming algorithm go? This we leave as a final open problem.

References

Brooker, R. A. (1967). Top-to-bottom parsing rehabilitated?, *CACM*, Vol. 10, p. 223.

Brooker, R. A., and Morris, D. (1962). A general translation program for phrase structure language, *JACM*, Vol. 9, p. 1.

Foster, J. M. (1968). A syntax improving program, *The Computer Journal*, Vol. 11, p. 31.

Ginsburg, S., and Greibach, Sheila A. (1966). Deterministic context-free languages, *Information and Control*, Vol. 9, p. 620.

Greibach, Sheila A. (1965). A new normal-form theorem for context-free phrase structure grammars, *JACM*, Vol. 12, p. 42

Griffiths, T. V., and Petrick, S. R. (1965). On the relative efficiencies of context-free grammar recognizers, *CACM*, Vol. 8, p. 289.

Knuth, D. E. (1968). Top-down syntax analysis, International Summer School on Computer Programming, Copenhagen, Denmark 1967.

Korenjak, A. J., and Hopcroft, J. E. (1966). Simple deterministic languages, Proceedings 7th Symposium on Switching and Automata Theory, *I.E.E.E.*, p. 36.

Kurki-Suonio, R. (1968). Note presented at International Summer School on Computer Programming, Copenhagen, Denmark 1967.

Kurki-Suonio, R. (1969a). Private communication.

Kurki-Suonio, R. (1969b). Notes on top-down languages, submitted for publication.

Lewis II, P. M., and Stearns, R. E. (1968). Syntax-directed transduction, *JACM*, Vol. 15, p. 464.

Naur, P., *et al.* (1963). Revised report on the algorithmic language ALGOL 60, *The Computer Journal*, Vol. 5, p. 349.

Rosenkrantz, D. J., and Stearns, R. E. (1969). Properties of deterministic top-down grammars, presented at the ACM Symposium on the theory of computing, Marina del Rey, California, May 5, 1969.

Wood, D. (1968). On generalised interpretive schemes for programming languages, Doctoral dissertation, Leeds University.

Wood, D. (1969a). The normal form theorem—another proof, *The Computer Journal*, Vol. 12, p. 139.

Wood, D. (1969b). A note on top-down deterministic languages, to appear in *BIT*.

Woodward, P. M. (1966). A note on Foster's syntax improving device, RRE Memorandum No. 2352, Royal Radar Establishment, Malvern.