REFERENCES

1. Hamblin, C. L. Translation to and from Polish notation. *Comput. J. 5* (Oct. 1962), 210–213.
2. Allard, R. W., Wolf, K. A., and Zemlin, R. A. Some effects of the 6600 computer on language structures. *Comm. ACM 7*, 2 (Feb. 1964), 112–119.
3. Hellerman, H. Parallel processing of algebraic expressions. *IEEE Trans. EC-15*,1 (Feb. 1966), 82–90.
4. Squire, J. S. A translation algorithm fos a multiple processor computer. Proc. 18th ACM Nat. Conf., Denver, Colorado, 1963.
5. Carr, III, J. W., and Weiland, J. A nonrecursive method of syntax specification. *Comm. ACM 9*, 4 (April 1966), 267–269.
6. Revised report on the algorithmic language Algol 60. *Comm. ACM 6*,1 (Jan. 1963), 1–17; see Sec. 3.3.5, pp. 7–8.
7. FORTRAN vs. BASIC FORTRAN. *Comm. ACM 7*, 10 (Oct. 1964), 590–625; see Sec. 6.4, pp. 598–599.
8. A proposal for input-output conventions in Algol 60. *Comm. ACM 7*, 5 (May 1965), 273–283; see Sec. A.2.3.2, p. 275.

A NOTE

Top-to-Bottom Parsing Rehabilitated?

R. A. Brooker
Manchester University, Manchester, England*

This note is concerned with the efficiency of the Top-to-Bottom parsing algorithm as used in connection with programming language grammars. It is shown, for instance, that retracing of unprofitable paths can often be eliminated by a suitable rearrangement of the productions defining the grammar. The essential weakness of the method is in dealing with complicated syntactic structures which are in practice only sparsely occupied, e.g., arithmetic expressions.

The question is sometimes raised as to the relative merits of syntax analysis "top down" and "bottom up" (see, e.g., the Discussion following Leavenworth [1]). There seems to be little published evidence.

Griffiths and Petrick [2] remark (in a paper on the relative efficiencies of context-free grammar recognizers), "In this comparison we found our SBT procedure to be enormously more efficient than our STB procedure for the Lisp and Algol programming language grammars considered, and generally superior for all other grammars considered except those for which the recognizers were deterministic."

While not doubting their conclusions *for the particular grammars they considered* (although even the authors themselves admit to some discrepancy between some of their conclusions and experience obtained in the field), it is the purpose of this note to draw attention to the remarks found in Cheatham [3]: "For programming languages of the current sort, there is no clear advantage in favor of either the top-down or bottom-up analysis techniques, insofar as efficiency of the analyzer is concerned. For either technique, it is possible to design a language and syntax specification on which the technique will perform very poorly, while the other one will not be nearly so bad. The choice between the techniques is generally made on the basis of considerations other than raw speed of the analysis, ...".

Now in [2] only one grammar is presented in detail which supports the authors' conclusions. It is

$$\begin{array}{ll} F \to C & L \to L' \\ F \to S & L \to p \\ F \to P & L \to q \\ F \to U & L \to r \\ C \to U \supset U & S \to U \vee S \\ U \to (F) & S \to U \vee U \\ U \to \neg U & P \to U \wedge P \\ U \to L & P \to U \wedge U \end{array}$$

The following sentence is one which they parse w.r.t. this grammar $\neg(\neg(p' \wedge (q \vee r) \wedge p'))$. If we write the grammar in the more concise form

$$\begin{array}{l} F \to C \mid S \mid P \mid U \\ C \to U \supset U \\ U \to (F) \mid \neg U \mid L \\ L \to L' \\ L \to p \mid q \mid r \\ S \to U \vee S \mid U \vee U \\ P \to U \wedge P \mid U \wedge U \end{array}$$

it will be clear why recognizing the above sentence could involve a vast amount of retracing. Thus top-to-bottom starts by looking for a "C", which means looking for a "U", then a "(", which it does not find, then "$\neg U$", which after a fantastic search it eventually finds; *then* it returns to the C-production and looks for a "$\supset$", which *it does not find*, and so returns to the F-production and starts looking for an "S" instead, and so on.

The authors of [2] remark, "In order to determine the extent to which the disparity in efficiency between the

* Department of Computer Science

STB and the SBT procedures were due to the inclusion of the left-branching rule $L \rightarrow L'$, this rule was removed and a set of sentences not containing primes was recognised." They found this made little difference, which is not surprising in view of the activity going on at levels above. (Fiddling while Rome burns!)

In top-to-bottom practice, one would recast the grammar by replacing the F, C, S, P productions by:

$$F \rightarrow U \wedge P' \mid U \vee S' \mid U \supset U \mid U$$
$$P' \rightarrow U \wedge P' \mid U$$
$$S' \rightarrow U \vee S' \mid U$$

in which form, merging can be usefully applied thus:

$$F \rightarrow U \begin{cases} \wedge P' \\ \vee S' \\ \supset U \\ \text{NIL} \end{cases} \qquad P' \rightarrow U \begin{cases} \wedge P' \\ \text{NIL} \end{cases} \qquad S' \rightarrow U \begin{cases} \vee S' \\ \text{NIL} \end{cases}$$

With the grammar in this form, every U is recognized and the trial and error element is confined to testing whether the symbol following a U is $\wedge$, $\vee$, $\supset$, or NIL. This amounts to a methodical examination of alternatives. An extreme example is the verification of a particular letter in the production:

$$\text{LETTER} \rightarrow a \mid b \mid c \mid d \mid \cdots \mid z$$

Since little retracing is involved the selectivity device described in [2] and [4] is only of marginal utility and was in fact dropped from the TB algorithm used in the Compiler Compiler.

Another feature of that algorithm is that exploration at each level terminates with the first successful alternative (these being ordered from left to right) so that it may be necessary to arrange the alternatives in order of preference. In the above grammar for instance the L productions would have to be replaced by:

$$L \rightarrow XY \mid x$$
$$X \rightarrow p \mid q \mid r$$
$$Y \rightarrow {}'Y \mid '$$

Thus the TB algorithm on sympathetic grammars is no worse (and conceptually a good deal simpler) than the SBT. We have no experience of LISP, but the ALGOL grammar can certainly be arranged to suit TB. The main difficulty with programming grammars lies in dealing with arithmetical expressions. Whatever definition we choose an ⟨EXPR⟩ can generate a fairly complicated tree structure. Very often however, the actual instance of an ⟨EXPR⟩ is something trivial, e.g., a or 1, and in analyzing w.r.t. an ⟨EXPR⟩ we generate a tree with many empty branches. It is the time spent in this activity, and on the subsequent inspection of these empty branches in the processing routines, that makes unmodified TB approach inferior to any method (e.g., precedence analysis) which deals only with the complexity actually present.

RECEIVED OCTOBER, 1966

REFERENCES

1. LEAVENWORTH, B. M. FORTRAN IV as a syntax language. *Comm. ACM 7*, 2 (Feb. 1964), 72.
2. GRIFFITHS, T. V., AND PETRICK, S. R. On the relative efficiencies of context-free grammar recognisers. *Comm. ACM 8*, 5 (May, 1965), 289.
3. CHEATHAM, T. E., AND SATTLEY, K. Syntax-directed compiling. Proc. AFIPS 1964 Spring Joint Comput. Conf., Vol. 25, April, 1964.
4. BROOKER, R. A., AND MORRIS, D. Some proposals for the realization of a certain assembly program. *Comput. J. 3* (1961), 220.

CORRIGENDA

NUMERICAL ANALYSIS

L. W. Ehrlich, "A Modified Newton Method for Polynomials," *Comm. ACM 10*, 2 (Feb. 67), 107–108.

In formulas (4), (6), (10), (11), (28), and (29), read $j \neq i$ in place of $j \neq l$.

Read formula (21) as

$$\gamma_{s+1} = \sum_{j=1}^{s} \frac{(\lambda_{s+1} - r_{s+1})}{(\lambda_{s+1} - \lambda_j)} \frac{(r_j - \lambda_j)}{(\lambda_{s+1} - \lambda_j)} + \sum_{j=s+2}^{n} \frac{(\lambda_{s+1} - r_{s+1})}{(\lambda_{s+1} - r_j)}.$$

COMPUTER SYSTEMS

Peter Calingaert, "System Performance Evaluation: Survey and Appraisal," *Comm. ACM 10*, 1 (Jan. 1967), 12–18.

Figure 1 should have appeared as follows:

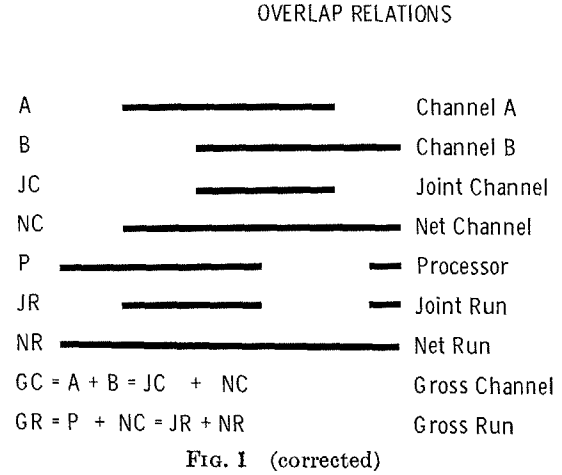

FIG. 1 (corrected)

Note that the elements for Channel B and the Processor are corrected.

CACM apologizes for the error, which was introduced in the printing process.