then was executed in the following manner. The Hollerith string of length *NCHARS*, starting with the *J*th character of *WORD*1 was scanned. If the Hollerith string of *N* characters, that is represented above by dots (and which was an external name) was found in this scan, joined by an equality symbol to a numerical field, then the contents of that field was stored as the new value of *VALUE*, and *LINK* was set to 1. Otherwise *VALUE* was left unchanged, and *LINK* was set to 0. It was permissible to interpose one or more fields of the form "... =" between the equality symbol, following the external name, and the numerical value in the input, where any Hollerith string not containing the equality symbol and not starting with a signed or unsigned digit could appear in place of the dots. Thus the input

$$K1 = K2 = K3 = 7$$

was recognized by CVNOFT as equivalent to $K1 = 7$ and $K2 = 7$ and $K3 = 7$. The form of conversion (integer or floating-point) depended only on the presence or absence of a decimal point in the input.

The original version of CVNOFT, which was used for some four years, has not been distributed, as a shorter version now can be constructed using IGD, FLD and a simple FORTRAN coded scanning routine.

Acknowledgments. The authors would like to thank Dr. J. W. Moskowitz for the benefit of helpful discussion. The work that is reported here was supported in part by grant GM 10-430 of the National Institutes of Health.

REFERENCES

1. RAMSAY, W. B. Input (BCD, OCTAL, or DECIMAL) under sense light control. SHARE Distrib. No. 1025.
2. LUBECK, S. P., AND SMITH, R. B. General symbolic input routine (FORTRAN). SHARE Distrib. No. 1294.
3. MITCHELL, M. F. Flexible decimal and alphabetic input routine for FORTRAN II. SHARE Distrib. No. 1469.
4. HASSIT, A. Format-free input allowing the use of the FORTRAN list statement. SHARE Distrib. No. 1473.
5. YARBOROUGH, L. D. Input data organization in FORTRAN. *Comm. ACM, 5* (1962), 508.
6. BARNETT, M. P. The evaluation of molecular integrals by the zeta function method. *Methods of Computational Physics, Vol. II*, p. 148, Academic Press, 1963.
7. BAILEY, M. J. A free format input routine to read FORTRAN variables and specified portions of arrays. Cooperative Computing Lab. Tech. Note No. 30, MIT, 1963, SHARE Distrib. 1432.
8. FUTRELLE, R. P., AND BARNETT, M. P. A FORTRAN subroutine to convert input data of arbitrary format. Solid State and Molecular Theory Group, Programming Note No. 22, MIT, 1958.
9. BARNETT, M. P., AND FUTRELLE, R. P. Syntactic analysis by digital computer. *Comm. ACM 5* (1962), 515, also SHARE Distrib. 1401.
10. BAILEY, M. J., BURLESON, P. B., CARTER, E. J. D., AND KELLEY, K. L. A BCD manipulation package for use within FORTRAN. Cooperative Computing Lab. Tech. Note No. 12, MIT, 1962; also SHARE Distrib. 1371.
11. BARNETT, M. P. Low level language subroutines for Use FORTRAN. *Comm. ACM 4* (1962), 492.

Variable Width Stacks

NAOMI ROTENBERG AND ASCHER OPLER
Computer Usage Company, Inc., New York, N. Y.

Character addressable, variable field computers permit ready establishment and manipulation of variable width stacks. Single machine commands may push variable field items down into such stacks or pop them up. The availability of a variety of field delimiters allows the machine to push down or pop up more than one variable width item with one command. Since these stacking operations can be made the basis of compiler decoding algorithms, the proper use of machines of this class for compilation has advantages over machines with fixed-length words.

With the increased usage of character addressable computers and the dominance of algebraic compilers in scientific computing, many compilers are being written for this type of computer. Since the majority of compilers have been constructed for fixed-word length machines, a belief that character addressable, variable length field machines are inferior for compiling purposes has developed. It is the intention of this report to indicate how the logical design of such computers can be used to advantage in compiling algebraic expressions.

The principal characteristics of these machines are the assignment of an address to every single character in memory and the ability to manipulate contiguous character fields by addressing one character and having the operation extend to successive characters until a field terminator is encountered. Several such machines allow the field to extend either to the right or to the left of the addressed character. Furthermore, the field size may be controlled by a choice of several termination mechanisms.

During the process of compilation the elements of the source language to be manipulated are usually variable. A reference to the source language manual for most procedure-oriented systems will verify this. For most systems the manipulation required as part of the statement decomposition exchanges variable length elements for fixed-length elements or pads these to a fixed-length equal to either the machine word size or to the size of the longest allowable element.

Using the natural features of a character addressable machine, it is possible to establish and manipulate variable width pushdown lists which then obviates the necessity of either substitution or expansion of language elements. To demonstrate the principle, we describe an exceedingly simple method for scanning algebraic expressions. The scheme makes use of three pushdown stacks. One of these is a variable width stack and the other two are character stacks. A variable width stack is a sequential list of items which are stored in a first-in, last-out manner, and each item is variable in the number of characters it contains.

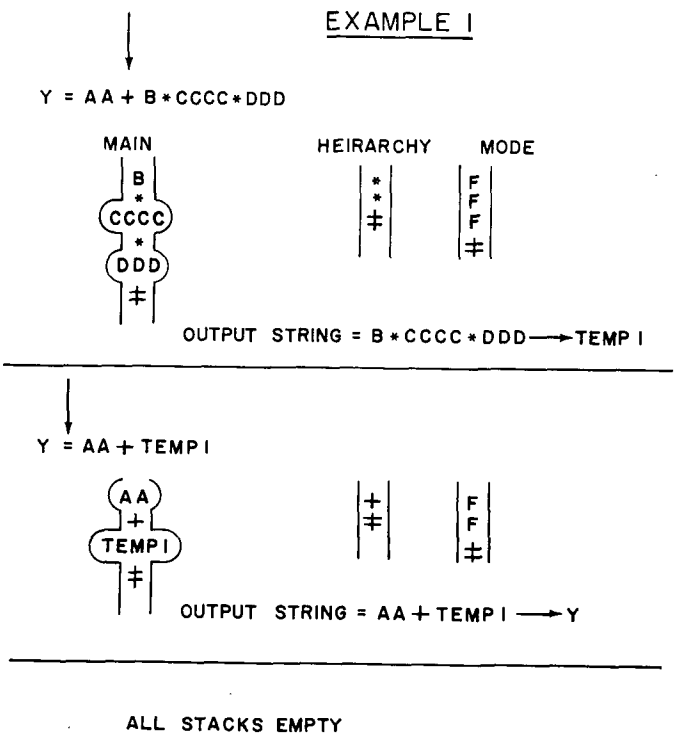
A character stack has the same general definition except that each item consists of only one character.

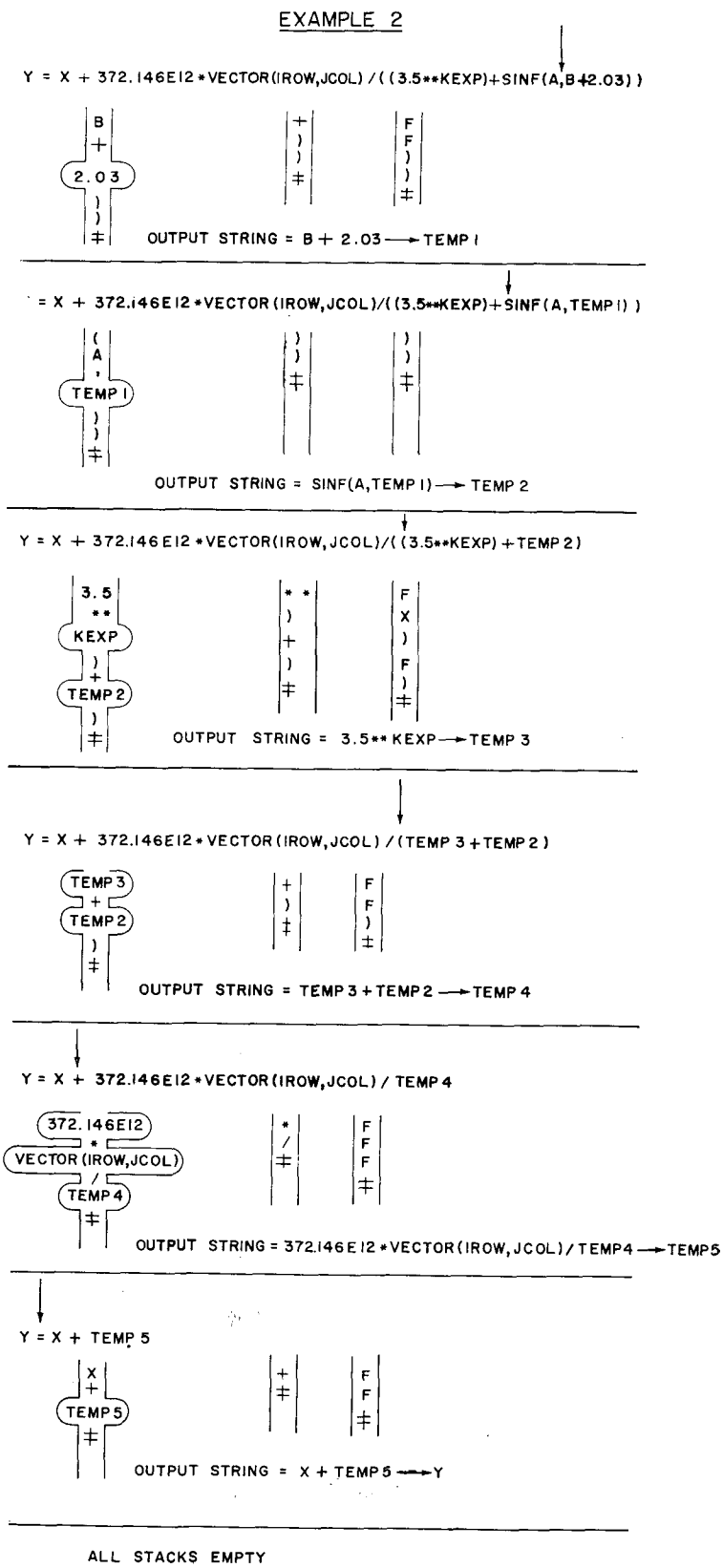
The algebraic scan is carried out in a single right-to-left sweep. Prior to this scan, a single left-to-right sweep validity checks and edits the expression. The decomposition applies the rules as given by FORTRAN. That is, it recognizes the role of nested parentheses, operator hierarchy and the left-to-right grouping of nonparenthesized operators of equal hierarchy. Functions and subscripts are allowed in the expression according to the FORTRAN rules.

The edit scan that precedes the decomposition examines each character and identifies and flags each operand and each operator. In addition, significant separators including commas, parentheses and equal signs are identified and, in certain cases, replaced with an appropriate field termination sign.

The method of decomposition is as follows. An index sweeps across the edited expression item-by-item and, according to the algorithm stated below, it either pushes down the item into the variable width stack or pops up items from the stack and delivers them to a post processor in the form of a string. The character stacks serve auxiliary functions in the scanning process. The hierarchy stack is used to push down and pop up operators as they are encountered in the scan. The mode stack is used to store a symbol which corresponds to the mode of the operand as it is pushed down into the main stack. The use of the auxiliary stacks is described below.

The control of all three stacks is parallel—whenever one stack is pushed down, all stacks are pushed down; whenever one stack is popped up, all stacks are popped up. Not only are the directions (up or down) parallel, but the control of the length of the sequence pushed or popped





is exercised in parallel according to the location in the stack of the corresponding delimiter.

The production of coding from a variable length string (e.g., $A + B + C + D + E \rightarrow Y$) can produce more efficient code than that produced from sequences of triples (e.g. $A + B \rightarrow T_1$; $T_1 + C \rightarrow T_2$; $T_2 + D \rightarrow T_3$; $T_3 + E \rightarrow Y$) unless a separate optimizing pass is performed.

To illustrate the use of the variable width stack, two examples are given. The first is rather trivial but should familiarize the reader with the general technique. The second example illustrates the handling of some practical problems.

Decomposition Algorithm

An index moves from right to left examining each field:

1. If the field is an operand, it is placed in the main stack and its mode is placed in the mode stack.
2. If the field is a delimiter, action is taken as follows:
   a. If it is a terminator, it is placed in all three stacks.
   b. If it is a right parenthesis, it is placed in all three stacks.
   c. If it is a comma used to separate the arguments of a function, it is placed in all three stacks.
   d. If it is a left parenthesis, the stack is cleared to the first right parenthesis encountered.
   e. If it is an "equal" sign, the stack is cleared.
3. If the field is an operator, either it is placed in the main stack and in the hierarchy stack or it causes the issuance of strings to the post processor.
   a. If the operator standing at the top of the hierarchical stack has greater power than the encountered operator, then the operands and operators in the main stack are issued to the post processor down to the first operator of equal or lower power.
   b. If an operator appears between commas or between a comma and a right parenthesis (i. e., if arithmetic is performed within an argument of a function), then when the next comma or function left parenthesis is encountered, the main stack is popped up to produce strings for the post processor down to the next comma or right parenthesis.
   c. As coding is issued from the main stack under control of the hierarchy stack, the mode stack output is checked for consistency according to the FORTRAN rules. This output is also used for selection of the proper exponentiation sub-routine entry.
4. All strings that are produced except the final one are of the general form $e_1\ op_1\ e_2\ op_2\ e_3\ \cdots \rightarrow$ temporary. The name of the temporary field replaces the issued subexpression in the stack.

Summary

We have found that performing the decomposition of statements written in source languages can be greatly facilitated by taking proper advantage of the natural features of character addressable computers. Manipulation of elements of the source language as variable width fields that may be scanned, stacked and manipulated as code producing strings saves both space and time in compilation. Our experience has verified that extremely rapid processors can be constructed developing these techniques.

An Experiment in Automatic Verification of Programs

G. M. Weinberg and G. L. Gressett*
IBM Systems Research Institute, New York, N.Y.

How effective is a compiler at replacing explicit verification, and what is the cost of this technique?

With present-day techniques, the process of programming usually involves a step of transcription of handwritten programs into some machineable form, such as cards or paper tape. Such a transcription has a finite probability of error; and therefore validation techniques are often used, the two most common ones being double-keying (as in key-verifying cards) and sight reading against the original documents. Key verifying is generally acknowledged to be the more reliable method, but its greater direct expense accounts for the frequent use of sight verifying techniques.

In the transcription of programs, the data (written instructions) will ordinarily be subject to a number of examinations by the originator (programmer) and correspondingly to a number of opportunities for making (expected) corrections. In addition, the instructions are ordinarily scrutinized by a processor (compiler) which has edit checks built into it for catching nontranscription errors but which incidentally catch a certain proportion of transcription errors as a byproduct. As a consequence, we find a number of programmers and installations who have decided to bypass any explicit verification procedure on the transcription of their programs. If this technique is at least as effective as either of the other two verification procedures, it may be that many other programmers will wish to adopt it.

Actually, the motivation for the study leading to this paper came from the consideration of a somewhat different problem: the design of a low-cost terminal (one of multiple-terminals) for remote operation of a large computer system. In this case, the transcription device also serves as a transmission terminal, so that the addition of some key-verification capability or provision for sight verification and correction could add considerably to both the initial equipment cost and the unit production costs.

An equally important consideration in such remote operations is the probability of a meaningful test run at an early trial. The frustration of having programs returned without execution because of some trivial error is magnified when the user is far away and feels a lack of control.

For either purpose, the same questions have to be answered: How effective is a compiler at replacing explicit verification and what is the cost of this technique?

---

* Parts of this work were done in fulfillment of requirements for graduation from the IBM Systems Research Institute.