A COMMON LANGUAGE FOR HARDWARE, SOFTWARE, AND APPLICATIONS

Kenneth E. Iverson
Thomas J. Watson Research Center, IBM
Yorktown Heights, New York

INTRODUCTION

Algorithms commonly used in automatic data processing are, when considered in terms of the sequence of individual physical operations actually executed, incredibly complex. Such algorithms are normally made amenable to human comprehension and analysis by expressing them in a more compact and abstract form which suppresses systematic detail. This suppression of detail commonly occurs in several fairly well defined stages, providing a hierarchy of distinct descriptions of the algorithm at different levels of detail. For example, an algorithm expressed in the FORTRAN language may be transformed by a compiler to a machine code description at a greater level of detail which is in turn transformed by the "hardware" of the computer into the detailed algorithm actually executed.

Distinct and independent languages have commonly been developed for the various levels used. For example, the operations and syntax of the FORTRAN language show little semblance to the operations and syntax of the computer code into which it is translated, and neither FORTRAN nor the machine language resemble the circuit diagrams and other descriptors of the processes eventually executed by the machine. There are, nevertheless, compelling reasons for attempting to use a single "universal" language applicable to all levels, or at least a single core language which may be slightly augmented in different ways at the various levels.

First, it is difficult, and perhaps undesirable, to make a precise separation into a small number of levels. For example, the programmer or analyst operating at the highest (least detailed) level, may find it convenient or necessary to revert to a lower level to attain greater efficiency in eventual execution or to employ convenient operations not available at the higher level. Programming languages such as FORTRAN commonly permit the use of lower levels, frequently of a lower level "assembly language" and of the underlying machine language. However, the employment of disparate languages on the various levels clearly complicates their use in this manner.

Second, it is not even possible to make a clear separation of level between the software (metaprograms which transform the higher level algorithms) and the hardware, since the hardware circuits may incorporate permanent or semipermanent memory which determines its action and hence the computer language. If this special memory can itself be changed by the execution of a program, its action may be considered that of software, but if the memory is "read-only" its action is that of hardware—leaving a rather tenuous distinction between software and hardware which is likely to be further blurred in the future.

Finally, in the design of a data processing system it is imperative to maintain close
communication between the programmers (i.e., the ultimate users), the software designers, and the hardware designers, not to mention the communication required among the various groups within any one of these levels. In particular, it is desirable to be able to describe the metaprograms of the software and the microprograms of the hardware in a common language accessible to all.

The language presented in Reference 1 shows promise as a universal language, and the present paper is devoted to illustrating its use at a variety of levels, from microprograms, through metaprograms, to "applications" programs in a variety of areas. To keep the treatment within reasonable bounds, much of the illustration will be limited to reference to other published material. For the same reason the presentation of the language itself will be limited to a summary of that portion required for microprogramming (Table 1), augmented by brief definitions of further operations as required.

MICROPROGRAMS

In the so-called "systems design" of a computer it is perhaps best to describe the computer at a level suited to the machine language programmer. This type of description has been explored in detail for a single machine (the IBM 7090) in Reference 1, and more briefly in Reference 2. Attention will therefore be restricted to problems of description on a more detailed level, to specialized equipment such as associative memory, and to the practical problem of keying and printing microprograms which arise from their use in design automation and simulation.

The need for extending the detail in a microprogram may arise from restrictions on the operations permitted (e.g., logical or and negation, but not and), from restrictions on the data paths provided, and from a need to specify the overall "control" circuits which (by controlling the data paths) determine the sequence in the microprogram, to name but a few. For example, the basic "instruction fetch" operation of fetching from a memory (i.e., a logical matrix) M the word (i.e., row) \( M^i \) selected according to the base two value of the instruction location register \( s \) (that is, \( i = \langle s \rangle \)), and transferring it to the command register \( c \), may be described as

\[
c \leftarrow M^{\langle s \rangle}.
\]

Suppose, however, that the base two value operation (i.e., address decoding) is not provided on the register \( s \) directly, but only on a special register \( q \) to which \( s \) may be transferred. The fetch then becomes

\[
a \leftarrow s, \\
c \leftarrow M^{\langle q \rangle}.
\]

Suppose, moreover, that all communication with memory must pass through a buffer register \( b \), that each transfer out of a memory word \( M^i \) is accompanied by a subsequent reset to zero of that word (that is, \( M^i \leftarrow \epsilon \)), that every transfer from a register (or word of memory) \( x \) to a register (or word of memory) \( y \) must be of the form

\[
y \leftarrow x 
\]

and that any register may be reset to zero, then the instruction fetch becomes

\[
\begin{align*}
1 & \quad a \leftarrow \epsilon \\
2 & \quad a \leftarrow s \lor a \\
3 & \quad b \leftarrow \epsilon \\
4 & \quad b \leftarrow M^{\langle q \rangle} \lor b \\
5 & \quad M^{\langle q \rangle} \leftarrow \epsilon \\
6 & \quad M^{\langle q \rangle} \leftarrow b \lor M^{\langle q \rangle} \\
7 & \quad c \leftarrow \epsilon \\
8 & \quad c \leftarrow b \lor c.
\end{align*}
\]

In this final form, the successive statements correspond directly (except for the bracketed pair 4 and 5 which together comprise an indivisible operation) to individual register-to-register transfers. Each statement can, in fact, be taken as the "name" of the corresponding set of data gates, and the overall control circuits need only cycle through a set of states which activate the data gates in the sequence indicated.

The sequence indicated in a microprogram such as the above is more restrictive than necessary and certain of the statements (such as 1 and 3 or 6 and 7) could be executed...
concurrently without altering the overall result. Such overlapping is normally employed to increase the speed of execution of microprograms. The corresponding relaxation of sequence constraints complicates their specification, e.g., execution of statement \( k \) might be permitted to begin as soon as statements \( h, i \) and \( j \) were completed. Sonzig (Reference 3) proposes some useful techniques and conventions for this purpose.

The "tag" portion of an associative memory can, as shown in Reference 2, be characterized as a memory \( M \), an argument vector \( x \), and a sense vector \( s \) related by the expression

\[
s = M \hat{x}
\]
or by the equivalent expression

\[
s = \overline{M} \vee \overline{x},
\]

obtained by applying De Morgan's law. Falkoff (Reference 4) has used microprograms of the type discussed here in a systematic exploration of schemes for the realization of associative memories for a variety of functions including exact match, largest tag, and nearest larger tag.

Because the symbols used in the language have been chosen for their mnemonic properties rather than for compatibility with the character sets of existing keyboards and printers, transliteration is required in entering microprograms into a computer, perhaps for processing by a simulation or design automation metaprogram. For that portion of the language which is required in microprogramming, Reference 5 provides a simple and mnemonic solution of the transliteration problem. It is based upon a two-character representation of each symbol in which the second character need be specified but rarely. Moreover, it provides a simple representation of the index structure (permitting subscripts and superscripts to an arbitrary number of levels) based upon a Lukasiewicz or parenthesis-free representation of the corresponding tree.

**METAPROGRAMS**

Just as a microprogram description of a computer (couched at a suitable level) can provide a clear specification of the corresponding computer language, so can a program description of a compiler or other metaprogram give a clear specification of the "macro-language" which it accepts as input. No complete description of a compiler expressed in the present language has been published, but several aspects have been treated. Brooks and Iverson (Chapter 8, Reference 6) treat the SOAP assembler in some detail, particularly the use of the open addressing system and "availability" indicators in the construction and use of symbol tables, and also treat the problem of generators. Reference 1 treats the analysis of compound statements in compilers, including the optimization of a parenthesis-free statement and the translations between parenthesis and parenthesis-free forms. The latter has also been treated (using an emasculated form of the language) by Oettinger (Reference 7) and by Huzino (Reference 8); it will also be used for illustration here.

Consider a vector \( e \) representing a compound statement in complete* parenthesis form employing operators drawn from a set \( \{ +, \times, -, \} \). Then Program 1 shows an algorithm for translating any well-formed statement \( e \) into the equivalent state­ment \( \ell \) in parenthesis-free (i.e., Lukasiewicz) form.

**Program 1** - The components of \( e \) are examined, and deleted from \( e \), in order from left to right (steps 4, 5). According to the decisions on steps 6, 7, and 8, each component is discarded if it is a left parenthesis, appended at the head of the resulting vector \( \ell \) if it is a variable (step 9), appended at the head of the auxiliary stack vector \( s \) if it is an operator (step 10), and initiates a transfer of the leading component of the stack \( s \) to the head of the result \( \ell \) if it is a right parenthesis (steps 11, 12). The behavior is perhaps best appreciated by tracing the program for a given case, e.g., if \( e = (\{ \{ x, +, y \}, x, \{ p, +, a \} \} \} \}, \) then \( \ell = (x, +, p, +, y, x) \).

**APPLICATIONS**

Areas in which the programming language has been applied include search and sorting procedures, symbolic logic, linear programming, information retrieval, and music.

---

*In complete parenthesis form all implied parentheses are explicitly included, e.g., the statement \((x + y)\times(p + q)\) is represented by \( e = \{ \{ x, +, y \}, x, \{ p, +, q \} \} \} \} \} \}.

†A branch is followed if the relation with which it is labelled holds between the quantities separated by the colon. The relation "\( e \)" denotes set membership.
## Table 1

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Definition</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalar</td>
<td>$a$</td>
<td>$a = (a_0, a_1, \ldots, a_{n-1})$</td>
<td>$a = (1, 0, 1, 0, 1)$, $a = (1, 0, 1)$</td>
</tr>
<tr>
<td>Vector</td>
<td>$A$</td>
<td>$A^j = (A_0^j, \ldots, A_{n-1}^j)$</td>
<td>$A = \begin{pmatrix} 0 &amp; 1 &amp; 2 &amp; 3 &amp; 4 \ 0 &amp; 1 &amp; 2 &amp; 3 &amp; 4 \ 0 &amp; 1 &amp; 2 &amp; 3 &amp; 4 \end{pmatrix}$, $B = \begin{pmatrix} 0 &amp; 1 &amp; 1 \bigtriangleup 0 &amp; 1 \ 0 &amp; 1 &amp; 1 \bigtriangleup 0 &amp; 1 \end{pmatrix}$</td>
</tr>
<tr>
<td>Matrix</td>
<td>$A$</td>
<td>$A^j = (A_0^j, \ldots, A_{n-1}^j)$</td>
<td>$A = \begin{pmatrix} 0 &amp; 1 &amp; 2 &amp; 3 &amp; 4 \ 0 &amp; 1 &amp; 2 &amp; 3 &amp; 4 \ 0 &amp; 1 &amp; 2 &amp; 3 &amp; 4 \end{pmatrix}$, $B = \begin{pmatrix} 0 &amp; 1 &amp; 1 \bigtriangleup 0 &amp; 1 \ 0 &amp; 1 &amp; 1 \bigtriangleup 0 &amp; 1 \end{pmatrix}$</td>
</tr>
<tr>
<td>Floor</td>
<td>$k \downarrow [x]$</td>
<td>$0 \leq k \leq x$</td>
<td>$k \downarrow [3] = 3$, $[3, 14] = 3$</td>
</tr>
<tr>
<td>Ceiling</td>
<td>$k \uparrow [x]$</td>
<td>$k \leq x \leq k+1$</td>
<td>$k \uparrow [3] = 3$, $[3, 14] = 3$</td>
</tr>
<tr>
<td>Residue</td>
<td>$k = m \mod n$</td>
<td>$k = m \mod n$</td>
<td>$k = 3 \mod 5 = 3$, $k = 6 \mod 5 = 1$</td>
</tr>
<tr>
<td>And</td>
<td>$w = u \land v$</td>
<td>$w = 1$ if $u = 1$ and $v = 1$</td>
<td>$w = 1$ if $u = 1$ and $v = 1$</td>
</tr>
<tr>
<td>Or</td>
<td>$w = u \lor v$</td>
<td>$w = 1$ if $u = 1$ or $v = 1$</td>
<td>$w = 1$ if $u = 1$ or $v = 1$</td>
</tr>
<tr>
<td>Negation</td>
<td>$w = \overline{u}$</td>
<td>$w = 1$ if $u = 0$</td>
<td>$w = 1$ if $u = 0$</td>
</tr>
<tr>
<td>Proposition</td>
<td>$w = (u \leq v)$</td>
<td>$w = 1$ if $u \leq v$</td>
<td>$w = 1$ if $u \leq v$</td>
</tr>
<tr>
<td>Full vector</td>
<td>$w = \varepsilon(n)$</td>
<td>$\varepsilon_i = 1$ (All 1's)</td>
<td>$\varepsilon(3) = (1, 1, 1, 1, 1)$, $\varepsilon = \text{zero vector}$</td>
</tr>
<tr>
<td>Unit vector</td>
<td>$w = \mathbf{1}(n)$</td>
<td>$\mathbf{1}_i = 1$ (1 in position $i$)</td>
<td>$\mathbf{1}(3) = (1, 0, 0, 0, 0), \mathbf{1}(5) = (0, 0, 0, 0, 0)$</td>
</tr>
<tr>
<td>Prefix vector</td>
<td>$w = \mathbf{1}(n)$</td>
<td>$\mathbf{1}_i = 1$ (1 in position $i$)</td>
<td>$\mathbf{1}(3) = (1, 1, 1, 1, 1)$, $\mathbf{1}(5) = (1, 1, 1, 1, 1)$</td>
</tr>
<tr>
<td>Suffix vector</td>
<td>$w = \mathbf{1}(n)$</td>
<td>$\mathbf{1}_i = 1$ (1 in position $i$)</td>
<td>$\mathbf{1}(3) = (1, 1, 1, 1, 1)$, $\mathbf{1}(5) = (1, 1, 1, 1, 1)$</td>
</tr>
<tr>
<td>Infix vector</td>
<td>$w = \mathbf{1}(n)$</td>
<td>$\mathbf{1}_i = 1$ (1 in position $i$)</td>
<td>$\mathbf{1}(3) = (1, 1, 1, 1, 1)$, $\mathbf{1}(5) = (1, 1, 1, 1, 1)$</td>
</tr>
<tr>
<td>Interval vector</td>
<td>$w = \mathbf{1}(n)$</td>
<td>$\mathbf{1}_i = 1$ (1 in position $i$)</td>
<td>$\mathbf{1}(3) = (1, 1, 1, 1, 1)$, $\mathbf{1}(5) = (1, 1, 1, 1, 1)$</td>
</tr>
<tr>
<td>Full matrix</td>
<td>$w = \mathbf{1}(m \times n)$</td>
<td>$\mathbf{1}_i = 1$ (1 in position $i$)</td>
<td>$\mathbf{1}(3) = (1, 1, 1, 1, 1)$, $\mathbf{1}(5) = (1, 1, 1, 1, 1)$</td>
</tr>
<tr>
<td>Identity matrix</td>
<td>$w = \mathbf{1}(n)$</td>
<td>$\mathbf{1}_i = 1$ (Diagonal 1's)</td>
<td>$\mathbf{1}(3) = (1, 1, 1, 1, 1)$, $\mathbf{1}(5) = (1, 1, 1, 1, 1)$</td>
</tr>
</tbody>
</table>

From the collection of the Computer History Museum (www.computerhistory.org)
<table>
<thead>
<tr>
<th>Operation</th>
<th>Formula</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression</td>
<td>( z \leftarrow \tilde{u}_x )</td>
<td>( \tilde{u}_x ) obtained from ( x ) by suppressing each ( x_i ) for which ( \tilde{u}_i = 0 )</td>
</tr>
<tr>
<td>Row compression</td>
<td>( z \leftarrow u/x )</td>
<td>( z^1 = u^1/\tilde{x}_1 )</td>
</tr>
<tr>
<td>Column compression</td>
<td>( z \leftarrow u/x )</td>
<td>( z_j = u_j/\tilde{x}_j )</td>
</tr>
<tr>
<td>Mesh</td>
<td>( z \leftarrow \tilde{u}_x )</td>
<td>( \tilde{u}_x = x ) and ( \tilde{u}_x = y ) (Select in order from ( x ) or ( y ) as ( \tilde{u}_i = 0 ) or 1)</td>
</tr>
<tr>
<td>Mask</td>
<td>( z \leftarrow /y \cdot u/x )</td>
<td>( \tilde{u}_x = \tilde{x} ) and ( \tilde{u}_y = \tilde{y} ) (Mesh using zero vector for ( y ))</td>
</tr>
<tr>
<td>Expansion</td>
<td>( z \leftarrow u/\tilde{x} )</td>
<td>( \tilde{x} = \tilde{u}_x - \tilde{x} ) and ( \tilde{u}_y = \tilde{y} ) (Mesh using zero vector for ( y ))</td>
</tr>
<tr>
<td>Catenation</td>
<td>( z \leftarrow v \cdot u )</td>
<td>( v = (x_0, x_1, \ldots, x_{</td>
</tr>
</tbody>
</table>

**Base 2 value**

- \( x \leftarrow \tilde{u}_x \) Value of \( u \) as a base 2 number
- \( x \leftarrow /u \) \( x_i = 1 \) if \( u_i = 1 \)
- \( x \leftarrow u \) \( x_i = 1 \) if \( u_i = 0 \)
- \( x \leftarrow \tilde{u}_x \) \( x_i = 1 \) if \( u_i = 0 \)

**Left rotation**

- \( x \leftarrow k \cdot x \) \( x_i = 1 \) if \( k \cdot u_i = 1 \)

**Right rotation**

- \( x \leftarrow k \cdot x \) \( x_i = 1 \) if \( k \cdot u_i = 1 \)

**Row reduction**

- \( x \leftarrow /y \cdot u/x \) \( x_i = 1 \) if \( y(x_i) = 1 \)

**Column reduction**

- \( x \leftarrow /y \cdot u/x \) \( x_i = 1 \) if \( y(x_i) = 1 \)

**Matrix product**

- \( x \leftarrow \tilde{x}_0 \cdot \tilde{x}_1 \) \( x_i = 1 \) if \( y(x_i) = 1 \)

**Miscellaneous**

- **Maximum prefix** \( w \leftarrow u/a \) \( w \) is the maximum length prefix (suffix) in \( u \)
- **Maximum suffix** \( w \leftarrow u/a \) \( w \) is the maximum length suffix (prefix) in \( u \)
- **Representation** \( z \leftarrow \tilde{x}_x \) \( z \) is the vector representation of the character \( x \)

\( a/0 = (3,5,7), Z/\Delta = (4,6,8), z/\Delta = (z_0, z_1, z_2) \)

\( \tilde{u}_x = (0,2,4), \tilde{u}_y = (0,1,3,5) \)

**Operations for Microprogramming**

- From the collection of the Computer History Museum (www.computerhistory.org)
1. Common Language for Hardware, Software, and Applications

PROGRAM 1

Translation from complete parenthesis statement to equivalent Lukasiewicz statement

Translation from complete parenthesis statement $e(0)$ to equivalent Lukasiewicz statement $l$

Search Algorithms. Figure 1 shows search procedures, the relations among the canonical forms of symbolic logic, and matrix inversion.

Matrix Inversion. The method of matrix inversion using Gauss-Jordan (complete) elimination with pivoting and restricting the total major storage to a single square matrix augmented by one column (described in References 14 and 15) involves enough selection, permutation, and decision type operations to render its complete description by classical...
Proceedings—Fall Joint Computer Conference, 1962 / 127

$k = \{\text{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}\}$

$t(k_i) = 1 + (6_i, n_i)$, where

$n = (19, 13, 20, 23, 20, 6, 19)$

$(n_i$ is the rank in the alphabet of the first letter of $k_i$)

$x = (2, 2, 3, 6, 3, 1, 2)$, where $x_i = t(k_i)$.

$d = (1, 2, 3, 6)$, and $x = 6$.

Data of examples

Overflow

Overflow with chaining

Single table with chaining

Open addressing system-construction and use of table

Figure 1. Programs and examples for methods of scanning equivalence classes defined by a 1-origin key transformation.
Matrix notation rather awkward. Program 2 describes the entire process. The starred statements perform the pivoting operations and their omission leaves a valid program without pivoting; exegesis of the program will first be limited to the abbreviated program without pivoting.

Program 2 - Step 1 initializes the counter \( k \) which limits (step 11) the number of iterations and step 3 appends to the given square matrix \( M \) a final column of the form \((1, 0, 0, \ldots, 0)\) which is excised (step 12) only after completion of the main inversion process. Step 7 divides the pivot (i.e., the first) row by the pivot element, and step 8 subtracts from \( M \) the outer product of its first row (except that the first element is replaced by zero) with its first column.* The result of steps 7 and 8 is to reduce the first column of \( M \) to the form \((1, 0, 0, \ldots, 0)\) as required in Jordan elimination.

The net result of step 9 is to bring the next pivot row to first position and the next pivot element to first position within it by (1) cycling the old pivot row to last place and the remaining rows up by one place, and (2) cycling the leading column \((1, 0, 0, \ldots, 0)\) to last place (thus restoring the configuration produced by step 3) and the remaining columns one place to the left. The column rotation \( \overrightarrow{M} \) rotates all columns save the first upward by one place, and the subsequent row rotation \( \overrightarrow{p} \) rotates all rows to the left by one place.

The pivoting provided by steps 2, 4, 5, 6, 10, and 13 proceeds as follows. Step 4 determines the index \( j \) of the next pivot row by selecting the maximum over the magnitudes of the first \( k \) elements of the first column, where \( k = \nu(M) + 1 - q \) on the \( q \)-th iteration. Step 6 interchanges the first row with the selected pivot row, except for their final components. Step 5 records the interchange in the permutation vector \( \overrightarrow{p} \) which is itself initialized (step 2) to the value of the identity permutation vector \( \overrightarrow{1}(k) = (1, 2, \ldots, k) \). The rotation of \( \overrightarrow{p} \) on step 10 is necessitated by the corresponding rotations of the matrix \( M \) (which it indexes) on step 9. Step 13 performs the appropriate inverse reordering among the columns of the resulting inverse \( M \).

1. \( k \leftarrow \nu(M) \)
2. \( p \leftarrow \overrightarrow{1}(k) \)
3. \( M \leftarrow M \oplus \overrightarrow{1} \)
4. \( j \leftarrow ((\overrightarrow{a} \uparrow M) \uparrow 1)_1 \)
5. \( \overrightarrow{p} \leftarrow \overrightarrow{p} \)
6. \( \overrightarrow{a} \leftarrow \overrightarrow{a} / M^j \)
7. \( M \leftarrow M^j \div M^j \)
8. \( M \leftarrow M - (\overrightarrow{a} \times M^j) \times M^1 \)
9. \( M \leftarrow M \uparrow (\overrightarrow{a} \uparrow M) \)
10. \( p \leftarrow 1 \uparrow p \)
11. \( k \leftarrow k - 1 \)
12. \( M \leftarrow \overrightarrow{a} \leftarrow \overrightarrow{M} \)
13. \( M \leftarrow M \downarrow \overrightarrow{1} \)

**Program 2**

Matrix inversion by Gauss-Jordan elimination

*The outer product \( Z \) of vector \( x \) by vector \( y \) is the "column by row product" denoted by \( Z \leftarrow x \times y \) and defined by \( Z^j = x_i \times y_j \). The row rotation \( \overrightarrow{k} \times X \) is a row-by-row extension of the rotation \( \overrightarrow{k} \times x \), that is, \( \overrightarrow{(k \times X)}^j = \overrightarrow{k}^j \times X^j \). Similarly, \( \overrightarrow{(k \times X)}^j = \overrightarrow{k}^j \times X^j \)."
A description in the ALGOL language of matrix inversion by the Gauss-Jordan method is provided in Reference 16.

REFERENCES