CUPL - An Approach to Introductory Computing Instruction*

R. W. Conway

and

W. L. Maxwell**

TR 68-4

January 1968

Department of Computer Science
Cornell University
Ithaca, New York 14850

* This work was supported in part by the National Science Foundation under Grant GP 6827.

** The CUPL Project group consists of G. Blomgren, H. Elder, H. Morgan, C. Pottle, W. Riddle, and R. Walker as well as the authors.

CUPL - An Approach to Introductory Computing Instruction*

R. W. Conway and W. L. Maxwell**
Department of Computer Science
Cornell University

CUPL is a second-generation language and processor designed specifically for introductory instruction in computer programming. It combines a severely simple syntax (based loosely on PL/I) and very extensive tutorial and diagnostic assistance by the processor. The processor is core-resident and compiles very rapidly. The result is an effective instructional system that can be used for large numbers of students with modest demands on computer capacity. Technically CUPL is interesting for the error-correcting capability of the compiler and the provision of direct operations for matrix algebra.

CUPL is a special purpose programming language for which a compiler and a supervisor are available for IBM 360 systems. Fundamentally the language and the implementation constitute a unique approach to the problem of introducing general purpose computer programming to large numbers of students. Our premise is that a subset of a "production" language is not the most appropriate vehicle for this purpose, and that standard compilers and supervisors do not provide the neophyte with as much diagnostic and tutorial assistance as is possible.

* This work was supported in part by the National Science Foundation under Grant GP 6827.

** The CUPL Project group consists of G. Blomgren, H. Elder, H. Morgan, C. Pottle, W. Riddle, and R. Walker as well as the authors.

CUPL is admittedly--in fact, deliberately--limited in scope and capacity and many students are compelled to graduate to a senior language whenever their computing problems become significant. This is not a difficult step and we willingly subject a fraction of students to it in order to obtain the advantages of the CUPL introduction to computing for the larger number of students. The essential concepts that must be conveyed in an introduction to computing are virtually independent of the language employed and, in fact, the language should be as transparent as possible. Time devoted to instruction in the mechanics of language or operating system is necessarily deducted from the time available for ideas and basic concepts.

CUPL is the direct descendant of the CORC language and operating system (1). Designed and implemented in 1962 for the Burroughs 220 and the Control Data 1604, CORC represented a number of novel concepts:

1. A very limited and simple source language syntax
   a. Limited number of types of statement--with all statements being executable imperatives (no declarative statements).
   b. No "type" distinction for variables--all values carried internally as floating-point.
   c. Fixed formal input and output operations.
2. A tolerant compile-and-go operating system
   a. Compiler makes more or less plausible corrections of every error of syntax so that a syntactically-correct executable object program is always produced. (Each correction is of course described on the program listing.)
   b. Every program enters execution and runs to termination or until a preset limit (time, page count, error count) is exceeded.
   c. No delivery or preservation of object code, hence no opportunity to compile programs in sections for subsequent assembly and linkage.

3. Core-resident compiler and execution monitor.
   a. First of a number of consecutive jobs loads the compiler and monitor from the systems tape; second and subsequent jobs are processed with negligible system setup overhead.
   b. System protected by monitoring values of all subscripts during execution.
4. Source language communication with programmer--source statement numbers and identifiers are preserved during execution.
   a. Each variable on output is automatically identified with name as well as value.
   b. Execution messages identify source language statements by number and variables by source language identifier.
   c. Automatic dump of final values of simple variables after termination of execution.
   d. Automatic count of frequency of encounter during execution for each statement or block label.

Since 1962 each of these ideas has appeared in other languages and systems, but CORC remains the extreme case in which they are all combined in a single system. CUPL preserves all of these characteristics of CORC with an even simpler syntax. CUPL also includes significant additions:

1. Dynamic allocation of storage to arrays.
2. Direct matrix algebraic operations.
3. Provision for dynamic execution tracing.

CUPL has been implemented in a much smaller memory (65K bytes) than CORC and can be multiprogrammed with other processes in a larger memory. The error correction is more ambitious and effective and the compiler reconstructs source statements to display the effect of corrections rather than describe the correction. Superficial changes in syntax were made to incline CUPL toward PL/I rather than the Burroughs implementation of Algol-58 upon which CORC was based, but CUPL is not a strict subset of PL/I and the similarity to CORC is unmistakable.

CUPL Source Language Syntax

The following is a brief description of the CUPL syntax. This is intended to illustrate the simplicity of the language and not serve as a precise definition (a full BNF description is available). For meta-language let:

$v_1, v_2, \ldots$	be variables (scalar, vector, or matrix)
$e_1, e_2, \ldots$	be well-formed arithmetic expressions
$r_1, r_2, \ldots$	be relational operators (=, GT, GE, LE, LT, NE)
$s_1, s_2, \ldots$	be statements
label	be a statement label
[ ]	indicate an optional syntactic element.

There are eight types of executable statements in CUPL:

[label]	LET v = e
[label]	GO TO label
[label]	READ $v_1, v_2, \ldots$
[label]	WRITE $v_1, v_2, \ldots$, 'literal message', ...
[label]	STOP
[label]	IF e_1 r_1 e_2 [AND e_3 r_2 e_4] [OR e_5 r_3 3_6] [THEN s_0] [ELSE s_2]
[label]	ALLOCATE v_1 (e_1, e_2), v_2(e_3), ...
[label]	PERFORM label [e TIMES]
[label]	PERFORM label WHILE e_1 r_1 e_2 [AND e_3 r_2 e_4] [OR e_5 r_3 e_6]
[label]	PERFORM label FOR v = e_1 TO e_2 [BY e_3]

A dynamic monitoring of value assignment is provided by a ninth statement type:

[label] WATCH v_1 v_2, ...

For each of the variables listed this causes an appropriate message to be printed the next ten times that a value is assigned by either a LET or a READ statement. Comments that are to appear only in the source listing are given as:

COMMENT literal message

An essential feature of the CUPL syntax is the block structure. Sequences of CUPL statements can be designated as a block by delimiting them with the words BLOCK and END with matching labels. For example:

```
TERM    BLOCK
        LET VAL3 = VAL3/X
        LET SUM = SUM + VAL3
        WRITE SUM, VAL3, X
TERM    END
```

A block can be located anywhere in the program--inserted between any pair of consecutive statements--without affecting the execution of the program. The block is skipped when encountered in the course of sequential execution and is entered (called) only by means of a PERFORM statement:

PERFORM TERM ...

As a matter of form a block is often placed directly following its controlling PERFORM statement but this is not necessary and, since two or more statements can refer to the same block, it is not always possible. Some programmers perfer to collect all blocks in a common position in the program.

This block structure is peculiar to CUPL and we believe that it makes an important contribution to the simplicity of the language. This single construct provides both iteration control ("DO loops"), and a rudimentary procedure and subroutine capability. Although some tasks are awkward (multiple arguement subroutines) and there is some extra writing (BLOCK line when the block immediately follows the controlling statement) the concept is easily understood and easily followed in post-mortem. The redundancy is put to good use in error analysis and correction.

Matrix Algebra

A significant feature of CUPL is the ability to program directly in terms of arrays rather than in terms of the individual elements of arrays. This means that one can use vector and matrix notation, in a reasonably natural and familiar way, in computation, logical conditions and input/output operations. Although this ability is still somewhat unusual in programming languages we believe that it is entirely consistent with CUPL's form and intent. Modern undergraduate mathematics is making increasing use of linear algebra and trying to lead students to think in terms of arrays and array operations. A computing language should not require the student to redefine each of these operations--element by element with nested blocks--each time that it is used. The only difficulty with .is feature of CUPL seems to be that the ease of use and the execution speed advantage attracts experienced users--who then object to certain other characteristics of the system.

Although a serious effort was made to make CUPL a compatible subset of PL/I, the definition of array operations was one of several areas in which the price of compatibility was judged to be excessive. CUPL adopted the normal definitions of linear algebra rather than the element by element operations of PL/I. The arithmetic operations and functions are described below. In this description A and B represent matrices (or matrix-valued expressions), a_{ij} and b_{ij} are individual elements, and a_r, a_c, b_r and b_c designate the number of rows and columns of A and B respectively. S represents a scalar-valued expression; s is its current value. Conformability conditions for each operation are given after the symbol "CC".

1. A ± B is the matrix C defined by $c_{ij} = a_{ij} \pm b_{ij}$.
CC: $a_r = b_r$, $a_c = b_c$. C is also $a_r \times a_c$.

2. S*A or A*S is the matrix C defined by $c_{ij} = sa_{ij}$. CC: none. C is also $a_r \times a_c$. Following standard notation, -A is allowed as a substitute for (-1)*A and A/S as a substitute for A*(1/S).

3. A*B is the matrix C defined by $c_{ij} = \sum_{k=1}^{a_c} a_{ik}b_{kj}$. CC: $a_c = b_r$. C is $a_r \times b_c$.

4. ABS(A) is the matrix C defined by $c_{ij} = a_{ij}$. CC: none. C is also $a_r \times a_c$.

5. TRN(A), the <u>transpose</u> of A, is the matrix C defined by $c_{ij} = a_{ji}$. CC: none. C is $a_c \times a_r$.

6. TRC(A), the <u>trace</u> of A, is a scalar whose value is $\sum_{i=1}^{a_r} a_{ii}$. CC: $a_r = a_c$.

7. SGM(A) is a scalar whose value is the sum of all the elements of A. CC: none.

8. DET(A) is the <u>determinant</u> of A, a scalar. CC: $a_r = a_c$.

9. INV(A) is the <u>inverse</u> of A; CC: $a_r = a_c$. C is also $a_r \times a_r$.

10. DØT(A,B), defined only when A and B are vectors, is the scalar $\sum_{i=1}^{a_r} a_i b_i$. CC: $a_c = b_c = 1$, $a_r = b_r$.

11. MAX(A,S,B,...,Z) is the maximum of the current values of all the scalars and all the elements of all the arrays in the list A,S,...,Z. CC: none. MIN(.....) is defined similarly for the minimum value.

12. PØSMAX(A) gives the position of the maximum element of A . PØSMAX(A) is the positive integer designating the <u>row</u> of A in which the maximum element occurs.* If the maximum appears in more than one row then the value of PØSMAX(A) is the smallest of the possible row subscripts. PØSMIN(A) is defined similarly. CC: none.

Array expressions can be used in the conditions employed in the IF and PERFORM...WHILE statements. The relations are defined in the usual way:

A = B means $a_{ij} = b_{ij}$ for all pairs i,j.

A NE B means $a_{ij} \neq b_{ij}$ for at least one pair i,j.

A LE B means $a_{ij} \leq b_{ij}$ for all pairs i,j.

A GE B means $a_{ij} \geq b_{ij}$ for all pairs i,j.

A LT B means $a_{ij} \leq b_{ij}$ for all pairs i,j and
$a_{ij} < b_{ij}$ for at least one pair i,j.

A GT B means $a_{ij} \geq b_{ij}$ for all pairs i,j and
$a_{ij} > b_{ij}$ for at least one pair i,j.

In each case the conformability conditions are $a_r = b_r$, $a_c = b_c$.

Arrays can be used directly in input and operation operations. The statement

```
READ A, B, ...
```

will read the elements of A from the data-list assuming that they are in row by row order, followed by the elements of B, etc. The statement

```
WRITE A, B, ...
```

will print the elements of A in row by row order. Each row begins a new print line (with a maximum of five elements per line) and is appropriately labeled.

* These functions are usually used with vectors, for which the row subscript determines the position. To locate both row and column in a general matrix one can use

```
LET RØWMAX = PØSMAX(A)
LET CØLMAX = PØSMAX(TRN(A(RØWMAX,*)))
```

In any of these contexts one can refer to an entire array, an individual element, or a particular row or column, subject only to the conformability requirements of the particular operation being performed:

A	represents the entire array
A(I,J)	represents a particular element
A(*,J)	represents the J^{th} column of A
A(I,*)	represents the I^{th} row of A.

IDN is used to represent the identity matrix. This is automatically adjusted to whatever size is required by the context in which it appears.

A matrix with all elements of equal value can be produced by giving a constant on the right of an assignment statement. For example,

```
LET UNIT = 1
```

assigns the value 1 to each of the elements of UNIT. This special type of assignment statement, and the identity matrix are the only cases where dimensions are automatically managed by the system. In general it is the programmers' responsibility to dimension his arrays so that all expressions are meaningful and conformable. This is, of course, a requirement of linear algebra and not a peculiarity of CUPL. CUPL monitors conformability conditions during execution and notifies the programmer of any lapses. CUPL then modifies dimensions to achieve the required conformability and execution proceeds. Continuation is in the hope of yielding additional diagnostic information; there is little chance that the computational results will be what the programmer intended.

Dynamic Storage Allocation

CUPL makes the assignment of storage space to arrays when an ALLOCATE statement is encountered during execution of the program, rather than during compilation. This makes it unnecessary to draw the distinction for the student between execution-controlling "statements" and compiler-controlling "declarations". More importantly, this permits the dimensions of arrays to depend upon the results of calculations and/or external data, and for the dimensions to change during execution of the program. This allows the use of data-directed routines such as the following:

```
        READ N
        PERFORM CYCLE N TIMES
CYCLE   BLOCK
        READ I, J
        ALLOCATE PRIMARY(I, I+3), SECOND(J)
        PERFORM MATRX4
CYCLE   END
        STOP
MATRX4  BLOCK
        etc.
```

It is also true, although of incidental importance for CUPL, that dynamic allocation permits efficient use of memory since all arrays do not have to be carried simultaneously and at the maximum dimensions that are needed at any time during execution. Dynamic allocation permits arrays to be expanded and contracted according to immediate needs. *

*Initial allocation of an array sets all its elements to 0. Subsequent reallocation of the array preserves values of elements common to both allocations, and zeros elements outside this range.

It is considerably more difficult to implement a dynamic storage management system but it is characteristic of CUPL that the processor be taxed more heavily than is usual in order to spare the new programmer having to learn at what stage different classes of action are performed.

Input and Output

Input and output functions, which often seem to consume instructional effort quite out of proportion to their significance, are a fruitful area for simplification in a limited-objective language. The CUPL communication statements are quite restrictive but are exceedingly simple to use. The only input to the system is a sequence of source card images, controlled by the READ statement; the only output is a sequence of printer line images, controlled by the WRITE statement. In both cases format is implicit in the system.

The WRITE statement:

```
WRITE X, TOTAL, SQUARE(I,J), P
```

displays source language names and current values of the listed variables in a three per line format:

```
X =   2.50000000 TOTAL =   3.70000000E+12   SQUARE(3,4)=4.56600000
P =   3.23456789
```

Each WRITE statement begins a new line. The automatic naming of output variables can be suppressed by marking particular variables with a "/". Arbitrary messages can be printed by enclosing the literal character string in quotes in the variable list. For example:

```
WRITE X, 'LINE TOTAL=', /TOTAL
```

might yield the printed line:

```
X =   2.50000000   LINE TOTAL = 3.70000000E+12
```

The line format is simply six 20 character fields. Each element--name or value--occupies one field and literals are assigned one or more fields depending on their length. If labeled output is specified -- X,Y,Z -- three names and three values fill a line. If unlabeled

output is specified -- /X,/Y,/Z, ... -- six values can appear on a line. The two modes can be intermixed arbitrarily. (The system automatically prevents the name and value of a single variable from being separated by the end of a line.) The commas between variables on the WRITE list, like almost all CUPL punctuation, are not really necessary. A blank space is adequate. However, two or more adjacent commas on the list indicate an empty position on the list and cause one or more fields to be skipped on the output. It is possible with this simple structure to produce fairly attractive output and this ability represents a significant improvement over the absolute rigidity of CORC.

Any data required by a program is listed after the flag *DATA following the last statement of the program. This is a continuous list in which the boundary between cards is of no significance (except that individual values cannot be divided over a card boundary.) Values may be placed one per card or several per card just so long as the order is maintained for proper encounter by the READ statements during execution. Values on the list are separated by either commas or spaces. Values on the list may be provided with the name of the variable to which the value is to be assigned:

```
*DATA  7, 13.5, -1.667E+5, X=4.5, 444, NEWBASE= 9
```

When such a name is given the execution monitor checks this against the variable on a READ list that actually reads this value. The name on the READ list controls the assignment that is made, but in the event of disagreement the monitor can issue a warning that there is surely trouble in the ordering of the data, or in sequence control in the program.

Arrays can be given directly on either READ or WRITE lists avoiding the necessity of using nested blocks to load or display the contents of arrays. In either case a row by row format is assumed.

By writing WRITE ALL at any point in the program the user can obtain a dump of the current values of all of the simple variables (not arrays) used in the program. Such a dump is provided automatically on termination of execution, but it can be requested

as many times earlier as the programmer requires. In the event of an unnatural termination (time limit, page limit, or error limit) the system automatically provides a sequence of WRITE ALL dumps--one inserted after each of the last twenty statements executed. The final dump also lists all of the labels (both simple statement labels and block labels) in the program along with a count of the number of times each was encountered during execution. As far as we are aware the provision of this information is unique in CUPL (carried over from CORC) and experience has shown that it is extremely useful in tracking down errors in a program. The system also provides a list of the statement numbers of the last sixteen statements executed.

Following the final dump the system automatically lists the first ten values from the *DATA list. In spite of repeated advice to do so, many students cannot bring themselves to echo-print input during the testing of a program and it is often impossible to diagnose difficulty without positive information as to just what was received by the early READ statements.

Error Correction in CUPL

In the detection of program errors CUPL differs from other processing systems only in degree, but once discovered, its treatment of errors is radical. This can be summarized in the following way:

1. The compiler transforms every source program into syntactically perfect form.
2. Every program reaches execution phase.
3. Execution is aborted only when time, page or error count limits are exceeded and not by individual events. (These limits are set by the header card for a batch run and can be easily varied.)

The idea is simply to keep the process going as long as possible to obtain maximum diagnostic information. By breaking the "one bug per pass" pattern one can significantly reduce the average number of job submissions required to achieve successful execution. Prolonging the life of a moribund program might appear to be wasteful of machine time, but based on experience with CORC we believe that the reduction in the average number of passes per job more than offsets the increase in the average per pass. Even if that were not the case the improved service to the user might well be worth the expenditure of additional machine time.

Of course the value of information obtained by continuing the process after encoutering a serious error depends in large part on the plausibility of the repair effected. We believe that CUPL represents a constructive demonstration that plausible repairs can be made in a useful proportion of difficulties. We have been cataloguing student programming errors for five years and refining our correction techniques over that period so that CUPL has some fairly interesting abilities. CUPL does not represent a general theory of error correction; its ability is the aggregate of several hundred ad hoc techniques.

The CUPL compiler is logically divided into two sections. The first transforms the source language program into a syntactically perfect intermediate language. The second assumes syntactically perfect input and produces object codes. These sections are very cleanly divided so that refinement of the repair techniques of the first section can continue without affecting the body of the compiler.

The major limitation in the present strategy of repair is that, with only a few exceptions, it is based on a single scan of the source program. In many cases more plausible repairs could be made after a multiple-pass scan but this greatly complicates and slows the processor and we rationalized our laziness with a "diminishing return" argument.

When one or more errors are encountered in a statement and the scanner introduces corrections into the intermediate language a "reverse translation" routine is triggered that reproduces a source language statement equivalent to what has been produced in intermediate form. Comparison of the card image and the corrected form usually suggests the nature of the error. In case this is not sufficient information the specific errors are indicated at the right side of the page, with a numbering system coded to explanatory paragraphs in an appendix of the CUPL Manual. The program listing appears as follows: (Line numbers are provided by the system and do not appear on the cards).

```
          LINE LABEL      STATEMENT                          ERRORS

          0001            READ A, B3, BASE
          0002  TEST      IF BASE LE 16 THEN GO TO LOW
          0003            CUM = B3**A

ERROR IN  0004            LET SUM = BASE +                   7D
CUPL USES 0004            LET SUM = BASE + 1

ERROR IN  0005  TEST      READ X, /ANGLE                     52,05
CUPL USES 0005            READ X, ANGLE
          0006            GO TO ADVANCE
WARNING   0007            PERFORM COMP A TIMES               04

ERROR IN  0008            LET X -A + B3                      11
CUPL USES 0008            LET X = -A + B3

ERROR IN  0009            LET X = /23                        01,12,70
CUPL USES 0009            LET X = 1

ERROR IN  0010  PASS      X - A                              01,02
CUPL USES 0010  PASS        (NO OPERATION IS PERFORMED)

ERROR IN  0011            PRINT X, ANGLE                     70
CUPL USES 0011            WRITE X, ANGLE
```

Several observations about CUPL corrections can be made from this example:

a. CUPL stands ready with the constant "1" to complete or replace damaged arithmetic expressions.

b. In some cases, such as the inaccessible statement in line 0007, CUPL will warn the programmer of a construction that is apparently pointless although syntactically correct.

c. The redundancy in the CUPL syntax is used by the system to try to reconstruct statements. For example, an assignment statement can survive with either the word "LET" or the operator "=" but CUPL will not try to make an assignment statement out of a construction that lacks both.

d. When a particular statement is beyond CUPL's comprehension (for example, line 0010) it is replaced by a null statement. This statement is still counted in execution and, if provided with a label, can still be a target for transfer.

One of the more interesting corrective abilities and one which not infrequently manages to restore a program to what the author really intended is the procedure for maintaining the distinction between label names and variable names. An identifier in CUPL can represent either a variable, a statement label or a block label, and although there are strict and explicit rules that a particular identifier should not be used for more than one of these purposes CUPL can often keep track of the intent when the rule is violated. In such cases CUPL creates new identifiers by providing a prefix before the given name. For example, in the following, "X" is used initially as a variable, then as a statement label, and finally as a block label. If the statements referring to the identifier are otherwise in reasonably good form CUPL can usually keep the names sorted out:

	LINE	LABEL	STATEMENT	ERRORS
	0001		LET X = Y + 3	
ERROR IN	0002	X	PERFORM X UNTIL X GT Y - P(J)	41,07
CUPL USES	0002	$X	PERFORM $$X UNTIL X GT Y - P(J)	
ERROR IN	0003	X	BLOCK	41
CUPL USES	0003	$$X	BLOCK	
ERROR IN	0004	X	WRITE X, Y, P	05
CUPL USES	0004		WRITE X, Y, P	
ERROR IN	0005	X	END	41
CUPL USES	0005	$$X	END	
ERROR IN	0006		GO TO X	07
CUPL USES	0006		GO TO $X	

Logically the most difficult of CUPL correction efforts is with respect to the block structure of a program. CUPL attempts to make sure that block labels are always present in matched pairs (on BLOCK and END lines) and that blocks are always closed in the opposite order of their opening. For example,

	LINE	LABEL	STATEMENT	ERRORS
	0001	B1	BLOCK	
	0002	B2	BLOCK	
	0003	B3	BLOCK	
ERROR IN	0004	B2	END	4C
CUPL USES	0004	B3	END	
ERROR IN	0005	B3	END	4C
CUPL USES	0005	B2	END	
ERROR IN	0006		END	48
CUPL USES	0006	B1	END	

CUPL assumes that the most common block location will be directly after the controlling PERFORM statement and reconstruction is biased toward this form. The single-scan strategy becomes a problem in this regard. CUPL cannot assume that a block always follows a PERFORM statement so that by the time an END line is encountered and it is clear that a BLOCK line was omitted the printed listing is irretrievable. CUPL can patch the object code to insert a BLOCK line but it cannot get it on the listing in the proper place. Error code 45 indicates this type of difficulty:

	LINE	LABEL	STATEMENT	ERRORS
	0001		PERFORM BA	
	0002		READ X, Y, Z	
WARNING	0003	BA	END	45
	0004		PERFORM BB	
CUPL USES	0005	BB	BLOCK	4B
ERROR IN	0006	BB	READ X, Y, Z	4A
CUPL USES	0006		READ X, Y, Z	
	0007	BB	END	
ERROR IN	0008		DO BC	70
CUPL USES	0008		PERFORM BC	
	0009		READ X, Y, Z	
ERROR IN	0010		END	45,47
CUPL USES	0010	BC	END	

CUPL also undertakes a certain amount of spelling correction for both reserved words and identifiers. Toward this end the usage of each identifier is catalogued as the program is scanned. When an identifier is used "unreasonably" (for example, a variable that appears exactly once in the program, or appears only on the left hand side of assignment statements) it is a candidate for the spelling analysis routine. This routine compares the mis-used identifier with all other identifiers used in the program. If it is sufficiently similar to one of the others CUPL will equate the two and declare that

> ".....APPEARS TO BE A MISSPELLING OF ..... AND THE TWO HAVE BEEN EQUATED"

There are many reasons why this does not always make the proper repair, but it rarely makes the program any worse and it succeeds in just enough cases to make it worth doing.

These examples represent a very small fraction of the error correction procedures. A better idea of their scope can be obtained by scanning the error messages in Appendix D of the CUPL Manual. However, it is very difficult to really appreciate the effect of these procedures without scanning the output of randomly selected student jobs.

Implementation

CUPL has thus far been implemented only for IBM 360 Systems. The program is written in 360 Assembly Language. Separate versions have been produced for DOS and OS. A minimum configuration of 65,536 bytes of core (F level system) and one 2311 disc is required,* but many options in configuration can be specified at the time that the system is generated. The program is highly compartmented so that all of the instructions that are dependent on either configuration or operating system are segregated into a single supervisor-control section; none of the primary section of the program need be touched.

A batch of source programs in the CUPL language appears to be a single task to DOS or a single job step to OS. A header card on the first program calls the CUPL supervisor-control module from disc. This supervisor retains control until the last program of the batch has been processed. Normally all sections of the CUPL system remain in core throughout the batch run. Object code is compiled directly into an area of core not occupied by the system. User working storage is also a distinct area of core. CUPL monitors the values of all subscripts during execution so that there is no way that a user's program can run out of control and damage either itself or the CUPL system. With this strategy the systems overhead between jobs is approximately 1 millisecond (on a Model 40) which is the time required to re-initialize the symbol table in core. Termination of execution of one program is simply a transfer to the compiler -- which has been inactive, but is still resident -- and a new user program is overlaid on the old.

Including the necessary resident modules of DOS the complete CUPL system occupies approximately 58,000 bytes of core. In an F level

* The system will operate without the disc if the optional scanner-overlay is not used.

system this leaves the user only 7,500 bytes (or 900 words since full-word precision is used throughout.) Based on five year experience with CORC and four months with CUPL this appears to be adequate for a majority of introductory student programs. However, when it develops during execution* that additional core is required CUPL overlays the scanner section of the compiler to make an additional 17,000 bytes available. The system is thus rather frugal with both time and space. The inactive compiler does not block the user from having larger amounts of core when needed, and the supervisor reloads the scanner between jobs (approximately 0.1 second) only when this is necessary. Of course, if more than F level core is available to the system the overlay and reload are invoked less frequently. The OS version of CUPL requires a minimum of G level (128K.)

The CUPL system is itself entirely relocatable and reentrant. It can readily be used in a partition with another processor in either background or foreground status. Reentracy, of course, permits a single copy of the system to process several different source programs in alternation and was provided in anticipation of a time-shared version of CUPL. A time-shared version will only require the replacement of the supervisory-control section and replication of the program status block.

System Performance

It is difficult to give precise and meaningful statistics of operating speeds but the following should give a rough idea of capability. Running on a 360 Model 30 with a 2540 card read/punch (1000 cards per minute) and a 1403-N1 printer (1100 lines per minute) the scanner is limited by the card read speed for error-free input. (Statements containing an error cause two print lines and a space and the printer is momentarily

* Note that the dynamic dimensioning of arrays means that core requirements are not known to the system until execution time.

limiting.) The generation of object code for typical (50-100 statement) programs takes place as the printer indexes to a new page to prepare for execution output and there is rarely any apparent pause. Execution time, of course, depends upon the particular program but is rather typically limited by the printer.

We have done more careful timing on a Model 40. Running in a tape-to-tape mode to eliminate reader or printer restriction the scanner is capable of something over 4000 statements per minute. Running as a tape-to-tape background partition behind DOS E level FORTRAN with the reader and printer assigned to FORTRAN there is no apparent degradation of FORTRAN performance--the WAIT light burns less brightly than usual.

Cornell is installing a Model 65 in October. With suitable blocking of input and output we expect a scan speed of more than 30,000 statements per minute on that machine.

The various forms of monitoring cause CUPL to execute more slowly than FORTRAN. The magnitude of the difference depends greatly on the type of program. The minimum is a penalty of approximately 25%; the worst possible case is a program that consists entirely of doubly-subscripted variables. Subscript monitoring will cause such a program to run at about one-eighth FORTRAN speed. However, we question whether this represents a serious disadvantage for the system since the existence of the matrix operations in CUPL makes much of the use of subscripted variables obsolete. The matrix operations are not only much more natural for the programmer, but they mean a substantial reduction in the number of source statements and a significant improvement in execution time. For example, for a program that consisted entirely of the repeated multiplication of two

ten by ten matrices, the following execution times were obtained on a Model 40:

E FORTRAN (using subscripts)	0.54 seconds per multiply
CUPL (using subscripts)	4.76 seconds per multiply
CUPL (using direct matrix mult)	.22 seconds per multiply

CUPL vs WATFOR

The argument as to whether CORC's source language simplicity and tolerant processing were adequate compensation for the fact that it was not a subset of FORTRAN was to a great extent irrelevant. The efficiency of batch processing with a core-resident system reduced by a factor of about ten the total time required to process student jobs. It permitted Cornell to embark on ambitious instructional programs and to adopt very liberal procedures with respect to undergraduate use of the computer. It was more than three years after the introduction of CORC before core-resident FORTRAN systems became available for the 1604.

Whereas one might have expected that the transition from tape to disc residence for systems would have essentially eliminated inter-job setup time, and made core-resident systems unnecessary, quite the opposite has occured. Increased demands on operating systems (and perhaps something less than optimal implementation) have resulted in incredibly large inter-job overheads. FORTRAN times for null-jobs of as much as 100 seconds are experienced by many small 360 installations. Effectively this means that unless such machines use a core-resident processor (and none is supplied by the manufacturer) they are not really very useful in an educational environment.

Since by now there are a number of core-resident systems one can have both efficiency and FORTRAN. A 360 version of WATFOR offers almost-compatible FORTRAN IV and other systems

will undoubtedly appear. WATFOR also offers substantially better diagnostic assistance than standard FORTRAN processors.

CUPL and WATFOR appear to be roughly comparable in speed of both compilation and execution. CUPL has some minor advantages:

1. CUPL requires less core. It will operate on an F level machine (65,536) where WATFOR requires G level (131,072). In these sizes CUPL will accomodate a program of approximately 350 statements; WATFOR about 200. CUPL overlays the compiler as required; WATFOR is permanently resident. These figures affect not only the size of the minimum operable system but also the necessary partition size in a multi-programming system.

2. CUPL error detection, correction and execution monitoring is much more extensive.

3. Matrix algebra and dynamic storage allocation are not available in WATFOR.

4. CUPL is somewhat more contemporary in form. Its relocatability makes it more adaptable to multi-programming; its reentrancy makes it adapatable to time-sharing supervisors.

Many people would probably prefer a relocatable, reentrant, error-correcting FORTRAN IV with matrix algebra that would operate in 65K (or less) of core but so far none has been announced. Actually we do not see CUPL and WATFOR as being directly competitive. WATFOR is an efficient way to run small FORTRAN programs; CUPL is an approach to introductory instruction. Cornell will use both processors. Many of the students who are introduced to computing through CUPL will graduate to FORTRAN and many of their FORTRAN programs will be processed through WATFOR.

CUPL in a Large-Scale Computing System

Cornell will provide CUPL processing on a 360 Model 65 running under HASP. Initially the system will be oriented to remote-job-entry and remote-output-delivery with bulk-core time-sharing due in mid-1968. The principal terminals in the system will be modified 360 Model 20's serving as card reading, line printing stations. With minor modifications the priority system under HASP will serve to batch jobs for CUPL and for WATFOR internally. Plans are to run a batch of CUPL and WATFOR jobs every ten or fifteen minutes. If necessary long-running production jobs will be checkpointed and rolled-out of at least one partition of core to permit the maintenance of this schedule. We estimate that one second of 65 CPU time spent on CUPL will generate three to four minutes of work for a basic Model 20 terminal. Although there will be a number of Model 20 terminals operating simultaneously it is obvious that the CPU demands for CUPL will be very modest.

The entire Cornell system (not just CUPL) will operate under a special job control language. This has an exceedingly simple structure and a tolerant, error-correcting scanner. It is intended to serve the needs of a large majority of users, but those who require greater flexibility can easily penetrate this language to obtain full OS facilities.

The system includes a special data management section outside of OS. This will permit CUPL and WATFOR users to have on-line storage of programs and data-sets and powerful editing ability in a simple and machine-efficient manner. We believe that this is crucial. A modern student-oriented system must offer on-line storage and unless this is done very simply and efficiently much of the advantage of core-resident processors will be negated.

The intent is to provide a remote-access, fast-turnaround system with on-line storage and editing for very large numbers of students with a minimum consumption of central facility capacity. The lack of split-second interactive capability for student work will be at least partially offset by the corrective actions of the compiler. This is certainly not as glamorous and probably not as effective as individual interactive terminals, but it would appear to be an order-of-magnitude less costly for a given number of students, both in terminal expense and in demand upon the central facility. It could be regarded as an interim system until large-scale time-sharing overcomes its current difficulties, and perhaps even then have a place as a low-cost alternative to time-sharing. In any event it should provide a reasonable and contemporary standard against which time-sharing can demonstrate its virtues.

Extensions of CUPL

We have no intention of eventually extending CUPL to become a general purpose production language. Most of CUPL's virtues lies in its simplicity and extensive assistance to the programmer and one or both of these would be lost in achieving the generality and efficiency that are necessary for a production language. Five years of experience have shown the present structure to be suitable for its intended purpose. Moreover we see no real benefit and some loss in postponing the graduation to a senior language for those students who have serious computing problems.

We are working on additions to CUPL that will permit introduction to the concepts of elementary list processing and simulation. Again this is based on CORC experience. CLP (2) was an extension of CORC that provided the basic features of SIMSCRIPT-type programming, and it proved to be tremendously useful. Students have been able to understand the essential concepts in this type of programming without spending a long time learning the details of a sophisticated list processing language. The necessary additions consist of structured operands,

several list management statements and co-routines (interruptable subroutines for quasi-parallel processes.) It appears that this will require a modification of the CUPL processor, rather than simple additions to it, and that at least 128K of core will be necessary. We are also in the process of implementing a CUPL - like language for instruction in the basic concepts of business data processing. (5) This has been sorely needed for some time since production languages in that area are even less suited for introductory instruction than is FORTRAN for scientific work. The result is that in a number of (graduate) schools of business an unfortunate and unnecessary amount of time has been spent in instruction in this area, but for the large majority of business students appropriate instruction in these crucial concepts has not been provided. We are implementing (interpretively in 128K) a language that will make it possible to teach the concepts of files, transactions, access methods and security problems without making a professional COBOL programmer out of the student. We expect the language to make it possible for a group of students to construct and operate an elementary management information system.

In spite of the arguments of the last section that remote-job-entry from high-speed terminals is an appropriate way to serve large-volume introductory instruction a time-sharing supervisor for CUPL will be produced. This will complement the existing supervisor and permit some interesting experimental comparisons of the two modes of operation. Of course, it may replace the present supervisor if the pressure for individual terminal computing is irresitable. The first time-shared CUPL will be a somewhat unusual system employing twelve-key keyboard telephones as terminals.(3) Information will be entered from the keyboard with only one stroke per character. The programmed scanner uses context to eliminate the ambiguity in transmission. A standard

audio-response unit will be used to provide output.

Distribution and Maintenance

Cornell is prepared to make CUPL available to anyone interested in trying the system. The normal distribution medium is magnetic tape but punched cards can be used. CUPL has been in use at Cornell for six months by more than one thousand students and is reasonably well checked-out. Errors undoubtedly remain and we intend to distribute periodic updates to anyone who has received a copy of the system. Instructional manuals, coding forms, 35mm slides and taped lectures for the system are also available.

July 1, 1967

References

1. Conway, R. W. and W. L. Maxwell, "CORC--The Cornell Computing Language," Communications of ACM, 6 (June 1963), 317-321.

2. Conway, R. W., J. J. Delfausse, W. L. Maxwell and W. E. Walker, "CLP - The Cornell List Processor," Communications of ACM, 8 (April 1965), 215-216.

3. Conway, R. W. and H. L. Morgan, "Tele-CUPL: A Telephone Time-Sharing System" Communications of ACM (September 1967).

4. Conway, R. W. and W. S. Worley, Jr., "Preliminary Description of the Cornell Operating System for the 360," Office of Computer Services, Cornell University, Ithaca, New York.

5. Morgan, H. L., "CUPL-DP, A Language for Introductory Instruction in Data Processing," Department of Computer Science, Cornell University, Ithaca, New York (July 1967).

6. Walker, R. J., CUPL--The Cornell University Programming Language, Department of Computer Science, Cornell University, Ithaca, New York (December 1966).