A FINAL COMMENT

Since this column will be my last as the book review editor for "The Newsletter," I wish to take this opportunity to express my thanks to everyone who has helped with reviews and suggestions during the past few years. Even though I can't mention all of you by name, I do wish to offer a special word of appreciation to Don Rosenthal, who first invited me to undertake this job and has helped me so often. I welcome the new book review editor, Larry Julien from Michigan Technological University, and wish him good luck as he begins his new job.

——End—— BYE Harry.

INTEGRATING COMPUTERS INTO THE UNDERGRADUATE CHEMISTRY CURRICULUM

by Harry E. Pence, Department of Chemistry SUNY Oneonta, Oneonta, NY

The Committee on Computers in Chemical Education sponsored a one-day symposium titled "Integrating Computers into the Undergraduate Chemistry Curriculum" at the ACS National Meeting in Washington, DC this Fall. The symposium was organized by Tom O'Haver (Univ. of Maryland) and Harry Pence (SUNY Oneonta) in order to encourage chemistry instructors at all levels to share information on their progress in integrating the variety of available computer software and hardware into an environment that will be most conducive for learning. Both the number of papers submitted to the symposium as well as the attendance demonstrated the high level of interest in this topic.

As might be expected, the various presentations represented a broad spectrum of approaches and technology. These papers covered all levels from high school through the senior year in college, and even at the introductory levels the computer techniques used were often extremely powerful and sophisticated. It was, however, obvious that even modest financial resources were sufficient to create productive computer integration. Several speakers described how they had used inexpensive software, matching grants, and other avenues to develop innovative programs.

The presentations at this symposium demonstrated the wide variety of ways in which computers are being used in undergraduate chemistry courses. Individualized learning is supported not only as computer-assisted instruction, but also by providing each student in large classes with unique homework and quiz assignments. Computer-controlled, multimedia presentation systems ("hyperbooks") will soon be widely available for lecture and self-study. Strategies for electronic literature searches are becoming a routine part of the instructional program. The increasing availability of resource rooms for chemical computing allows students to share their knowledge and participate in applications that go beyond those discussed in class.

It may be unfair to single out one development that was most exciting, but the frequent mention of programs that allow students to visualize chemical systems was especially impressive. Even though such methods have only recently obtained wide acceptance at the research level, several papers described their use in introductory classes. A broad range of software is capable of supporting molecular modelling and visualization, ranging from share-ware like MacMolecule to very elegant and expensive packages, like the CAChe system. These systems allow students at all levels to interact with molecules with a facility that's impossible with models. Allowing students to manipulate molecular systems in this way not only captures their imaginations, but it also seems likely that those who learn three-dimensional visualization in this way will develop a new understanding of chemistry and chemical reactions.

This symposium gave an exciting look at one of the more active areas of innovation in chemical education. No one seriously expects that computers will solve all problems in chemical education, but if these papers are an accurate indication, computers have an important role to play in whatever form the actual solutions may take.

INFORMAL NOTES ON PROGRAMMING-LANGUAGES

by K. W. Loach

Chemistry Dept., SUNY Plattsburgh.

LOACHKW@SNYPLAVA.BITNET or (518)564-4116

This is a follow-on from Part One (Comp. Chem. Educ. Newsletter, Fall, 1991), which covered the 'classical' languages

Fortran, Cobol, Pascal, etc. See Part One also for an introduction to the topic overall. The emphasis is on languages that can be used on personal computers, especially under DOS.

PART TWO: NEO-CLASSICAL LANGUAGES

This deals with well-defined languages that are not yet widely used by chemists. Each shows some interesting features and some useful capabilities not easily available in the classical languages.

Prolog:

The language was defined by A. Colmerauer (et al.) in the early 70's at the University of Marseilles. It grew at the University of Edinburgh and became widely known in the 'Edinburgh syntax' from Clocksin and Mellish's text. Prolog came to world notice in 1981 when it was selected by the Japanese government for their 'Fifth Generation Computer Systems Project'. It is often used in artificial intelligence work, as an alternative to Lisp. Prolog exists in many dialects and is not yet standardized. (This discussion uses mainly the Edinburgh syntax.)

'Prolog' is short for 'programming in logic'. Prolog programs define data relationships, but specify few procedural details. Instead, Prolog uses a generalized and powerful strategy of recursive backtracking, to discover all allowable combinations of database facts that fit the declared relationships.

.Suppose we have an aliphatic organic chemistry data-base, e.g.:

```
ketone(acetone).
ketone('2-propanone').
.....
primary_alcohol(ethanol).
sec_alcohol('2-propanol').
.....
```

Each line of the above is a typical Prolog program predicate (or database fact), ending in a period and asserting a relationship for one or more arguments. Predicate names and argument values require lower-case letters (this can be over-ruled by use of singly-quoted values, e.g. 'Br' or '2-propanol'). A predicate is not a function and does not have a value, e.g. to provide the boiling point of acetone, we would have to assert:

```
bp(acetone, '56.5Celsius').
```

(These examples are simplistic; in a useful system such predicates could be more informative, and could perhaps be generated automatically from a standard structural and reaction dataset.)

The database could then be queried with predicates containing variables (beginning with upper-case letters), e.g.

```
?- ketone(X).
...inquiry
  X = acetone      ...response
  X = 2-propanone ...another response
  .....            ...and so on
```

The database is scanned exhaustively for all values of X such that the inquiry is true. (If no match could be found, the system would respond: "NO".)

Reactions could be added to the database as assertions of general rules, e.g.

```
reduce(X, 'NaBH4', Y) :-
  ketone(X),
  sec_alcohol(Y).
```

meaning, X can be reduced by NaBH4 to Y if X is a ketone and Y is a secondary alcohol. This rule would then allow queries such as:

```
?- reduce(acetone, 'NaBH4', X)...inquiry
  X = 2-propanol          ..response
```

meaning, to what product would acetone be reduced by NaBH4? (Of course, the above reduce() predicate is a necessary but not sufficient condition; structural rules would have to be added to preclude e.g. the reduction of acetone to 2-butanol, but that is beyond the scope of this brief introduction.) The above rule could be applied in other ways, e.g.

```
?-reduce(X,'NaBH4','2-propanol').
 X = acetone
?-reduce(ketone(_), X, sec_alcohol(_)).
   X = NaBH4
```

and so on. In the last query above, the underline arguments (_) are generalized Prolog place-holders for arguments whose values are currently unimportant.

In general, Prolog rules can be written of the form:

```
p1(a1...) :- p2(...), p3(...).....
```

meaning the relation p1 is asserted for matching values of its arguments if predicates p2.... can be verified for their arguments. Of course, verification of p2.... may require further backtracking through the database. A predicate becomes an inquiry by giving one or more of its arguments as variables, instead of values. The important thing is that recursive backtracking will find every combination of argument values (explicit and implied) for which an inquiry can be verified.

Recursive backtracking can be very powerful, but can also lead to very long run-times and very large numbers of solutions (the 'combinatorial explosion') for a large database. Matching can be made more efficient by use of a 'cut' predicate, which can preclude search down some logical paths, if such further search is not wanted or unlikely to be useful.

A Prolog database is a file of

assertions and rules, in predicate form. The database is dynamic, because assertions and rules can be added, deleted and altered freely during the run of a program. Like Lisp and Snobol, Prolog programs are capable of generating and executing new code during the run, so the programs can be self-modifying and adaptive.

The above gives only a very brief taste of Prolog capabilities. Other operations are possible. Early Prologs could do only 16-bit integer arithmetic (in the range +32768), but newer versions can do rational, complex and double-precision floating-point arithmetic. Standard predicates exist to process data lists, e.g. ['F','Cl','Br','I'], trees (as lists of lists), arrays (as lists), strings (as lists of symbols) and graphs (as lists of nodes, with connections). Molecules could be coded and processed as atomic or functional-group graphs. There are also predicates to handle input, output and file manipulations, including Prolog data-base editing. Values (e.g. '2-propanol', 39.06) can be converted to and created from strings of characters. Prolog libraries are available to supply pre-written predicates for most common operations.

Prolog is particularly suited to the expression and application of logical relationships, (including fuzzy logic). In chemistry, it would be useful for exploring structure/property relationships, deducing structures from spectra, and generating chemical structures. Because it can handle logical inference, it would be usable in computer-assisted learning.

ADA Prolog (Computer Solutions).
PDC Prolog (Prolog Development Center).
Prolog (Arity).
Prolog (Cogent Software).
Prolog-86 Plus (Coders Source).
Prolog++ (Quintus Computer Systems).
tiny-Prolog (Austin Code Works).

Dr. Dobbs Journal, Mar. 1985, p.36.
Dr. Dobbs Journal, Apr. 1986, p.46.
Dr. Dobbs Journal, July 1987, p.30.

G. J. Kleywegt et al., Chemometrics Tutorials, Chapters 6, 7 (Elsevier, 1990).

W. F. Clocksin, C. S. Mellish, Programming in ˇˇˇˇˇProlog, 3rd.ed. (Springer-Verlag, 1987).

B. Filipic, Prolog Users Handbook (Wiley-Halsted, 1988).

N. C. Rowe, Artificial Intelligence Through Prolog (Prentice-Hall, 1988).

AWK:

AWK was never intended as a full language, but became one anyway. It was invented by Aho, Weinberger and Kernhigan (hence the acronymic name) as a utility for the Unix system. They wanted a 'tiny language' that could be used to write one-line programs to be used as text-filters and data-file scanners. All three of them being superb programmers, they created better than they knew. They were surprised when AWK began to be used as a serious programming language. DOS and Mac versions of AWK are now available. AWK has inspired a descendant called Perl (a sort of super-AWK).

AWK is an interpreted language. It resembles C but without the elaborate declarations and data structures. The syntax is simpler. AWK has a number of special features that makes it very versatile and powerful in scanning and manipulating text files.

The simplest AWK program has the structure:

```
{ action }
```

where 'action' is one or more AWK commands in free form layout. An input file is read, line by line, and the action clause is executed repeatedly using each line in turn as input, until end-of-file.

A more general program structure is:

```
pattern1 { action1 }
pattern2   { action2 }
.....
```

This would process the input line by line. The first line would be scanned in turn for each of pattern1, pattern2, and so on. Whenever a pattern was matched, then the corresponding action would be executed, with the current line as input. When all patterns had been applied, the next line would be read and the whole pattern-matching cycle repeated. This would continue, until all the file had been read, and all patterns had been applied to every line. It is possible to match patterns to chosen words or phrases, instead of the whole line. Patterns can contain arithmetic expressions and can test numeric values. If any pattern has no action, then the matched line is printed by default.

AWK patterns can contain 'regular expressions', e.g / hex[aey]ne/ would match 'hexane' or 'hexene' or 'hexyne', / [A-Z][a-z]+/ would match any word with an initial capital letter, and so on. Very complex patterns can be created, if de-

sired.

AWK input and output can be very easy, because of the I/O defaults. There is often no need for loops or formats. Pattern matching allows selective processing of input lines. It is possible to alter the defaults so that the unit of processing is a word, file, paragraph or page, rather than a line.

AWK data structures are very simple and flexible. There are no data declarations. Any variable can contain either numeric or text values. A numeric value is treated either as a number or as a string of numeric characters according to context. All arithmetic is floating point. Variables are automatically initialized to a zero and/or an empty string before first use. Arrays are one-dimensional and dynamic (i.e. of varying length) and can contain mixed numeric and string values. An array can be indexed with either numeric or text values, e.g. bp["acetone"] to obtain a boiling point from a bp data array. It is possible to test for use of a subscript, e.g. if ("acetone" in bp) print bp["acetone"]

AWK should not be considered an 'only language', but is a very valuable 'supplementary language'. It is useful for writing short programs, for transforming files, and for preparing file excerpts and data-base reports. Many experienced programmers now use it for 'program prototyping'. They try out algorithms in AWK, then translate the tested code into a more efficient, strongly typed language for production use. In some colleges, students are taught AWK as their first computer language, because of its simplicity and power.

Coherent AWK (Mark Williams).
Minix BAWK (Prentice-Hall).
MKS AWK (Mortice Kern Systems).
PolyAwk (Sage/Polytron).
A. V. Aho, B. W. Kernighan, P. J. Weinberger, The AWK Programming Language, (Addison-Wesley, 1988).

D. Dougherty, SED and AWK, (O'Reilly, 1991).

R. Kolstad, Unix Review, 8(5)30; 8(6)79; 8(7)44.

L. Wall, R. L. Schwarz, Programming PERL, (O'Reilly, 1991).

Forth:

The Forth language was created in the early 60's by C. H. Moore. It had its first major application in the early 70's as a data-control language for a radio-telescope. It is often used as a real-time instrument control language, but has spread to other applications. The Forth community is small but enthusiastic, with something of the 'true-believer' attitude once common among APL users. There are many dialects and two commonly used standards (Forth-79 and Forth-83). Forth is a readily extensible language, so dialects often differ greatly in the details of their floating-point arithmetic, string handling, file manipulation, numeric I/O and system interfaces.

At its heart, Forth is a postfix (or reverse polish notation) calculator language, based on the manipulation of an abstract stack. Numbers, strings and memory addresses can be pushed onto the stack. The value currently at the stack-head can be inspected and removed. Data can also be moved between the stack and addresses in memory, and this in turn allows the definition of named variables, arrays, matrices and strings. All such data structures are manipulated through the stack. This insistence on a stack-based operation gives Forth its characteristic style, very different from most other languages.

In the following one-line examples, entered numbers are pushed onto the stack top when encountered. Each operator acts on the one or two numbers at the stack top; the operands are popped from the stack and the result is pushed back onto the stack; e.g.

```
.                   ...pop and print.
3 5 + .             ...print (3+5)
3 5 + 2 /     ....print ((3+5)/2).
DUP     ...duplicate stack-top.
3 5 + DUP * .     ...print (3+5)2.
```

Named variables are created by reservation of an addressed memory segment, then values can be assigned to and read from the named address, e.g.

```
CREATE IC 2 ALLOT ...IC = 2-byte integer.
  3 5 + 2 / IC !...assign IC = (3+5)/2.
  IC @ .            ...access and print IC.
```

Arrays can be handled in the same way, but by reserving multibyte memory space. e.g.
CREATE AC 20 ALLOT
...AC = 10-integer array.
Manipulation of array elements requires explicit array-element address arithmetic, but usually there are library commands designed for this.

Forth can very readily be extended to supply capabilities lacking in the primitive language. Each of the operators we have seen above (such as . + DUP and ALLOT) are called 'words'. A word is the name of a pointer (i.e. a memory address) to a brief segment of machine-code. The machine code segments use stack-top and memory-addresses as operands. New words can be

readily defined, e.g.

```
: SQUARE DUP * ;        ...define 'square'.
: CUBE DUP SQUARE * ;   ...define 'cube'.
12 CUBE .               ...print 123
```

Both of these new words operate on the stack-top.

As a result of the 'word'-based structure, a Forth program is a sequence of pointers (or jumps) to brief primitive segments of machine-code . This is called 'threaded code'. Forth programs do not need to be compiled, and yet are much faster in execution than interpreted languages. Because the primitive word-segments are accessed by address, they do not need to be copied each time they are supplied to a program sequence, so Forth programs are usually very much more compact than the equivalent machine-code created by a compiler. The word-based organization lends itself very well to program development by a bottom-up approach (rather than the more widely used top-down approach of functional programming).

Forth resembles assembly language in many of its operating details and its efficiency of execution, and yet it is high-level and transferable. At first sight, Forth source-code seems dense and cryptic, but it becomes intelligible on acquaintance. Its compactness and efficiency make Forth a valuable language for numerical real-time instrument control with small computers. Forth has string and text processing capabilities and has been used as an implementation language for interpreters, compilers and text-editors.

If you consider acquiring a Forth system, pay great attention to the systems-library of defined words, to ensure that it has the range of capabilities that you need. The Kelly and Spies text covers most of the principal Forth dialects and has an good bibliography of Forth texts and sources of software and information.

M. G. Kelly, N. Spies, Forth: A Text and Reference, (Prentice-Hall, 1988).

L. Brodie, Starting Forth, (Prentice-Hall, 1981).

HOW I USE COMPUTERS IN MY LABORATORIES

by Edward Kelly
Chemistry Department
Marian College
3200 Cold Spring Rd.
Indianapolis, IN 46222

Marian College is a small, liberal-arts college on the northwest side of Indianapolis. Despite limited resources there has been progress in developing the use of computers in my courses.

The first microcomputers purchased by the school were TRS-80, models III and IV. Eventually they ere replaced by Apple and IBM computers. The Chemistry Department was offered full-time use of the TRS-80 computers. While not state-of-the-art, the TRS-80 has proved to be reliable and useful for introductory interfacing experiments. Finding a suitable interfacing circuit was my first problem. Fortunately I found one while attending a Chautauqua Short Course (Microcomputers as Laboratory Tools, Rex Berney, University of Dayton, 1986). For about $25 per computer I was able to interface the TRS-80 computers.

There is a strong emphasis in my labs on data collection, the analysis of that data and the interpretation of the results. The role of the computer in the lab to do the collection and assist with the analysis of data has been given high priority.

The lab manual (Inquiries Into Chemistry, Michael Abraham and Michael Pavelich, 2nd ed., Waveland Press, Inc., 1991), used for the first chemistry course for science majors, General Inorganic Chemistry, uses the discovery approach. Discovery experiments have the students collect data, graph the data, obtain an algebraic equation and "discover" the appropriate chemical principle. During the first semester the students do their own graphing to develop their technique; during the second semester the students use Graphical Analysis II (Vernier Software, 2920 W. 89th St., Portland, OR 97225) to learn how to use the computer to do their graphing. They always do a Least-Squares Analysis to find the best straight line through their data points; several experiments require a transformation of the data to obtain the desired straight line.

There are two types of discovery labs: guided inquiry (directions are given for the experiment, but only minimal directions are given for the analysis of the data) and open inquiry (only a statement of the problem to be investigated is given). Several of these labs have been modified to allow the use of a computer to collect and analyze the data. Additional interfacing labs have been developed and incorporated into the