the I.P. Sharp newsletter

MARCH-APRIL 1976

Inspector

A Computer-based Microfilm Retrieval System

This package was developed for use in conjunction with the Kodak Recordak Microstar Microfilm Reader (PR-1) and Image-Control interface (IC-5). It automates the laborious and sometimes error-prone procedures necessary to identify all those microfilm images which have specific attributes. Examples of applications would include mug-shots, library records, tax records, engineering records, etc.

English (or French, German, Spanish) commands are given to *INSPECTOR*, which then searches a central microfilm library. Search criteria can be as simple or complex as desired, for instance (in the case of a mug-shot data base),

SEX IS MALE

or

SEX IS MALE AND AGE IS 28 TO 32
AND (HAIR-COLOR IS BROWN
OR DARK-BLONDE)

The *INSPECTOR* system searches millions of frames of microfilm data, on any desired criteria, to locate all relevant information. Selected images can then be displayed on the reader and hard-copy images made (in about 15 seconds) if desired. One or several of the frames of the film on view may be selected for printing. The system is interactive, and updates are stored that are automatically displayed when the "old" frame is referenced.

Please contact your local Sharp representative for further information regarding this package.

ACT - ACTUARIAL PACKAGE

by David Crossley

ACT originated in 1971 or thereabouts, when D.R.W. Jamieson, FSA, of Sun Life, put together a set of Actuarial functions in *APL*, having designed a notation in *APL* that was very easily related to the actuarial halo notation. Since then, the *SHARP APL* package has been improved in numerous ways and only the notation is in common with the original package. Briefly, the changes are:

1. All of the 200 or so functions have been rewritten and some new ones have been added. As a result, many functions are more efficient, there are fewer function calls, and arguments may be arrays as well as vectors.

2. The package can be applied to situations involving variable interest bases, select-and-ultimate mortality tables, and joint lives.

3. There is now a file of mortality tables, projection scales and related data. There are currently about 100 tables in the constantly growing file.

The package is now considerably more versatile and easier to use. Even non-actuarial users may find some use for the group of compound interest functions. No package is ever complete, and certain applications are yet to be included. In particular, a better treatment of single and joint-life annuities and multiple-decrement applications, is desirable.

If the aim at one company where I worked, was "to put oneself out of a job", I succeeded so well that I was asked to leave. Now at the completion of this stage of development of the ACT package, I hope I have interpreted the dictum correctly.

The manual: "The *SHARP APL* Actuarial Package" is available from your local *SHARP APL* representative.

NEW PUBLICATIONS

The *SHARP APL* Reference Card - completely revised.
Manuals: The *SHARP APL* Actuarial Package.
X-11 in *SHARP APL*.
The *SHARP APL* Library Catalogue, to be released in May, consists of:
- a short introduction describing maintenance and structure,
- a brief description of each workspace or group of related workspaces,
- an index by library number
- an index by workspace name
- an index by application.

SHARP APL TECHNICAL NOTES - *SATN*-5 - Batch *APL*.
- *SATN*-8 - High speed printing of files.
- *SATN*-10 - Sorting of *SHARP APL* files.

APL ECONOMICALLY - Cheaper by the Batch.

BTASKS (batch *APL* tasks) are now available to users of the *SHARP APL* system. In contrast to *NTASKS*, which are no-terminal tasks created by the user via the □*RUN* function, *BTASKS* are created by the *BTASK* system scheduler from a request submitted by the user. Therefore, a user's *BTASK* request does not actually initiate the task.

Batch *APL* tasks share many of the applications of *NTASKS*. They are particularly applicable to packages that do a fair amount of processing with little user interaction. Unlike *NTASKS*, however, *BTASKS* can optionally be restarted after a system failure or shutdown. Execution costs of *BTASKS* are also less than those of *NTASKS*, as the tasks tend to run during off-peak hours.

BTASKS may conveniently issue requests for high-speed file prints, file sorts, or other batch tasks. *BTASK* requests are submitted through the functions available in workspace 1 *BTASKREQ*. The use of these functions is described in detail in *SATN*-5, entitled "*BATCH APL*". The *SATN* also discusses how *BTASKS* are scheduled, and some of the problems associated with restarting them. Much of what is said about restartability can equally well be applied to *TTASKS* and *NTASKS*.

A copy of *SATN*-5 is available from workspace 1 *SATN*, or from your local *SHARP APL* representative, who will be happy to discuss how well suited your application is to *APL* subtasks.

SHARP APL CROSS-TABULATION

Analysis of questionnaires has been one of the greatest stumbling-blocks in many market research studies. These studies, carried out by many large companies, relate current demand to actual production and future consumption.

I.P. Sharp Associates now offers a cross-tabulation package to analyse both large and small questionnaires. This analysis can be done easily and quickly - system involvement is minimal, so knowledge of *APL* is not necessary, nor is there any problem of coding and interpretation. Questionnaires can be sampled question by question, up to eight levels of searching, for instance a four-level search of sex versus age versus nationality versus area of residence, etc.

The user may request category analysis such as A and B, but not C or D. Frequency counts are produced with row, column and cellular percentages and different statistics such as chi-squares. Searches of associated files can also be accomplished using a very simple program.

Now that *SHARP APL* offers a batch facility, this system also incorporates low-cost processing. Please contact Isaac Ehrlich in the Toronto office, or your local Sharp representative, for further information.

INTELLIGENT COMMUNICATIONS

We are currently in the midst of a major upgrade of our communications network. Up until now we have relied on time division multiplexing and of course in Canada on Dataroute. These facilities are in the process of being replaced by our own internally developed packet switching system. The principal advantage from the user's point of view will be an improvement in the quality of communications lines, since data will be re-transmitted automatically in the event of an error. From our point of view we are able to get better line utilization through the use of statistical multiplexing.

Each major city in our network will be equipped with a Computer Automation Alpha-16 minicomputer which receives all incoming local calls. The Alpha assembles the packets of data from each attached terminal and transmits the composite data on a high speed synchronous circuit directly into the IBM 3705 front end processor. Since we operate a private network and support only *APL* we are able to optimize the packet length to balance line-loading with acceptable response time.

The Intersystems (Alpha) software development has been carried out in Amsterdam by Michael Harbinson and his crew and the complementary 3705 software has been the responsibility of Roger Moore.

The first major node will be established in Rochester N.Y., by May and the second in London, England, shortly thereafter. We intend to complete the installation of the intelligent network during this calendar year.

COMMUNICATIONS SYSTEM ERROR MESSAGES

by Roger Moore

With the installation of the 3705 front end computer, the failure mode of the *SHARP APL* system was changed considerably. The 3705 is almost always operational and prepared to answer an incoming call regardless of whether or not *APL* is capable of servicing the terminal. The 3705 is connected to the 360/75 through a Byte Multiplexor Channel. The Byte Multiplexor Channel consists of nearly 200 Multiplexor Channel Addresses, each of which can support one user. After a user attempts to sign on to *APL*, a 3705 generated error message may be printed at the terminal.

AWAITING FUNCTION FOUR indicates that the 3705 is waiting for an operator command to reconnect it to the 360/75 (after a system reset or a 3705 reload). The operator signals reconnection by selecting the proper 360/75, setting a console knob to function four and pushing the 3705 interrupt key.

APL PROBABLY DOWN is the 3705's guess as to the state of the 360/75 immediately after connection to the byte multiplexor channel. In particular when a byte multiplexor channel address is available for a new user, *APL* signifies this by sending an *ENABLE* command to the 3705 using that channel address. The 3705 procrastinates a response to the *ENABLE* command until a user attempts to connect to *APL*.

The *APL PROBABLY DOWN* message is printed when all idle multiplexor channel addresses are command free.

APL PROBABLY HUNG indicates that none of the assigned channel addresses are usable but that unlike *APL PROBABLY DOWN* some of these addresses are in a transient state. This transient state suggests that the 360/75 is no longer servicing multiplexor channel interrupts.

APL DEFINITELY HUNG indicates that the 3705 found a suitable multiplexor channel address and attempted to claim it for some user, but did not receive acknowledgement from *APL*. This problem can theoretically be caused by resource exhaustion in the 360/75.

NO MPX ADDRESS AVAILABLE indicates that all of the multiplexor channel addresses allocated to the 3705 are in use and that no other user can be connected via that 3705 until some current user signs off.

NO PRB AVAILABLE or
NO BUFFER AVAILABLE indicate that a 3705 resource has been exhausted. These messages are most likely to occur when the 3705 is busy printing *AWAITING FUNCTION FOUR* on many other terminals. The 3705 resource is likely to be available in five or ten seconds and so a retry might be worthwhile (users are contending for these resources at every input line).

SYSTEM RESET DETECTED will be printed once when the 3705 detects a channel reset. The channel reset can be caused by automatic error recovery in the byte multiplexor channel or by operator action. It usually indicates that *APL* will be down for at least ten minutes while the 360 is reloaded and workspace crash recovery is performed.

PATIENCE PLEASE originates in the *APL* signon processor in the 360/75 and is a synonym for *NUMBER IN USE*. It indicates that the signon password has been entered correctly but that the signoff for the previous session has not completed due to queueing problems (such as attempting to untie files). If the condition does not clear within a few minutes it is advisable to inform the *APL* operator.

Later this year, the time division multiplexors in some large sites will be replaced by Intersystems concentrators. The concentrator is a stored program computer like the 3705 and is also capable of printing error messages on the user's terminal.

SYSTEM FULL indicates that the 3705 is operational but not capable of accepting any more calls from the concentrator network. The 3705 has returned a "busy" signal to the concentrator which then prints this message. This condition will not clear until some current user of the network signs off.

LINES DOWN indicates that the concentrator is unable to reach the 3705 due to communication line failure.

P.S. Users of 2741's can avoid further confusion by beginning the first input line with a right parenthesis, ")" instead of a naked carriage return, so that *AWAITING FUNCTION FOUR* will not print as "*P[PO←OSA CTSX←OQS CQTN*" or "*GKGYUYR, QERFUYIR QIES*".

SHARP APL COURSES

Introductory courses:

DALLAS	April 19-23	May 17-21	June 21-25
LOS ANGELES	April 5-9	May 3-7	June 7-11
MONTREAL		May 17-21	
OTTAWA	April 5-9	May 3-7	June 7-11
ROCHESTER	April 19-23	May 17-21	June 21-25
TORONTO	April 19-22	May 10-13	June 14-17
LONDON, ENGLAND	April 21-23,26,27		June 2-4,7,8

Intermediate courses:

TORONTO	April 26-28		June 28-30

Seminars:
Advanced *APL* - two days

TORONTO	April 20,21	May 11,12	June 15,16

Commercial Applications of *APL* - one day

LONDON, ENGLAND	April 13
MANCHESTER, ENGLAND	April 6

DUESSELDORF, WEST GERMANY

We are pleased to announce the opening of the newest Sharp office, overlooking the river Rhine:

I.P. Sharp GMBH, - telephone (0211) 579084
Kaiser-Friedrich-Ring 98,
4000 Duesseldorf,
West Germany.

Please note the following new addresses:

OTTAWA:

I.P. Sharp Associates Limited,
Suite 600,
265 Carling Ave.,
Ottawa, Ontario K1S 2E1.
(613)236-9942

BOSTON:

I.P. Sharp Associates, Inc.,
Suite 812,
148 State Street,
Boston, Mass 02109.
(617)523-2506

Technical Supplement

by Clement Kent

EXECUTE (⍎), ⎕*LX* and *RESTARTING TASKS*

Some misunderstandings have occurred about how to use ⎕*LX* to restart *TASKS*. In particular, one of the most common questions seems to be: "How can I write a *RESTART* function so that the expression →*RESTART* works, both when typed in immediate execution mode at the terminal, and when executed as ⎕*LX* upon loading a *CONTINUE* workspace?" This problem requires a careful analysis of several cases.

First of all, one must understand the nature of ⎕*LC* and it's relation to the)*SI*. ⎕*LC* is a vector which is the numeric equivalent of the)*SI* (which can be obtained as a literal matrix from 2 ⎕*WS* 2). There is one entry in ⎕*LC* for every function line, ⍞, ⎕, and ⍎ in the)*SI*. ⍞, ⎕, and ⍎ show up in ⎕*LC* as 0, and thus cannot be distinguished from each other except by reference to 2 ⎕*WS* 2. If ⎕*LC* is evaluated on line *N* of a function, ⎕*LC*[1] will be *N*. If evaluated by an expression such as ⍎'⎕*LX*', the first member will be 0.

Thus, if typed in immediate execution mode, the expression →⎕*LC* either

1) does nothing if the)*SI* is empty, or

2) restarts execution of the function at the top of the)*SI* on the line ⎕*LC*[1].

Now, if we want *RESTART* to return a result line so that →*RESTART* is equivalent to →⎕*LC*, we must do something like:

```
RESTART[1]  LINE←1↓⎕LC
```

to eliminate the entry in ⎕*LC* for line [1] of *RESTART*.

But now, what happens if ⎕*LX*←'→*RESTART*' and we load a workspace with this latent expression? Automatically, ⍎'→*RESTART*' is done. Suppose *CALCULATE* was suspended on line 5. Then, when we are on line 1 of *RESTART*, the first three entries in the)*SI* and ⎕*LC* are:

```
)SI                ⎕LC
RESTART[1]         1
⍎                  0
CALCULATE[5]       5
```

Now the expression 1↓⎕*LC* yields 0 5, so ⍎'→*RESTART*' is the same as ⍎'→0' which does not restart line 5 of *CALCULATE*.

One's first reaction to this problem might be to change *RESTART* as follows:

```
RESTART[1]  LINE←1↓⎕LC ◇ LINE←(LINE>0)/LINE
```

This deals nicely with both of the above cases. However, it fails miserably in the case described below. Suppose *CALCULATE* contains the line:

```
CALCULATE[6]   'PRINT REPORT?' ◇ INP←⍞
```

If the line dropped while we were in ⍞ mode, waiting to reply to this question, the state of the *CONTINUE* workspace would be:

```
2 ⎕WS 2              ⎕LC
⍞                    0
CALCULATE[6]         6
```

As documented in *SATN*-7, upon loading the workspace the ⍞ is changed into an execute of '⍎⎕*LX* ◇ ⍞'. It is as if the function line had become *CALCULATE*[6] 'PRINT REPORT?' ◇ *INP*←⍎'⍎⎕*LX* ◇ ⍞'

According to *SATN*-6, an executed expression has the value of the last statement in that expression, which in this case is the ⍞. Thus, the result of ⍎'⍎⎕*LX* ◇ ⍞' is the same as the result of ⍎'⍞', is the same as ⍞. Therefore we seem to have preserved the meaning of line 6 while allowing our latent expression to retie files, etc. This is surely desirable. However, notice that in this case,

```
R←⍎'⍎⎕LX ⋇ ⍞'   ↔  R←⍎'⍎''→RESTART'' ⋇ ⍞'
                ↔  R←⍎'⍎''→6 ⋇ ⍞'
```

Now *SATN*-6 tells us that branching to a positive line number within an executed expression is only permissible if the expression is the leftmost thing on its function line. Thus,

```
FOO[1]  →6        or
FOO[1]  ⍎'→6'     or  ⍎'→6 ◇ ⍞'  or
FOO[1]  ⍎'⍎''→6''◇ ⍞'
```

are all valid lines, but

```
FOO[1]  R←⍎'→6 ◇ ⍞'  or R←⍎'⍎''→6'' ◇ ⍞'
```

are not.

So our ⎕*LX* fails to provide the correct branch statement, and in fact produces a syntax error.

Let's examine what the state indicator looks like in this case, when we are on the first line of *RESTART*:

```
2 WS 2               ⎕LC
RESTART[1]           1
⍎                    0
⍎                    0
CALCULATE[6]         6
```

What happens if we branch to 0 in our *□LX*?

```
R←⍞'⍞□LX ◇ ⍞'   ↔   R←⍞'⍞''→0'' ◇ ⍞'
                ↔   R←⍞'⍞'
```

which is what we wanted. So we can use

```
RESTART[1]  LINE←1↓□LC ◇ SI←1 0↓2 □WS 2
       [2]  LINE←('⍞'=1↑,SI)↓LINE
```

This function now works in all of the above cases.

REPORT GENERATION and *□OUT*

Unless you need the report yesterday, the best way to produce it is not to beat your terminal to death. In this section we explore the advantages of alternate modes of report generation.

As the canonical example, let us use one of the most common ways of generating output:

```
(FORMAT STRING) □FMT (ARGUMENT LIST)
```

In this example we will use the following variables:

```
FS←'15A1,X5,10BCF10.2'
R←50 15⍴15↑ROWNAMES'
DATA←50 10⍴(500?10000000)÷100
FS □FMT (R;DATA)
```

```
ROWNAMES        51,941.64 83,096.54  3,457.   684.23 41,748.60
ROWNAMES        68,677.28 58,897.67 93,043   ,119.06 91,032.09
ROWNAMES        76,219.81 26,245.30  4,74   36,533.87 24,703.89
ROWNAMES        98,255.03 72,266.05 75,3    43,641.15 76,649.48
ROWNAMES        47,773.18 23,777.45 27   2  6,056.44 90,465.31
ROWNAMES        50,452.29 51,629.20 3    43  7,374.91 50,070.71
ROWNAMES        38,414.22 27,708.19  1.43 77,020.46 82,781.73
```

If this table is printed on the terminal, it involves printing up to 50×120=6000 characters. In addition, there is the CPU cost of executing the format statement, which is approximately:

```
⍝<TIME> RETURNS THOUSANDTHS OF A CPU UNIT PER EXECUTION

20 TIME 'Y←FS □FMT (R;DATA)'
660
```

Finally, the printing of the table would take about 3 minutes on a 30 cps terminal without the tabs set (it could take as little as 1 minute with tabs set to 5, or as much as 6-7 minutes on a 2741 type terminal).

Summarizing these results:

1) - Cost of printing table for non-tabbing 30 cps terminal

	CONNECT	*CPU*	*CHARACTERS*	*TOTAL*
APPROX. COST ($)	.40	.25	2.00	2.65

Setting tabs could cut connect and character charges by up to 50%.

2) - Cost of printing the table on 30 cps terminal with *)TABS* 5

	CONNECT	*CPU*	*CHARACTERS*	*TOTAL*
APPROX. COST ($)	.20	.25	1.00	1.45

Clearly it would be more economical to move to a report generation procedure that used the file printing facilities available in *SHARP APL*. Let us assume that the report we are printing is a fairly long one, say 25 pages or more, so that the $2.50 minimum charge is not greater than the charge of 10 cents a page. Then, the cost of printing the report is (in the simplest case - no special forms, etc.) 10 cents a page plus the cost of CPU time used to generate the print file. This is:

```
     20 TIME '(FS ⎕FMT (R;DATA)) ⎕APPEND OUTFILE'
720
```

3) - Cost of printing the table via *FILEPRINT*:

	OVERHEAD/PAGE	*CPU*	*TOTAL*
APPROX. COST ($)	.10	.25	.35

This is 4 times cheaper than even the cheapest terminal printed report using tabs. However, some common large reports using tables run to several hundred pages, and for these reports reducing the CPU cost can save a lot of money.

Here is where the use of *⎕OUT* is important. To quote *SATN-3*: "If *⎕FMT* is the only expression on the line, then the conversion of its arguments to a character matrix is not done for the *⎕OUT* file. Instead, a control message is appended to the file.... " This means that if you do not use the result of the *⎕FMT* statement in any way (if it is the only expression on the line), then the result need not be generated, with the attendant saving on CPU.

```
     X←⎕OUT 0 1
     T←100 TIME1 'FS ⎕FMT (R;DATA)'
     X←⎕OUT 1 0 ◇ ⎕←T
44.27
```

The *TIME1* function above executes the expression in its right argument using a 3 *⎕FD* - defined local function (see August/September 1975 Newsletter), rather than *⍎'EXPRESSION'* - because any expression containing *⎕FMT* which could yield a result causes the result of *⎕FMT*

to be calculated. All of these expressions cost about the same:

```
Y←FS ⎕FMT (R;DATA)
⍞'FS ⎕FMT (R;DATA)'
(FS ⎕FMT (R;DATA))
```

even with ⎕*OUT* set to 0 1, but this one

```
FS ⎕FMT (R;DATA)
```

is much cheaper. Note that so long as we are not printing the report on the terminal, we could run the report program as an *NTASK* or *BTASK*.

4) - Cost of *HSPRINT* via ⎕*OUT* generated files.

	OVERHEAD/PAGE?	*CPU*	*TOTAL (APPROX)*
TTASK	.10	.015	.12
NTASK	.10	.011	.11
BTASK	.10	.088	.11

This final method reduces report generation to an irreducible minimum - which is more than 25 times cheaper than simply printing the report on the terminal.

Questions, comments and contributions are welcome and should be addressed to

Clement Kent
(mailbox *KENT*)
I.P. Sharp Associates,
Toronto.

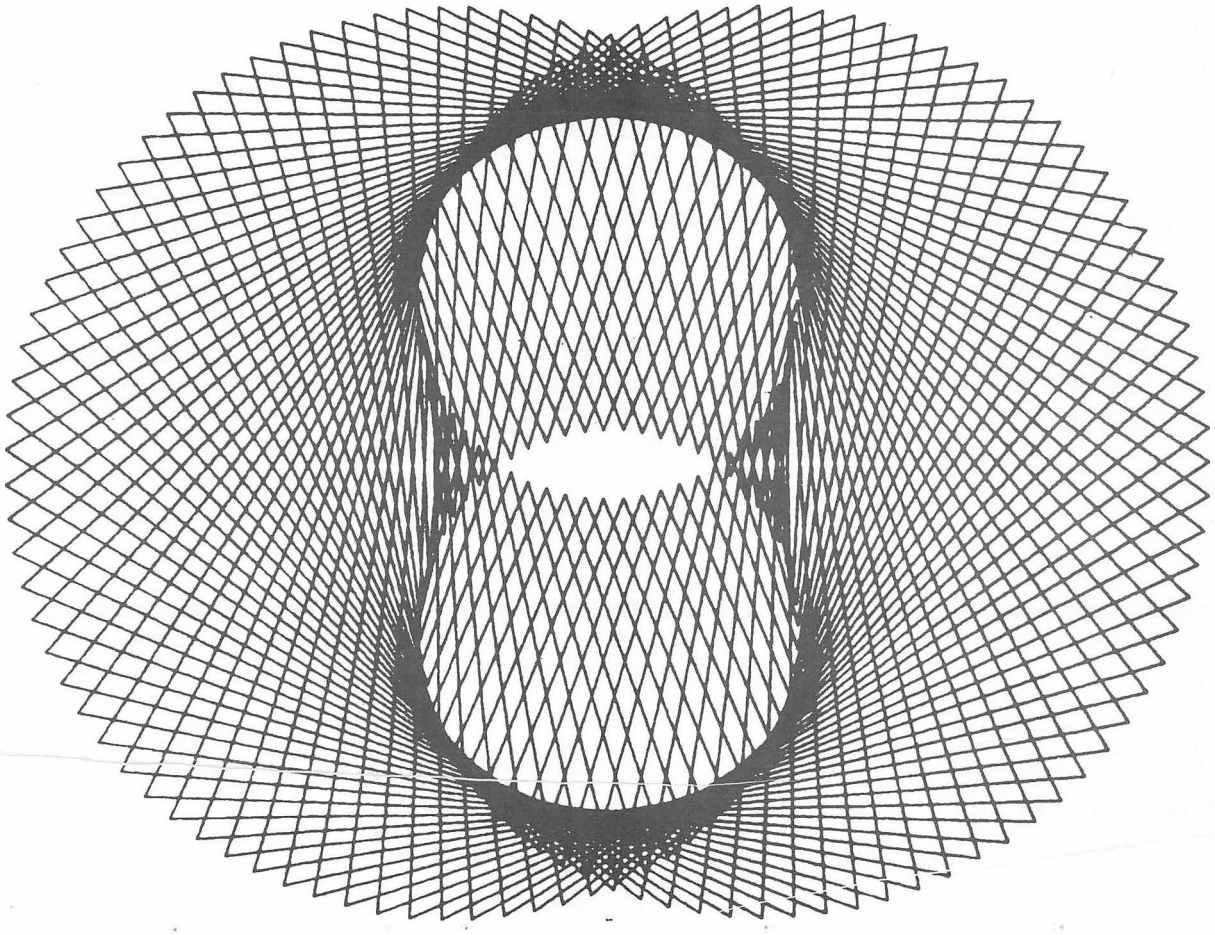
CORRECTION!

Please accept our apologies for the error on page 1 of the January-February Technical Supplement.

AUTO-RESTART - Line one of all programs should read:

```
⎕LX←'RESTART ⋆ →1↓⎕LC'
```

to automatically restart the *CONTINUE* workspace after a drop with the appropriate file re-tieing and branch commands.



STARK & CLEVER APL

Some people call it a "scientific" language. Some people call it a "mathematical" language. Some people are most struck by its use for interactive systems, so to them it's an interactive language. But most of us just think of it as THE LANGUAGE WITH ALL THE FUNNY SYMBOLS, and here they are:

```
⋆ρ∪∩αc÷ω↑ε|∧\⊃⎕;⌈∘∇'∆
¨¯<≠≤≥=>)∨:⊥_(↓⊤→ι~○?⌊-
12384657]9.BF[UN←ITOQD+
PRVCAZ×WYEMO/XL·,SJGKH
```

Enthusiasts see it as a language of inconceivable power with extraordinary uses. Cynics remark that it has all kinds of extraordinary powers for inconceivable uses-- that is, a weird elegance, much of which has no use at all, and some of which gets in the way.

This is probably wrong. APL is a terrific and beautiful triumph of the mind, and a very useful programming language. It is not for everybody, but neither is chess. It is for bright children, mathematicians, and companies who want to build interactive systems but feel they should stick with IBM.

APL is one of IBM's better products, probably because it is principally the creation of one man, Kenneth Iverson. It is mainly run on 360 and 370 computers, though implementations exist for the DEC PDP-10 and perhaps other popular machines. (Actually Iverson designed the language at Harvard and programmed it on his own initiative after moving to IBM; added to the product line by popular demand, it was not a planned product and might in fact be a hazard to the firm, should it catch on big.)

APL is a language of arrays, with a fascinating notation. The array system and the notation can be explained separately, and so they will.

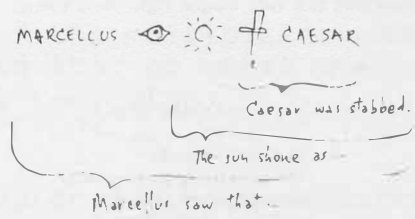
Let's just say the language works on things modified successively by operators. Their order and result is based upon those fiendish chicken scratches, Iverson notation.

THAT NIFTY NOTATION

The first thing to understand about APL is the fiendishly clever system of notation that Iverson has worked out. This system (sometimes called Iverson notation) allows extremely complex relations and computer-type events to be expressed simply, densely and consistently.

(Of course, you can't even type it without an IBM Selectric typewriter and an APL ball. Note the product-line tie-in.)

The notation is based on operators modifying things. Let's use alphabetic symbols for things and play with pictures for a minute.



In considering the successive meanings of this rebus we are proceeding from right to left, as you note, and each new symbol adds meaning. This is the general idea.

You will note, in this example, the curious arrangement whereby you can have several pictures, or operators, in a row. This is one of the fun features of the language.

Reprinted from:

COMPUTER LIB

©1974 Theodor H. Nelson.
All rights reserved.

by permission.

TWO-SIDED OPERATORS

In old-fashioned notations, such as ordinary arithmetic, we are used to the idea of an operator between two things. Like

2 + 2

or in algebra,

x X y

These, too, occur in APL; indeed, APL can also nest two-sided operators-- that is, put them one inside the other, like the leaves of a cabbage. Old-fashioned notations nest with parentheses. But APL nests leftward. It works according to a very simple right-to-left rule.

x X y X 2 + 2

the result of this
is operated on by
the next thing and operator,
yielding another result
which is in turn operated on by
the next thing and operator,
yielding final result.

ONE-SIDED OPERATORS

We are also used to some one-sided operators in our previous life. For instance:

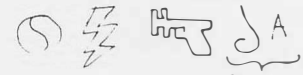
- 1

means the negation of 1;

- (- 1)

means negating that.

APL can also nest one-sided operators.



first operator is
applied to A;
result is worked on
by second operator;
result is worked on
by third operator;
result is worked on by
fourth operator,
yielding final result.

SAME SYMBOLS WORK BOTH WAYS

Now, one of the fascinating kickers of APL is the fact that most of the symbols have both a one-sided meaning and a two-sided meaning; but, thank goodness, they can be easily kept straight.

Here is a concrete example: the symbol ⌈ or "ceiling." Used one-sided, the result of operator ⌈ applied to something numerical is the integer just above the number it is applied to: ⌈7.2 is 8. Used two-sided, the result is whichever of the numbers it's between is larger: 10⌈6 is 10. (There is also ⌊, floor, which you can surely figure out.)

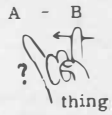
Now, when you string things out into a long APL expression, Iverson's notation determines exactly when an operator is one-sided and when it is two-sided:

As you go from right to left,

another thing?
another op?
OP THING

you generally start with a thing on the right. Then comes an operator. If the next symbol is another thing, then the operator is to be treated as a two-sided operator (because it's between two things). If the object beyond the first operator is another operator, however, that means APL is supposed to stop and carry out the first operator on a one-sided basis. Example:

A - B



thing,
op,
thing.

Conclusion:
It's two-sided.
Interpretation:
"subtract B from A."

A + - B

thing,
op,
op--
stop.

Conclusion:
The first operator
is one-sided.
Interpretation:
"negate B."
Then take next symbol.

Here is another example showing how we chug along the row of symbols and take it apart. Again, the alphabetical entities represent things.

first operation (one-sided)
second operation (two-sided)

Try dividing up these examples:

ELEANOR SAM SUSIE

One more thing needs to be noted. Not only can we work out the sequences of operations, from right to left, between the symbols; the computer can carry them out in a stable fashion. Which is of course essential.

INSIDE

The truth of the matter is that APL in the computer is a continuing succession of things being operated on and replaced in the work area.

first thing
. . . UG ϕ YARGH
thing which results
from operator ϕ
done on YARGH,
thing that results from operation
done to that by UG

and so on.

What is effectively happening is that the APL processor is holding what it's working on in a holding area. The way it carries out the scan of the APL language, there only has to be one thing in there at a time.

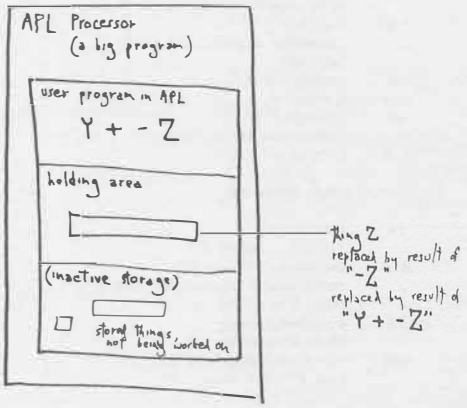
Suppose we have a simple user program,

```
Y + - Z
```

Starting at the right of this user program, the main APL program puts Z into the work area. That's the first thing. Then, stepping left in the user program, the APL processor follows the rules and discovers that the next operation makes it

```
- Z
```

which happens to mean, "the negation of Z." So it carries this out on Z and replaces Z with the result, -Z. Then, continuing to scan leftward, the APL processor continues to replace what was in the work area with the result of each operation in the successive lines of the user program, till the program is completed.



A WEIRD EXAMPLE, TO HELP WITH THE NOTATION.

Just for kicks, let us make up a notation having nothing to do with computers, using these Iverson principles:

1) If an operator or symbol is between two names of things, carry it out two-sidedly. If not, carry it out one-sidedly.

2) Go from right to left.

The best simple example I can think of involves file cards on the table (named A, B, C...) and operators looking like this:

```
0⟩  45⟩  90⟩  180⟩  45⟩  90⟩  180⟩
```

to which we may assign the following meanings:

ONE-SIDED: ROTATION OPERATORS

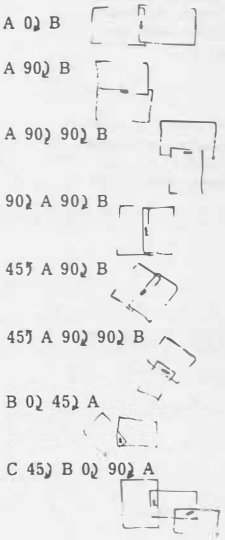
0⟩ A	do nothing to A
45⟩ A	rotate A clockwise 45°
90⟩ A	rotate A clockwise 90°
	etc.

TWO-SIDED: STAPLING OPERATORS

B 45⟩ A	staple A (thing named on the right) to B (thing named on the left) at a position 45° clockwise from middle of B's centerline.

And equivalently for other angles.

Now, using these rules, and letting our things be any file cards that are handy, here are some results:



WOOPS! One of these examples is wrong! Can you find it?

It's hard to believe, but there you are. This notation seems adequate to make a whole lot of different stapled patterns.

Exercise! Use this nutty file card notation to program the making of funny patterns. Practice with a friend and see if you can communicate patterns through these programs, one person uncomprehendingly carrying out the other's program and being surprised.

The point of all this has been to show the powerful but somewhat startling way that brief scribbles in notations of this type can have all sorts of results.

SOME APL OPERATORS

It would be insane to enumerate them all, but here is a sampling of APL's operators. They're all on the pocket cards (see Bibliography).

For old times' sake, here are our friends: (And a cousin thrown in for symmetry.)

+A	plain A (whatever A should happen to be)
A+B	A plus B (whatever A should happen to B, heh heh)
-B	negation of B
A-B	A minus B
×B	the sign of B (expressed as -1,0 or 1)
A×B	A times B

And here are some groovies:

!A	factorial A (1×2×3 ... up to A)
A!B	the number of possible combinations you can get from B, taken A at a time
?A	a random integer taken from array A
A?B	take some integers at random from B. How many? A.

But, of course, APL goes on and on. There are dozens more (including symbols made of more than one weird APL symbol, printed on top of each other to make a new symbol).

Consider the incredible power. Single APL symbols give you logarithms, trigonometric functions, matrix functions, number system conversions, logs to any arbitrary base, and powers of e (a mysterious number of which engineers are fond).

Other weird things. You can apply an operation to all the elements of an array using the / operator: +/A is the sum of everything in A, ×/A is the combined product of everything in A. And so on. Whew.

As you may suspect, APL programs can be incredibly concise. (This is a frequently-heard criticism: that the conciseness makes them hard to understand and hard to change.)

MAKE YOUR OWN

Finally and gloriously, the user may define his own functions, either one-sided or two-sided, with alphabetical names. For instance, you can create your own one-sided operator ZONK, as in

ZONK B

and even a two-sided ZONK,

A ZONK B

which can then go right in there with the big boys:

A ⌽ ZONK ⍳↓ B

Don't ask what it means, but it's allowed.

APL THINGS, TO GO WITH YOUR OPERATORS

As we said, APL has operators (already explained) and things. The things can be plain numbers, or Arrays (already mentioned under BASIC). Think of them as rows, boxes and superboxes of numbers:

```
2 4 6 8 10          a one-dimensional thing

2 4
3 5                 a two-dimensional thing

1 5
6 8                 a three-dimensional thing,
                    seen from the front. Maybe
                    we better look at the levels
                    side by side:
                    1 3     2 4
                    5 7     6 8
```

APL can have Things with four dimensions, five and so on, but we won't trouble you here with pictures.

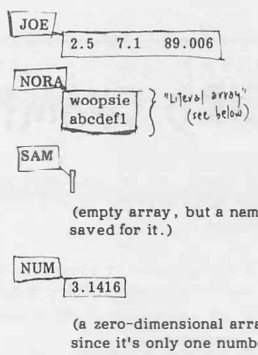
Oh yes, and finally a no-dimensional thing. Example:

```
75.2
```

It is called no-dimensional because there is only one of it, so it is not a row or a box.

Seriously, these are arrays, and Iverson's APL works them over, turns them inside out, twists and zaps through to whatever the answers are.

As in BASIC and TRAC, the arrays of APL are really stored in the computer's core memory, associated with the name you give them. The arrays may be of all different sizes and dimensionality:



Each array is really a series of memory locations with its label and boxing information-- dimensions and lengths-- stored separately. One very nice thing about APL is that arrays can keep changing their sizes freely, and this need be of no concern to the APL programmer. (The arrays can also be boxed and reboxed in different dimensions just by changing the boxing information-- with an operator called "ravel.")

Copies of the book are $7.00 postpaid from: Hugo's Book Service
Box 2622
Chicago, Illinois 60690
U.S.A.

Package of 10 copies $50.00 postpaid.

SNEAKY REPEATER STATEMENT IN APL?

One of the APL operators, "iota" (⍳), seems to make its own program loop within a line. When used one-sided, it furnishes a series of ascending numbers up to the number it's operating on. This until the last one is reached.

You type: 3 × ⍳ 7
APL replies: 3 6 9 12 15 18 21

In other words, one-sided iota looks to be doing its own little loop, increasing its starting number by 1, until it gets to the value on its right, and chugs on down the line with each.

Very sneaky way of doing a loop.

However! It isn't really looping, exactly. What the iota does is create a one-dimensional array, a row of integers from 1 up to the number on its right. This result is what then moves on leftward.

/continued next issue.

SHARP SPECIAL SYSTEMS

by Hugh O'Rourke

I.P. Sharp Associates has a ten-year history of experience in the design and implementation of minicomputer-based systems for process control, data processing and telecommunications applications. Activities in this area were formalized under the Special Systems heading over two years ago, and have grown to include the work of more than fifty computer professionals throughout the company.

In addition to our continuing role as consultants in all aspects of hardware and software systems implementation, several turnkey systems for a wide variety of customers were delivered. An Airport Information System, including the on-line data entry and display of flight arrival and departure times, is now operating at Schiphol Airport in Holland. Plant Operation and Information Systems for Post Office sorting plants in Ontario, have been installed for the Department of Public Works. Several digital read-out systems for use in metallurgical analysis via vacuum emmission spectrometers are now operating in plants and research facilities in both Canada and the United States, and a system of data concentrator computers is currently being installed on our own *SHARP APL* communications network.

Discussions of these and other projects will be included in ensuing issues of the Newsletter and further information about Special Systems projects and capabilities may be requested from your local Sharp office, or Head Office in Toronto.

Suite 1400, 145 King Street West, Toronto, Canada M5H 1J8

Update

☐ Please amend my mailing address as indicated.

☐ Add to your mailing list the following name(s).

☐ Send me SHARP APL manuals and product literature as listed.

☐ Note my comments: _____

_____

_____

_____

_____

_____

_____

The Newsletter is a regular publication of I.P. Sharp Associates Limited. Contributions and comments are welcomed and should be addressed to: Jeanne Gershater, Editor, I.P. Sharp Newsletter, Suite 1400, York Centre, 145 King Street West, Toronto, Ontario, M5H 1J8.

I.P. Sharp Associates Limited

Head Office: Suite 1400, 145 King St. West, Toronto, Canada M5H 1J8 (416) 364-5361

Canada – Regional Offices

Calgary
Suite 1000,
615 – 2nd Street S.E.,
Calgary, Alberta
T2G 4T8
(403) 265-7730

Edmonton
310, 9939 Jasper Ave.,
Edmonton, Alberta
T5J 2W8
(403) 426-5313

London
Suite 1400,
275 Dundas St.,
City Centre, Canada Trust Tower,
London, Ontario
(519) 434-2426

Montreal
Suite 1610,
555 Dorchester Blvd. West,
Montreal, Quebec
H2Z 1B1
(514) 866-4981

Ottawa
Suite 600,
265 Carling Ave.,
Ottawa, Ont
K1S 2E1
(613) 236-9942

Vancouver
Suite 604,
1112 West Pender St.,
Vancouver, B.C.
V6E 2S1
(604) 682-7158

England

I.P. Sharp Associates Limited
Gloucester
29 Northgate St
Gloucester
0452 28106

London
118 - 119 Piccadilly,
Mayfair, London W1V 9FJ
England
(01) 629-1564

Europe

Intersystems, B.V.
Herengracht 244,
Amsterdam 1002,
The Netherlands
(020) 250401

APL Europe S.A.
Ave. du Général de Gaulle 39,
105 Bruxelles,
Belgium
(649) 94 30

I.P. Sharp GMBH
Kaiser-Friedrich-Ring 98
4000 Duesseldorf
West Germany
(0211) 579084

U.S.A. – I.P. Sharp Associates, Inc.

Boston
Suite 812,
148 State St.,
Boston, Mass. 02109
(617) 523-2506

Chicago
Suite 424,
8501 West Higgins Rd.,
Chicago, Ill. 60631
(312) 693-5895

Dallas
Suite 1148,
Campbell Centre,
8350 Northcentral Expressway,
Dallas, Texas 75206
(214) 369-1131

Minneapolis
Suite 104,
5001 Cedar Lake Road,
St. Louis Park,
Minn. 55416
(612) 374-9406

New York City
Suite 250, East Mezz.,
Pan Am Bldg.,
New York, N.Y. 10017

Newport Beach
Suite 1135,
610 Newport Centre Drive,
Newport Beach, Ca. 92660
(714) 6445112

Rochester
Suite 1150,
183 Main Street East,
Rochester, N.Y. 14606
(716) 546-7270

San Francisco
Suite C409,
900 North Point Street,
San Francisco, Ca. 94109
(415) 673-4930

Seattle
Suite 3735,
SeaFirst Building,
1001 Fourth Ave.,
Seattle, Wa. 98154
(206) 938-0500

Washington D.C.
1815 Fort Myer Drive
Arlington, Va. 22209
(703) 527-3333

White Plains, N.Y.
Suite 39, Station Plaza,
250 East Hartsdale Ave.,
Hartsdale, N.Y. 10530
(914) 472-6380

SHARP APL Local Access In:

Canada
Calgary
Edmonton
Halifax
Kitchener
London
Montreal
Ottawa
Quebec City
Saskatoon
Sault Ste. Marie
Toronto
Vancouver
Winnipeg

U.S.A.
Atlanta
Boston
Buffalo
Chicago
Cleveland
Dallas
Des Moines
Los Angeles
New York City
Rochester
San Francisco
Santa Ana
Seattle
St. Paul, Minn.
Syracuse
Washington
White Plains

Europe
London
Gloucester
Amsterdam
Duesseldorf
Paris

APL Operator (416)363-2051