

Checkpoint

March 6, 1974

[Typeset version by Paul McJones and Academia.edu Co-scientist, 2026]

This is a report on the current implementations of CRMS APL syntax and the CRMS APL terminal interface. Both topics are presented at a level appropriate for APL\360 programmers. Neither implementation has been around long: last week 15 new terminals arrived. Today's CRMS APL syntax bears little resemblance to the syntax implemented two weeks ago. No doubt the rate of change will slow down as more CRMS APL programmers acquire habits they resist breaking.

A separate report on the current implementation of CRMS APL inter processing is forthcoming.

Background

Associated with every CRMS APL program are

- A text segment containing functions exactly as they were typed in. Multiple blanks are not automatically squeezed out as they are in APL\360.
- A data segment containing the internal representation of every variable in the program, assorted status flags, and the APL stack. The data segment corresponds to the APL\360 concept of workspace.

Before a program is executed, the APL translator converts the text segment into code more palatable to the APL machine. Then, the stack oriented APL machine is allowed to read the code, and read from and write on the data segment.

Many of the so-called 'APL primitive functions', such as rho, iota, and floor, are in the instruction set of the APL machine. Exact definitions of these instructions may be found in the 'CRMS APL Processor Reference Manual'.

Some of the more complex 'APL primitive functions', such as dyadic rotate and circle, are not in the instruction set. Instead, these 'primitives' are defined in terms of CRMS APL functions which only use 'primitives' in the instruction set. A listing of all CRMS APL functions that are also 'APL primitive functions' is kept on file in the lab, and available on a need-to-know basis. Understanding the listing does not require knowledge of any low-level language, but considerable familiarity with the APL language wouldn't hurt.

CRMS-APL primitives not in APL\360

Monadic ϵ , the existence function, returns 1 if its right argument is defined, and returns 0 if its right argument is the undefined value. The right argument must be a simple (unsubscripted) variable.

Example: $\rightarrow INIT \Delta VAR + \epsilon VAR$
Syntax errors: $\epsilon V[2]$
 $\epsilon VAR1, VAR2$
 $\epsilon VAR1 \leftarrow VAR2$
 $\epsilon FUNCTION CALL$
 $\epsilon 2\ 3$

Nilladic $\cup$, the undefined value function, returns the undefined value. Misuse of this function can cause the peculiar value error.

Example: $A \leftarrow B \leftarrow C \leftarrow \cup$
Value error: $A + A \leftarrow \cup$
Peculiar value errors: $V[2] \leftarrow \cup$
 $2 + \cup$

Monadic $\downarrow$, the push function, returns no explicit result, and puts its right argument on top of the stack.

Nilladic $\uparrow$, the pop function, returns the top of the stack. Misuse of this function can cause the stack underflow error.

Example: $\downarrow 2 \times \uparrow$ $\cap$ ADD THE TOP OF THE STACK TO ITSELF

Dyadic *INTCH* interchanges the top two stack entries, and returns the new top of the stack.

Example: $3\ 4 \leftrightarrow (4\ INTCH\ 3), \uparrow$
 $\downarrow (2\ INTCH\ \uparrow) * \uparrow$ $\cap$ SQUARE THE TOP OF THE STACK

Monadic τ , the translate function, is a 'scalar function' like $\times$ and $[]$. τ converts characters to their internal numeric representations, and converts integers between 0 and 255 inclusively to their character representations.

Example: $'3579' \leftrightarrow \tau(\tau'0') + 10\ 10\ 10\ 10\ \tau\ 3579$
 $9876 \leftrightarrow 10 \downarrow (\tau'9876') - \tau'0'$ $\cap$ ASSUME '0'-'9' IN PROPER ORDER

Monadic $\perp$, the indicator function, is another 'scalar function'. $\perp$ converts characters to 0's and numbers to 1's.

Example: $1\ 0\ 1\ 0\ 0\ 1 \leftrightarrow \perp\ 23 \times '4' = 9'2$

Dyadic *//*, the delimiter pseudo-function, eats its right argument.

Example: $A \leftarrow 1 \ // \ B \leftarrow 2 \ // \ \rightarrow 3 \times \iota CONDITION \ // \ D \leftarrow 4$
 $X \leftarrow (5 \ // \ \rightarrow DIV0ERR \times \iota A = 0) \div A$

$\backslash$, the ecuder operator, is not a function, just as the inner product and reduce operators are not functions. Ecuder is best explained by example.

Example: $10 \ 9 \ 7 \ 4 \leftrightarrow + \backslash \iota 4$
 $1 \ 'A' \leftrightarrow = \backslash 'AA'$
 $2 \ 2\rho 4 \ 6 \ 3 \ 4 \leftrightarrow + \backslash [1]2 \ 2\rho \iota 4$

Monadic $\$$, the ID operator, returns an ID-number unique for each user defined function. As with all APL operators, ID is a function of a function. As with the reduce ($/$) and ecuder ($\backslash$) operators, $\$$ accepts optional arguments (in brackets). A nilladic, monadic, or dyadic function is referenced when the optional argument is 0, 1, or 2 respectively. ID-numbers are used by some of the inter process routines.

Example: $\$STARTUP$
 $\$[1]MONADIC_AND_DYADIC_FUNCTION_NAME$

Shorthand primitive functions

RETURN is shorthand for $\rightarrow 0$
 $A \ IF \ B$ is shorthand for $B\rho A$
 $A \ UL \ B$ is shorthand for $(\sim B)\rho A$
 $ANDREDUCE \ B$ is shorthand for $\rho, B\rho 1$

Example: $\rightarrow 3 \ IF \ 17 < C \leftarrow C - 1$
 $\rightarrow 9 \ UL \ 15 > C \quad \text{ } n \ THINK \ OF \ 'UL' \ AS \ 'UNLESS'$

User defined functions

The header

$\nabla R \leftarrow \rho L \ FN[\circ A; \circ B; C; D; \rho E] \ M$

tells us that R is the result variable of the explicit dyadic function FN . L is the left argument; A, B, C, D , and E are extra arguments (E is the first extra argument); and M is the right argument. Arguments A, B, C, D and M are passed by-value, as in APL\360. Arguments L and E are passed by-reference as in FORTRAN IV (ostensibly). When dyadic function FN is called, the left argument and the first extra argument must be simple variables. The first extra argument may not be missing:

$X8 \leftarrow X7 \ FN[X6; X5; X4; X3;] \ X1 \quad \text{ } n \ THIS \ IS \ A \ SYNTAX \ ERROR$

because missing arguments may not be passed by-reference.

If the left argument were missing in the call, then monadic function FN would be called. In CRMS APL, it is not an error to define identically named monadic and dyadic functions. (Many of the primitive function names, such as minus (-), refer to both monadic and dyadic functions.) Also, in CRMS APL, it is not an error to define identically named nilladic and dyadic functions. But, identically named nilladic and monadic functions are unallowable for syntactical reasons.

When dyadic function FN is called, the second through fifth extra arguments may be missing.

```
X8←X7 FN[ ;X5; ;X3;X2] X1
```

Here, the third and fifth extra arguments are missing. In this case, the undefined value is passed to arguments A and C . An equivalent function call is

```
X8←X7 FN[u;X5;u;X3;X2] X1
```

The existence function, monadic ϵ , can ascertain whether A and C are undefined.

The ' $\circ$ ' preceeding A and B in the function header tells us that A and B are optional arguments. Optional arguments must be pass by-value extra arguments. If a function has any optional arguments, all its optional arguments must be consecutive, and the last extra argument must be an optional argument.

Optional arguments may be omitted (as opposed to missing) in the function call. When dyadic function FN is called, the fourth and fifth extra arguments may be omitted:

```
X8←X7 FN[ X4;X3;X2] X1
```

Here, the undefined value is passed to arguments A and B . In fact, the above call is equivalent to

```
X8←X7 FN[ ; ;X4;X3;X2] X1
```

Brackets in the function call are unnecessary if all extra arguments are optional. Thus,

```
▽ RESULT ← ROTATE[◦DIMENSION] VARIABLE
```

might be the header of a function simulating the monadic ϕ primitive function.

Error messages

CRMS APL syntax errors are detected by the APL translator, not the APL machine. These errors are documented elsewhere. The APL machine is capable of detecting system errors and all non-syntactical user errors. The non-syntactical user errors are:

- Data segment overflow: analogous to *WS FULL*.
- Value error: attempt to load an undefined variable onto the stack.
- Peculiar value error: an operand already on the stack is undefined.
- Rank error
- Length error
- Wrong indicator error: a character encountered where a number should be.
- Numeric overflow: floating point exponent overflow.
- Numeric underflow*: floating point exponent underflow.
- Division by zero: attempt to divide any number, including 0, by 0.
- Domain error
- Wrong subscript count: incorrect number of subscripts.
- Index error: subscripts out of bounds.

Debugging aids

- Statement trace/stop: breakpoint feature analogous to ~~TA~~ or ~~SA~~.
- Function call trace/stop*
- Function return trace/stop*
- Opcode trace/stop*

* Interrupt can be masked out at microcode level.

About the new terminals...

The CRMS terminal interface is designed to operate DTC's model 300 Diablo terminals. The 300 is newer and lighter than the IBM 2741, the traditional APL\360 terminal. The 300 also costs more.

This report is typed with the Diablo APL-ASCII daisy wheel. Besides containing all the graphics on IBM's APL Selectric typeball, the daisy wheel contains diamond, dollar sign, left and right braces, and left and right tacks ($\diamond$ \$ { } $\vdash$ $\neg$). And in addition to the positioning operations of the 2741—space, backspace, carriage return with linefeed, and tab—the 300 can reverse linefeed, tabset, tabclear, and plot periods so close together they touch.

The 300 is over twice as fast as the 2741. The half duplex 2741 is incapable of transmitting and receiving simultaneously. The CRMS terminal interface exploits the full duplex nature of the 300 by supporting type-ahead and an extremely powerful editor.

Proposed keyboard for DTC 300

Editorial: The original handwritten keyboard sheet follows. Its six suggestions are transcribed below; key legends remain in facsimile.

1. Move attn key to the right of the right shift.
2. Move lock key to the right part of the control key.
3. Move control key to the attn & lock keys.
4. Move return key(s) to the repeat & (copy) keys.
5. Move repeat key to the return₁ key.
6. Move the break key to the left part of the return₂ key.

3/6/74

PROPOSED KEYBOARD FOR DTC 300

\$	..	-	<	≤	=	≥	>	≠	√	^	-	÷	+	-	here
◇	1	2	3	4	5	6	7	8	9	0	+	x	{	}	is
tab	?	w	e	p	~	↑	↓	l	o	*	→	lf	return ₁	return ₂	
	Q	W	E	R	T	Y	U	I	O	P	←	bs			
attn	lock	α	Γ	L	-	▽	Δ	o	i	□	(	)	repeat	(copy)	break
		A	S	D	F	G	H	J	K	L	[	]			
control	shift	C	D	N	U	I	T	I	;	:	\	shift			
		Z	X	C	V	B	N	M	,	.	/				

SUGGESTIONS

1. move attn key to the right of the right shift.
2. move lock key to the right part of the control key.
3. move control key to the attn & lock keys.
4. move return key(s) to the repeat & (copy) keys.
5. move repeat key to the return₁ key.
6. move the break key to the left part of the return₂ key.