APL 82, W. H. Janko, W. Stucky (ed.),
APL Quote Quad, Vol. 13 No. 1,
© 1982 by ACM, Inc.

THE HUNTING OF THE SNARK

Philip R Chastney
The Cedars, Chapel Lane, Ebley
Stroud, Gloucestershire, England GL5 4TD

1. INTRODUCTION

Consideration of the formal semantics of APL particularised to consideration of the unusual properties of the null expression. It was something of a surprise to discover that these properties had already been fully documented 100 years ago by Lewis Carroll (Cal). Since that opus is not normally a part of the lore of APL, this paper attempts to re-interpret the original findings in 20th century terminology.

APL has a number of objects which are, in some sense, null or empty - two empty vectors, multitudinous other empty arrays, undefined objects and local variables with names but no values. There is at least one other null entity. It has no name, and it has no value. Worse yet, it is denoted by the empty string, and is therefore not easily seen.

We have sailed many months, we have sailed many weeks,
(Four weeks to the month you may mark),
But never as yet ('tis your Captain who speaks),
Have we caught the least glimpse of a Snark.

2. THE SEARCH

They sought it with thimbles, they sought it with care,
They pursued it with forks and hope,
They threatened its life with a railway share,
They charmed it with smiles and soap.

APL's null object has the same elusive nature as Lewis Carroll's Snark. Unfortunately, methods successfully employed in the nineteenth century are not applicable here, and we must pursue our search through a study of the language. "The study of a natural or artificial language is traditionally split into three areas:

(i) Syntax: this deals with the form, shape, structure etc., of the various expressions in the language.

(ii) Semantics: this deals with the meaning of the language.

(iii) Pragmatics: this deals with the uses of the language." (Gol)

2.1 Syntax

We may index a vector with a scalor, as in *V*[1]. We may index a vector with a vector, as in *V*[2 1 2 3]. We may index a vector with any array, as in *V*[2 3 5⍴⍳6]. We may also index a vector with "nothing", as in *V*[]. The value of that last expression is not the same as *V* itself, because we get somewhat different results from *V*[]←0 and *V*←0. What, if anything, is enclosed between those square brackets?

2.2 Semantics

Among the primitive domains of APL, we may distinguish the data domains and the syntactic domains, each domain having (to use Dana Scott's terminology (Scl)) a "least" or "bottom" element. The data domains include arrays of numbers and arrays of characters, and their respective "least" elements can be evaluated from (⍳0) and (''). The syntactic domains include expressions, statements and user-defined functions. The "least" function has a name, no arguments, no result, and no lines apart from the header. The "least" statement is represented by the empty string. What is the "least" expression?

2.3 Pragmatics

Using APL conversationally, the end of input is marked by hitting carriage return. In immediate execution mode, the statement preceding the carriage return is evaluated, and any appropriate display is made. If we type in '' we get an empty display - that is, an empty string, followed by a carriage return. If we simply hit carriage return, without any preceding input, there is no display: in fact, there is no response whatever, apart from the usual six spaces prompting for the next line. What exactly was evaluated, that produced so little response?

The answer, in all three cases, is a 'snark'.

3. BRACKETS AND SEMICOLONS

Snarks commonly occur near brackets and semicolons in indexing expressions, so closer study of snarks requires a closer study of brackets and semicolons.

Brackets perform two distinct duties simultaneously: they control the order of evaluation, somewhat like parentheses, and they denote the indexing operation, somewhat like the primitive dyadic mixed functions - and we need to study these two duties separately.

3.1 Parentheses

Figure 1 is an adaption of R H Lathwell's function EVAL (La1), which we can use to study the effects of parentheses. The function's flow-of-control is based on a state-symbol table, where the states 3, 2 and 1 correspond to the search for a right-argument, the search for an infix function, and the search for a left-argument, the state being determined by the type of symbol at the top of the stack. In particular, assuming right-to-left parsing, we can see that a right parenthesis causes the popping of the stack from the previous right marker.

(Parentheses are a subject eminently worthy of study, and interested readers are referred to Carl Lindemann Jr (Li1). Attempts to dispense with parenthese either produce a less attractive syntax (Cu1,Qu1), or require a switch to prefix or postfix notation.)

3.2 Brackets

Figure 2 is an extension of that idea, which demonstrates the dual role of brackets. A right bracket is placed on the stack as marker, and the occurrence of the left bracket causes (i) the collapsing of the stack from the corresponding right marker, into a single list, and (ii) the placing on the stack of the I-beam, to denote a function in search of a left argument.

3.3 Semicolons

As well as demonstrating the peculiar position of brackets within the hierarchy of primitive functions, EVAL2 also enables us to study the action pushed onto the stack, without a new marker, and recognition of a new expression is started. This places the semicolon in the same syntactic class as parentheses, with the same place in the operator hierarchy as the right parenthesis.

In current definitions of APL, the purpose of the semicolon is to separate the expressions which define the elements to be selected from each separate dimension of the left argument. Or, to put it another way, the semicolon helps to build the list defined by the sequence or expressions, which will be applied by the left bracket, to its left argument.

3.4 Summary

The expressions *A*[*I*], *A*[*I*;*J*] and *A*[*I*;*J*;*K*] clearly have sequences of one, two and three expressions between their respective pairs of brackets, and equally clearly their respective left arguments must have one, two and three dimensions. Each dimension requires a separate expression (or a snark), and these items are separated by semicolons. In the third case, we have an indexing list of three items; in the second case we have an indexing list of two items, and - by extrapolation - in the first case we have a list containing one item.

Before applying the list to its left argument, part of the action of the brackets is to enclose and build that list from the results of the separate expressions, so we have identified a primeval enclosure operator, and a list-element separator, and we can now go on to extend their applicability.

4. POSSIBLE EXTENSION

4.1 Axis and Frame

To ensure that no incompatibilities are introduced, we need to consider the alternative use of brackets to denote the "axis" operator. If the right bracket occurs immediately to the left of a value, it is the delimiting right scope marker for an axis operator. Using brackets to denote enclosure, we can re-express the "on frame" construct proposed by Bernecky and Iverson (Be1) in more familiar syntax, as a straightforward extension of the axis operator.

```
     +\ö(<IR) B   ↔     +\[IR] B
F ö((<IL),<IR) B   ↔   F[IL;IR] B
```

4.2 Lists

4.2.1 The first major extension would be to define a monadic version of left bracket. The syntax is no problem and, for semantics, it could be used to denote the "strict enclosure" function which is already a part of its dyadic definition, and hence to build linear lists in a manner consistent with that definition: for example, `[A;B;C]` evaluates to a three-element list; `[A;B]` evaluates to a two-element list; `[A]` evaluates to a single-element list containing *A*; and `[]` evaluates to a single-element list containing a snark. This removes the anomoly that "the subscript expressions yield APL data structures, but the collection enclosed in delimiters does not" (Lel).

4.2.2 Since one array equals another only if all corresponding pairs of elements are identical, we may test for equality with the expressions `[A]=[B]`

4.2.3 Using semicolons as separators between list-elements, we can construct lists with parentheses as effectively as with brackets, as in, for example, `(A;B;C)`. (N.B. The relative positions of assignment and semicolon within the operator hierarchy, make it important to rember the parentheses - the expression `X←(A;B;C)` is not identical to `X←A;B;C` .)

This would enable us to call an n-place function with non-conformable arguments, as in, for example, `F(ARG1;ARG2;ARG3)` and this is indeed the approach taken with `⎕FMT`.

Any of the items could be omitted, and a snark substituted instead, and a neat application of this would be to check for the elided argument, and evaluate it so that the arguments bear a specified relation to each other. For example, a function SPIN could be constructed for the compound interest formula `S=P×(1+I)*N` and used as follows:

what is the value of 100 @ 5% per period, after 20 periods?

```
    X←SPIN( ;100;0.05;20)
```

what is the rate of interest if a principal of 100 grows to 1000 after 15 periods?

```
    X←SPIN(1000;100;;15)
```

It is a matter of taste and convenience whether the function returns all the related values, or just the unknown quantity.

4.3 Manipulating Lists

4.3.1 A striking feature of lists is their similarity to character arrays:

vector	`'ABC'`	linear list	`(A;B;C)`
scalar	`'A'`	"simple" array	`(A)`
empty vector	`''`	empty list	`()`

The similarity persists when we come to consider extending the domains of the primitive functions to include this new class of APL objects. Arithmetic is not possible, of course, but all the other primitives can be made to apply to lists in a straightforward manner. Snarks act as "fill values" for "overtake" operations, of course, and that the only items worthy of further comment are "display" and "indexing".

4.3.2 The display of lists was defined a long time ago, only then it was called "mixed output", as in, for example:

```
    'TABLE'; 2 3ρι6
```

and those far-sighted pioneers even defined the display for a list containing snarks, as in the following example, which will print a snark between the two arrowheads.

```
    '***→';;'←***'
```

4.3.3 Indexing operations always were defined for list arguments on the right, provided that the list was defined between the indexing brackets. Extending the domain of dyadic left bracket to include externally defined lists would allow the development of rank-independent code.

```
    IJK←(ρρA)↑(I;J;K)
    X←A[IJK]
```

Of course, this is Ghandour and Mezei's "slice" function, with the added ability to use snarks to represent elided indices.

4.3.4 Hitherto, the principal way of collecting non-conformable data items has been to use a component file. When an object is put to file, it appears in the file index as a single component; when it is read from file, the object created in core is not a single scalar component, but the original APL array itself, with its original shape and value. In the terminology of generalised arrays, the component is automatically "disclosed". It would seem sensible to apply a similar convention when indexing lists, and do an automatic disclosure whenever the index is scalar.

At this point, something magical happens. The data-handling file functions are disc-to-core (or core-to-disc) transactions, which are a proper subset of the core-to-core transactions that we can define for linear lists.

`'LIST' FTIE 1`	(not necessary)
`A←FREAD 1, I`	`A←LIST[I]`
`FSIZE 1`	`ρLIST`
`A FAPPEND 1`	`LIST←LIST,[A]`
`FDROP 1,K`	`LIST←K↓LIST`
`A FREPLACE 1,I`	`LIST[I]←[A]`

It would be nice to be able to replace the old file functions with the new revised syntax in a consistent manner. The attraction is particularly strong with paged virtual systems, but with other systems, designating which information is stored in core and which is to be stored principally on disc, requires some mechanism for identifying a name as pertaining to a file, and that, in turn, requires concepts which are external to this discussion.

4.3.5 Finally, it is worth noting that the "reshape" function, and indexing a list with an argument of high rank, allows us to use linear lists to build nested arrays of any shape, size and depth.

4.4 Monadic Functions

Consider the expression *A*÷*B*. What happens if A is a snark? Using Λ to denote a snark, we get the syntactically identical expression of Λ÷*B*. But a snark is really denoted by the empty string, and so, making the appropriate substitution, we derive the expression ÷*B* which clearly has an empty string to the left of the function symbol.

When a snark occurs in indexing expressions such as *A*[*I*;;*K*] it stands for the default selection of all rows. Similarly, when a snark is used as the left argument in ÷*B* it denotes a default left argument of 1. Those primitive functions commonly called "monadic" are now seen to be dyadic functions with a snark for left argument.

Extending the notion to user-defined functions does no great violence to the present syntax, but poses the problem of how to recognise a snark inside the function. By exactly the process already described for division, we can make the following transformations:

```
A=B →→  Λ=B →→  =B
```

The final expression is true only if the value on the right is equal to the snark on the left, and could be used to test and branch according to the nature of the left argument, with something like:

```
→(=A)↓DYADIC,MONADIC.
```

The trouble with this interpretation is that it is so spectacularly counter intuitive, that it is probably best left unimplemented.

A better alternative (if we can agree a suitable modification to the treatment of VALUE ERRORs) is to test the enclosed right argument against an enclosed snark, thus:

```
→([]=[A])↓DYADIC,MONADIC.
```

4.5 Escape and Expunge

It is possible that snarks occur as right arguments, but since the resultant grammar is ambiguous, we can never be sure - except when the function name occurs immediately to the left of a delimiter. Possible delimiters are the right parenthesis, the semicolon, the right bracket, and the end-of-line marker.

4.5.1 A snark can definitely be detected to the right of a branch arrow, when it is followed by an end-of-line delimiter - the entire statement being known as "escape". Extending the idea, the expression →(0;)[*L*] will "return" if *L*=⎕*IO*, and "escape" if *L*=⎕*IO*+1.

4.5.2 Branching and assignment have much in common, besides their visual similarity. At the machine level, the similarity is obvious - the one causes a new value to be loaded into the program counter, while the other causes a new value to be loaded into a register or memory location - let us reinforce that similarity by extending to assignment the privilege of having a snark for its right argument. After executing *A*←*B* the left argument *A* acquires the values, shape and type of *B*, and in most implementations, it even uses the same memory locations as *B*. After executing *A*←Λ the left argument *A* acquires the values, shape, and type of Λ. But Λ denotes a snark, and a snark has no values, no shape and no type. In short, to expunge *A*, we type *A*←.

But oh, beamish nephew, beware of the day,
 If your snark be a boojum; for then
You will softly and suddenly vanish away
 And never be met with again.

5. <u>CONCLUSIONS</u>

Snarks undoubtedly exist, even if we are unable to see them. A snark may conveniently be thought of as being the null expression; it maybe more correctly identified as the result obtained on evaluating the null expression - both the null expression and its result are denoted by the empty string, and at no point is the difference critical. Recognition of the occurrence of snarks in other contexts can be used to revise the formal syntax (Fa2) or APL, as shown in Figure 3, for greater simplicity, uniformity and generality - all desirable properties (Fa1).

With these modest changes to syntax, and with the semantics outlined above, we have a "true extension" of the language (Br1).

No new symbols are introduced, none of the present definitions is changed; no existing (error-free) programs will be affected - again, all desirable properties.

In return,

(i) we get an "enclose" function, and avoid the need for "idem" or "disclose", giving us the ability to build linear lists;

(ii) by extending the domains of existing premitives, we can define arrays or any complexity;

(iii) escape, expunge, and mixed output are made available in a syntactically consistent manner;

(iv) "frame" can be made consistent with the axis operator;

(v) further small changes in the future could be made to allow file functions, ambivalent user-defined functions and a test for existence to be a part of the standard definition.

Finally, it is not possible to provide this additional power by writing cover functions in APL - the change has to be effected through a change in syntax.

<u>REFERENCES</u>

(Bel) Bob Bernecky and Kenneth E Iverson. "Operators and enclosed arrays". Proceedings of a Users' Conference, I P Sharp Associates, Toronto 1980.

(Brl) J A Brown. "Evaluating extensions to APL". APL79 Conference proceedings, p.149. ACM, 1979.

(Cal) Lewis Carroll. "The hunting of the Snark". Originally published by Macmillan, London, 1876.

(Cul) H B Curry. "On the use of dots as brackets in logical expressions". J Symbolic Logic, p.26. July 1937.

(Fal) A D Falkoff and K E Iverson. "The design on APL". IBM J.Res.Develop, Vol.17, No. 4, p.324. July 1973.

(Fa2) A D Falkoff and D L Orth. "Development of an APL standard". APL Quote-Quad, Vol 9, No.4, Part 2. ACM June 1979.

(Ghl) Ziad Ghandour and Jorge Mezei. "General arrays, operators and functions". IBM J.Res.Develop, Vol.17, No.4, p.337. July 1973.

(Gol) Michael J C Gordon. "The Denotational Description of Programming Languages". Springer-Verlag, 1979.

(Lal) Richard H Lathwell. "Some implications of APL order-of-execution rules". APL79 Conference Proceedings, p.330, ACM, 1979.

(Lel) G R Lewis. "A new array indexing system for APL". APL75 Conference Proceedings, p.235. ACM, 1975.

(Lil) Carl Lindemann Jnr. "Mathematics Made Difficult".

(Pel) Roland H Pesch. "Indexing and indexed replacement in APL". APL81 Conference Proceedings, p.258. ACM, 1981.

(Qul) W V O Quine. "Mathematical Logic", p.37. Harvard University Press, 1951.

(Scl) Dana Scott. "Outline of a mathematical theory of computation". Oxford University Computing Laboratory, Programming Research Group, 1970.

Figure 1

```
    ∇ R←EVAL1 INPUT;⎕IO;CR;FNS;I;QAV;S;
            STACK;SYMBOL;TAB;TMP
[1]   ⍝
[2]   ⍝ BASIC SYNTAX ANALYZER
[3]   ⍝
[4]    ⎕IO←1
[5]    FNS←'<≤=≥>≠∨∧+-×÷?∊⍴~↑↓⍳○*←→⌈⌊⊥⊤|'
[6]    TMP←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[7]    QAV←FNS,'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
            TMP,')('
[8]   ⍝
[9]    TAB← 3 5 ⍴0
[10]  ⍝ STATE = 1 →→ LOOKING FOR LA
[11]  ⍝ STATE = 2 →→ LOOKING FOR FN
[12]  ⍝ STATE = 3 →→ LOOKING FOR RA
[13]  ⍝ COLS:   FN VAL  )   (  EOL
[14]   TAB[1;]←MON,DYA,STK,MON,MON
[15]   TAB[2;]←STK,ERR,ERR,POP,EXT
[16]   TAB[3;]←ERR,STK,STK,ERR,EXT
[17]  ⍝
[18]   CR←CARRIAGERETURN
[19]   I←1+⍴S←CR,R←INPUT
[20]   STACK←''
[21]  NXT:SYMBOL←S[I←I-1]
[22]  RUL:R←R,(SYMBOL≠CR)/((⍴S)↑(I-1)↑S),
            ' : ',SYMBOL,STACK
[23]   →TAB[3⌊TYPE 1↑STACK;5⌊TYPE SYMBOL]
[24]  ⍝
[25]  MON:STACK←TEMP,2↓STACK
[26]   →RUL IF SYMBOL≠CR
[27]   R←R,((⍴S)↑CR),' : ',STACK
[28]   →EXT
[29]  ⍝
[30]  DYA:SYMBOL←TEMP
[31]   STACK←2↓STACK
[32]   →RUL
[33]  ⍝
[34]  POP:→ERR IF ')'≠1↑1↓STACK
[35]   SYMBOL←1↑STACK
[36]   STACK←2↓STACK
[37]   →RUL
[38]  ⍝
[39]  STK:STACK←SYMBOL,STACK
[40]   →NXT
[41]  ⍝
[42]  EXT:→0 IF(1=⍴STACK)∧~')'∊STACK
[43]  ERR:R←R,CR,'SYNTAX ERROR',S,CR,
            (¯1⌊1-I)↑'∧'
    ∇
```

```
     ∇ R←TEMP
[1]    R←1↑TMP
[2]    TMP←1↓TMP
     ∇

     ∇ R←TYPE SYMBOL
[1]    R←1++/(QAV⍳SYMBOL)>QAV⍳'|Z)(⌈;'
     ∇
```

Figure 2

```
     ∇ R←EVAL2 INPUT;⎕IO;CR;DEPTH;FNS;I;QAV;
             S;STACK;SYMBOL;TAB;TMP
[1]   ⍝
[2]   ⍝ BASIC SYNTAX ANALYZER,
[3]   ⍝ INC. INDEXING AND LISTS
[4]   ⍝
[5]    ⎕IO←1
[6]    FNS←'⌶<≤=≥>≠∨∧+-×÷?∊⍴~↑↓⍳○*←→⌈⌊⊥⊤|'
[7]    TMP←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[8]    QAV←FNS,'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
             TMP,'])(⌈;'
[9]   ⍝
[10]   TAB← 3 7 ⍴0
[11] ⍝ COLS:  FN   VAL ])    (    [    ;   EOL
[12]   TAB[1;]←MON,DYA,STK,MON,MON,DUN,DUN
[13]   TAB[2;]←STK,ERR,ERR,POP,PK2,STK,EXT
[14]   TAB[3;]←ERR,STK,STK,PK1,PK2,STK,EXT
[15] ⍝
[16]   CR←CARRIAGERETURN
[17]   I←1+⍴S←CR,R←INPUT
[18]   STACK←''
[19] NXT:SYMBOL←S[I←I-1]
[22] RUL:R←R,(SYMBOL≠CR)/((⍴S)↑(I-1)↑S),
             ' : ',SYMBOL,STACK
[21]   →TAB[3⌊TYPE 1↑STACK;TYPE SYMBOL]
[22] ⍝
[23] DUN:I←I+1
[24]   SYMBOL←TEMP
[25]   STACK←2↓STACK
[26]   →RUL
[27] ⍝
[28] MON:STACK←TEMP,2↓STACK
[29]   →RUL
[30] ⍝
[31] DYA:SYMBOL←TEMP
[32]   STACK←2↓STACK
[33]   →RUL
[34] ⍝
[35] POP:→PK1 IF ')'≠1↑1↓STACK
[36]   SYMBOL←1↑STACK
[37]   STACK←2↓STACK
[38]   →RUL
[39] PK1:DEPTH←⌊/STACK⍳'])'
[40]   →ERR IF ')'≠¯1↑DEPTH↑STACK
[41]   SYMBOL←TEMP
[42]   STACK←DEPTH↓STACK
[43]   →RUL
[44] ⍝
[45] PK2:DEPTH←⌊/STACK⍳'])'
[46]   →ERR IF ']'≠¯1↑DEPTH↑STACK
[47]   SYMBOL←'⌶'
[48]   STACK←TEMP,DEPTH↓STACK
[49]   →RUL
[50] ⍝
[51] STK:STACK←SYMBOL,STACK
[52]   →NXT
[53] ⍝
[54] EXT:→0 IF~∨/'])'∊STACK
[55] ERR:R←R,CR,'SYNTAX ERROR',S,CR,
     ∇       (¯1⌊1-I)↑'∧'
```

Figure 3

```
statement          ::=  label-name : state-
                        ment-body
                    |   statement-body

statement-body     ::=  comment
                    |   sequence
                    |   → expression

sequence           ::=  expression
                    |   sequence; expression

expression         ::=  null
                    |   subexpression

subexpression      ::=  expression fn-indent-
                        ifier subexpression
                    |   object ← expression
                    |   simple-expression
                    |   simple-expression
                        [ sequence ]
                    |   ⎕ [ sequence ]
                    |   ⍞   sequence ]
                    |   [ sequence ]
                    |   object

simple-expression  ::=  constant-identifier
                    |   niladic-identifier
                    |   ( sequence )

object             ::=  variable-identifier
                    |   variable-identifier
                        [sequence]
                    |   ⎕
                    |   ⍞
```