My Favorite Idiom

October, 1988

Dr. James A. Brown

IBM Santa Teresa Lab J88/E42
555 Bailey Ave.
San Jose, Calif. 95141 USA

Introduction to Idioms

This paper discusses *APL* idioms: what they are, which ones are commonly recognized by people, which ones can be recognized by the *APL2* interpreter, and which ones are recognized by the *APL2* interpreter. Included is a discussion of why idioms are so valuable.

What is an idiom?

When designing and describing a programming language, it is common to borrow terminology from natural language. Iverson does this to a greater extent than most in his dictionary of *APL* (which describes extensions to Sharp *APL*) (Iv1). The word "idiom" is one of those borrowed from natural language. Unlike Iverson's adopted words, this one may not be entirely appropriate.

Here's the definition of "idiom" in natural language:

```
"An expression ... having a meaning which
 cannot be derived as a whole from the
 cojoined meanings of its elements."
         Webster's new collegiate dictionary
```

An *APL* idiom is essentially exactly the opposite. Here's a proposed definition of "idiom" in *APL*:

```
"An expression whose meaning is precisely
 determined from the meanings of its elements
 yet, as a whole, represents a single concept."
          Yours truly
```

This difference in meaning exposes itself in the usefulness of idioms. For example, the translation of an idiom from one natural language to another is the most difficult task of a translator. In fact, it is this kind of trouble that caused the automatic translation projects of the Artificial Intelligence people in the 1950's to fail.

On the other hand, *APL* idioms have a lot to do with what makes *APL* productive to read and write. The reasons for this are discussed later.

Example Idioms

Here are a few idioms and the ideas each represents:

Summation: `+/V`

This is, in some sense, a limiting case of an idiom. It certainly is a sequence of symbols that represents one idea. Such an expression would not normally appear in an idiom list. If you want to sum a vector in *APL*, you wouldn't do it any other way unless you wanted to control rounding errors (in which case you might, for example, sort the vector first).

Eliminate duplicates: `((V⍳V)=⍳⍴V)/V`

This is the most common example given of an *APL* idiom. It is a classic case of where the combination of symbols expresses precisely the desired operation but that fact is not immediately obvious from the individual operations. While this is the common way to eliminate duplicates, it can be more efficient to use other expressions. For example, if the vector is sorted (or can be sorted), `(1,2≠/V)/V` is probably faster.

Delete leading blanks: `(+/∧\TXT=' ')↓TXT`

This is a common idiom sometimes called the teepee idiom because the slashes and the "and" symbol resemble the tents constructed by the American Indians. Another idiom which has a name is "pick each enclose" (⊃⍤⊂) which is called the chipmunk idiom named after a small furry animal that collects nuts.

Why should people recognize idioms?

Idioms have a lot to do with what makes *APL* so powerful. The symbolic nature of *APL* operations and the array orientation of data and functions means that significant computations can be expressed in a short space. That means you can see a whole expression at a glance and that is extremely valuable when one expression equals one idea. In some sense, you can see the forest without any trees (to paraphrase and American idiom). Thus, if you know idioms, *APL* becomes more readable because you can read the idea instead of the individual primitives, and *APL* becomes more productive for the programmer because she can convert an idea into an expression faster.

Why should machines recognize idioms?

Most *APL* language processors are interpretive. They alternate between scanning *APL* expressions looking for single primitives and their arguments and the execution of these primitives. If an *APL* interpreter can recognize an idiom, it can evaluate the meaning of the phrase rather than its individual primitives. This improves system performance three ways:

- Elimination of syntax analysis

  Once the *APL* interpreter has recognized an idiom, it does not need to again scan each of the symbols that comprise the idiom. Instead, a scheme can be used to skip to the left end of the idiom completely eliminating the time used for syntax analysis.

- Elimination of intermediate results

  If an idiom involves more than one primitive function, its normal interpretive execution would involve allocating space for the result of the rightmost function. After execution of the rightmost function, its result becomes an argument of the next function in the idiom. When the second function completes, the space for the result of the first function is freed (in the implementation, this is called marking the space garbage). This process continues for other primitives in the idiom expression. If an idiom can be executed as a whole, the time for space allocations, de-allocations, and garbage collection (the process of reclaiming space marked garbage) can be reduced. This is more beneficial than it appears because if the space is never allocated, machine time is never spent storing or fetching data from the area. Finally, sometimes intermediate results are bigger than the final result. In these cases, an idiom may execute when its individual parts would get a *WS FULL*.

- Use of special implementation routines

  Special execution routines can implement the idea instead of the pieces. Thus, the whole idiom executes in the same time as a single primitive. On the negative side, addition of an idiom to an *APL* system is almost as difficult as adding a new primitive. "Almost" because no new symbol is involved and the idiom routine can choose to not implement exceptional cases. Also, the idiom routines themselves take space making the *APL* system larger.

When are idioms recognized?

Idioms are recognized whenever a defined operation is introduced into the workspace by any of the following mechanisms:

- Normal completion of function editing
- `⎕FX`
- `⎕TF`
- `)COPY )PCOPY`
- `)IN`

An implication of this style of idiom recognition is that introducing a function into the workspace is more expensive. An application that uses function files to dynamically introduce functions into the active workspace will run slower. If the function, once introduced, is executed many times, the additional time in idiom recognition is minimal. If a function is executed once and then erased, the recognition time may be wasted. This makes the use of packages and *APL2* external functions preferred over function files.

Another consequence of recognition at function fix time is that new idioms added after a workspace is `)SAVE`d will not be recognized until the functions are fixed again.

Finally, expressions in immediate execution, quad input, and any of the forms of execute (`⍎ , ⎕EC , ⎕EA`) do not benefit from idioms. This is important to know if you write performance measuring programs. If you execute a character string, idioms are ignored. If you fix a function containing the expression you want to time, idioms are used and the execution time is a more accurate measure of the expression in an application. Because *APL* does not have an adequate conditional construction, many people use execute to simulate a conditional as follows:

```
⍎(condition)/'expression'
```

While perhaps convenient, this construction is very expensive in execution time in general and, in particular, causes idioms to be ignored.

Problems in idiom recognition

Because idioms are not recognized at execution time, name classes are not known. This makes idiom recognition difficult (and expensive source program analysis is not practical or reliable at execution time). For example, given an expression containing the phrase `A-1`, it cannot be determined if `-` is monadic or dyadic. Given an expression like `(V⍳V)=⍳⍴V`, it is reasonable to assume that `V` is array valued. However, `V` could be a shared variable or a niladic function and the expression would be well formed yet not mean what the idiom normally means. It is possible that `V` is a monadic defined function in which case the expression contains an error. These considerations make idiom recognition difficult but not impossible.

Another characteristic of idioms is that they make execution performance bumpy. If you write exactly the right expression, you get great performance. A slight deviation from the idiom (perhaps redundant parentheses or an imbedded assignment) and you resort to normal interpretive execution.

Finally, different implementations may recognize different idioms (or none at all) and so have different performance characteristics. This can make selection of optimum coding techniques hard but no worse than different special case code within primitives on different implementations. The best advice is to code what you consider good *APL* style and demand good performance from the supplier of your implementation.

What constitutes a recognizable idiom?

APL2's implementation of idioms is limited by the desire to impose no performance penalty on non-idiom execution and by the difficulty of recognizing arbitrary expressions. The first restriction is that an expression containing only one primitive is not considered an idiom even though it may be in a limiting sense. Thus, `+/V` is not a recognizable idiom.

Recognizable idioms are classified according to the valence of the rightmost primitive in the expression which is called the root function of the idiom. Thus, an idiom is classified as a monadic idiom if its rightmost function is monadic and as dyadic if its rightmost function is dyadic. In normal right to left execution, the rightmost primitive in an idiom is encountered first. The fact that this is the start of an idiom is recorded in the internal representation of the *APL* program. The problem of unknown function valence is partially solved at this time. At execution time, valence is certain. If the function is marked at the start of a monadic idiom but a left argument has been coded, the idiom is ignored and and the function is called as an ordinary dyadic primitive. If the function is marked as the start of a dyadic idiom but no left argument has been coded, again the idiom is ignored. If the valence of the idiom matches the execution time valence of the function, the function is not called and, instead, the special execution routine for the idiom is called. A consequence of this scheme is that at the time the idiom execution routine is called, the left argument of the rightmost function of a dyadic idiom has already been evaluated and thus does not take part in idiom execution.

Consider the following potential dyadic idioms:

```
((V⍳V)=⍳⍴V)/V
(+/∧\TXT=' ')+TXT
```

In each case, the left argument of the rightmost operation is the expression in parentheses which will have been evaluated before the right operation is ready for evaluation. Thus, neither of these common idioms can be recognized by *APL2*.

On the other hand, these sub-expressions can be recognized:

```
(V⍳V)=⍳⍴V
+/∧\TXT=' '
```

The first is a monadic idiom and the second is a dyadic idiom.

For all idioms, monadic or dyadic, the right argument of the rightmost operation has already been evaluated. This does not normally matter but it does mean that:

```
(V⍳V)=⍳⍴X
```

will be recognized as an idiom if:

```
V←X    or    X←V
```

were executed first. It is the value of the right argument not the name that participates in the idiom.

A bumpy problem

Suppose you're interested in deleting blanks from a character string. `+/∧\TXT=' '` is a recognized idiom and could be used in this computation. Suppose you wrote `+/∧\' '=TXT` instead. Is this a recognized idiom? Not necessarily! Suppose you're interested in trailing blanks and use one of these expressions:

```
+/∧\⌽TXT=' '
+/∧\⌽' '=TXT
+/∧\' '=⌽TXT
```

To *APL* these are all different idioms. *APL2* recognizes all of these. In general, an attempt is made to recognize all reasonable permutations of an idiom so that you can code what you feel is the most natural with no performance penalty.

Recognized Idioms in APL2/370

Here is a partial list of recognized idioms classified by root function:

Comma

```
⍴,A
0 relational ⍴,
```

Equal

```
∧\A=B
∧\⍬A=B
+/∧\A=B
+/∧\⍬A=B
```

Not equal

```
∨\A≠B
∨\⍬A≠B
+/∨\A≠B
+/∨\⍬A≠B
```

Rho

```
⍴⍴A
0∈⍴A
1↑⍴A
¯1↑⍴A
1↓⍴A
¯1↓⍴A
↑⍴A
×/⍴A
×/¯1↓⍴A
⍳⍴A
⍳⍴⍴A
⍳¯1↑⍴A
⍳1↑⍴A
N relational ⍴A
N relational ⍴⍴A
(A⍳A)=⍳⍴A
⍬≡( )⍴A
```

Iota

```
⍬≡A×⍳B
(⍳⍴A)=A⍳A
```

Take

```
→(  )↑A
```

Drop

```
↑N↓A
```

Epsilon

```
~A∈B
```

Slash or slash bar

```
→(  )/A
A⍳⌈/A
A⍳⌊/A
```

Equal Underbar

```
A relational ≡ A
```

Each

```
⍴¨⍴¨A
↑¨⍴¨⍴¨A
```

Period

```
+/A∘.=B
```

Backslash

```
+/∧\A
```

Enclose

```
↑0⍴⊂A
```

Magnitude

```
+/|A
-/|A
⌈/|A
⌊/|A
+≠|A
-≠|A
⌈≠|A
⌊≠|A
```

Conclusion

Use of idioms enhances readability, writeability, understandability, and performance of *APL* programs. *APL2* idiom recognition can recognize a subset of the possible idioms. This paper does not list all idioms because of the author's desire for people to use the constructions that they feel are the most appropriate and most elegant. Implementers of *APL* systems should make the desirable and elegant *APL* expressions run efficiently.

References

(Iv1) Iverson, Kenneth, "A Dictionary of *APL*," APL Quote Quad, Vol. 18, No. 1, Sept. 1987