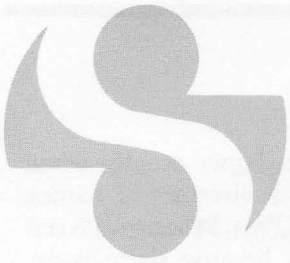
JANUARY/FEBRUARY 1982
Volume 10/Number 1



I.P. Sharp
newsletter

technical supplement 36

A REGULAR EXPRESSION PATTERN MATCHING PROCESSOR FOR APL

Mark R. Dempsey, Rochester

Leslie H. Goldsmith, Toronto

Editor's note: This paper originally appeared in *APL Quote Quad*, Volume 12 Number 1, September 1981.

Authors' note: An announcement will be made on the SHARP APL system when the pattern matching processor is available. A few changes to our original paper have been incorporated into this reprint.

Abstract

Regular expressions are a powerful notation for expressing text patterns. This paper discusses classical regular expressions and their extension into the domain of APL. These extensions are manifest in terms of locator templates, which describe patterns to be searched for, and action templates, which specify an action to be performed when a match is encountered. An algorithm for implementing these concepts is briefly discussed. The algorithm compiles a template into code which is then interpreted in the context of a specific text vector to be searched. Numerous examples are espoused to demonstrate the utility of pattern matching over a wide range of problems.

1. Introduction

String searching, like sorting, has become one of the classical software engineering problems. In its most elementary form, the problem consists of finding a string of characters (often a word or a phrase) within a larger string that may or may not contain it. Even with this simple starting point, a number of questions immediately present themselves. Do we find *all* occurrences of the target string, or do we stop after having found the first match? What do we *do* once the desired match or matches have been located? In answer to this last question, one possibility is just to return the positions in the text where each match occurred. But frequently, it is not these indices that are of direct interest. They may be of ancillary importance if what we are really trying to do is change every occurrence of the target string to something else, or if we simply want to display the line or the context associated with each occurrence.

The basic string search problem can get arbitrarily more complex if we do not know *exactly* what string of characters we are looking for. For example, suppose we wish to search a directory of names (arranged last name first) for all persons whose initials are "MD" or "LG". This becomes a very difficult problem if it is viewed from the position of standard string searching, because there is an element of uncertainty in exactly what names we will find, even though we know precisely what we are looking for. The key is that we are searching for a *pattern* of characters that belongs to a particular grammar, rather than a definitive string. This notion is prevalent in string processing languages such as SNOBOL [7], and has been promulgated in earlier work by Thompson [12], Aho and Corasick [2], and Richards [11]. Pattern matching is also employed by the Multics **qedx** editor [10] and by the UNIX **grep** [13] and **awk** [3] commands.

Regular expressions provide an extremely powerful framework within which string search patterns can be defined. The kernel of this power and flexibility is a special metalanguage which embodies the search specifications. Any pattern that can be represented in terms of this extended regular expression language can be searched for, by assembling it into a simple **locator template**. An optional **action template** may also be specified, to evoke some particular action (such as replacing the original match or compiling detailed information about each occurrence) wherever the locator template is satisfied. The approach is capable of easily performing such previously complex tasks as recognizing token or word boundaries, parsing command lines, rewriting the syntax of expressions, and effecting transformations on functions.

2. Formal Regular Expressions

Regular expressions are a notation which is suitable for describing syntactic tokens in a language. This section introduces some definitions and terms which are useful in discussing languages. It then describes classical regular expressions, and briefly explains their formal relationship to automata which can be simulated programmatically. A rigorous treatment of finite-state machines may be found in [1] and [4].

2.1 Strings and languages

An **alphabet** is a finite, unordered set of symbols. A **string** over an alphabet A is a finite sequence of symbols from A. The **length** of a string is the total number of symbols in it. For example, the set `{0, 1}` forms the binary alphabet, and the string "`00101`" is a string of length 5 over the alphabet. A **language** is a set of strings over an alphabet. `{010, 00101}` is a language consisting of two strings over the binary alphabet. The **empty** or **null string**, denoted by ϵ, is a special string of length zero that is valid over any alphabet.

If x and y are strings, then the **concatenation** of x and y, written xy, is the string formed by following the symbols in x by those in y. The concatenation of the null string with any string is that string; more formally, $\epsilon x = x\epsilon = x$ and, in particular, $\epsilon\epsilon = \epsilon$.

Concatenation is analogous to a noncommutative product operator, which leads to the consideration of exponentiation of strings as repeated concatenations. For example, $x^1 = x$, $x^2 = xx$, $x^3 = xxx = x^2x$, and so on. As a useful convention, we take x^0 to be ϵ for any string x, in much the same way as $M^0 = I$ for any matrix M and conformable identity matrix I. ϵ is thus the identity string for concatenation (and hence for exponentiation).

The concatenation of two languages L and N is defined to be the set consisting of all possible concatenations of sequences from L with sequences from N. Exponentiation of languages follows directly from this definition and from the preceding discussion. In particular, the concatenation of a language L with itself, written LL or L^2, is the set of all strings formed by concatenating two strings from L.

The **Kleene closure**, or simply the **closure** of a language L, denoted L^*, is the language $L^* = \bigcup_{i=0}^{\infty} L^i$. The **positive closure** of L, denoted $L+$, is the language $L+ = \bigcup_{i=1}^{\infty} L^i$ and is equivalent to LL^*, the concatenation of L and the closure of L.

If x is a string, then any string formed by discarding zero or more symbols from the end of x is called a **prefix** of x. A **suffix** of x is any string formed by discarding zero or more leading symbols from x. A **substring** of x is any string formed by deleting a (possibly empty) prefix and suffix from x. For example, "abc" is a prefix of "abcde", "cde" is a suffix, and both "bc" and "bcde" are substrings. For any string x, both x and ϵ are prefixes, suffixes, and substrings of x. Moreover, any prefix or suffix of x is clearly also a substring of x, but the converse is not true.

2.2 Definition of regular expressions

With these definitions as a formal framework, we can now describe regular expressions themselves. Each regular expression denotes a language. If A is an alphabet, then the regular expressions over A and the languages they denote are defined recursively as follows:

1. ϵ is a regular expression denoting the set $\{\epsilon\}$.
2. For each arbitrary c in A, c is a regular expression denoting the set $\{c\}$, a language with only one string.
3. If p and q are regular expressions denoting the regular sets (languages) P and Q, respectively, then:
   a) $p|q$ is a regular expression denoting $P \cup Q$.
   b) pq is a regular expression denoting PQ.
   c) p^* is a regular expression denoting P^*.
4. Nothing else is a regular expression.

Rule 3 above establishes the inductive nature of this tripartite recursive definition, while Rules 1 and 2 form the elementary basis for defining primitive regular expressions.

2.3 Finite automata and regular expressions

Our primary interest in regular expressions is their applicability to pattern matching. Suppose we have specified a language by means of a regular expression, and we are given some string x. We wish to determine if x is in the language denoted by the regular expression. One of the best theoretical ways to do this is to construct a generalized transition diagram from the expression. This diagram is called a **nondeterministic finite-state automaton** (NFA). An NFA to recognize the language `(0|1)*100` is shown in Figure 1.

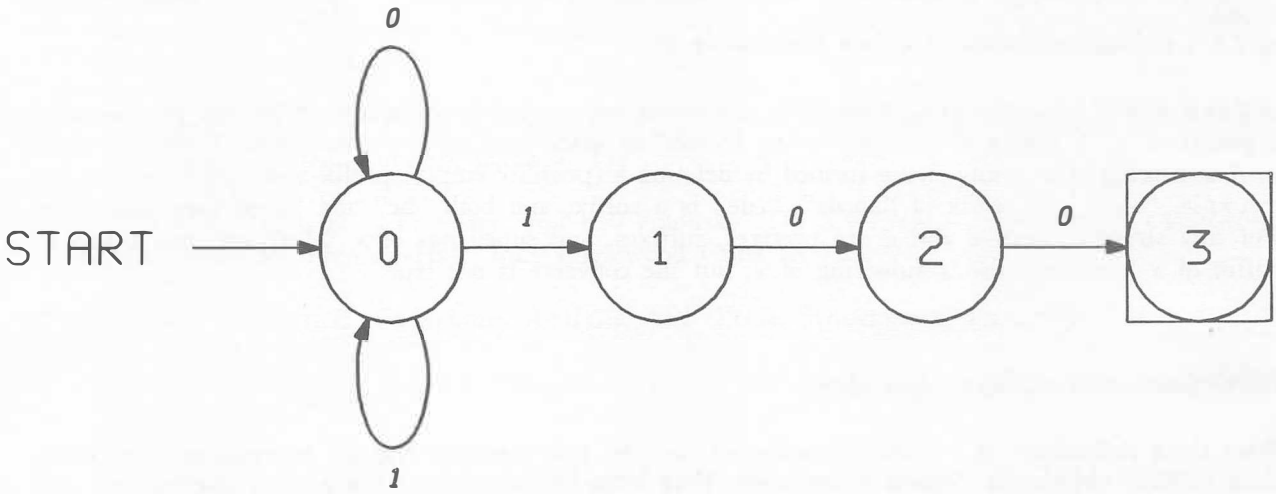


Figure 1. A nondeterministic finite-state automaton.

An NFA is simply a labeled directed graph, whose nodes represent machine **states**. One state is distinguished as being the **start state**, and one or more states are marked as **accepting states**. In Figure 1, State 0 is the start state and State 3 is the accepting state. The machine is nondeterministic because, for a given character in a given state, there may be zero or more choices of next move; in particular, the same character can label more than one transition arc out of a state. An equivalent **deterministic finite-state automaton** can always be constructed from an NFA, but may have as many as 2^n states for an NFA that has n states, so the conversion is not always efficient.

For an implementation of a regular expression recognizer in APL, the use of an NFA stymies the array-oriented power of the language, by forcing the algorithm to be decomposed to too fine a level of granularity. Indeed, while formal NFA's serve well for exposition, they would be a singularly inefficient means of implementing a pattern match in APL. The situation becomes even more exaggerated in the next section, when the regular expression notation is expanded in terms of its power and complexity to become better tailored to applications in the APL environment. An implementation scheme which lends itself more readily to APL is a nondeterministic machine that is only partially driven by state transitions. This implementation approach will be addressed later.

3. The Template Metalanguage

3.1 Locator templates

Literal characters and metacharacters

A pattern consists of one or more **pattern components**, each of which is built up from some combination of literal characters and metacharacters. Every character in a pattern is said to be in **literal mode** or in **regular expression mode**. In literal mode (the default), metacharacters have no special meaning and are interpreted literally (the "escaping" metacharacters themselves are exceptions to this). In regular expression mode, metacharacters are recognized as special characters.

A string of text is placed in regular expression mode by enclosing it in braces ({}). The diaeresis (¨) has the effect of complementing the mode of the character which follows it. If "⋆" and "+" are metacharacters, then in the pattern "*A⋆{B+}C*", "+" is treated as a metacharacter and "⋆" is not: The converse is true in the pattern "*A¨⋆{B¨+}C*". Since the examples in this section are brief, we assume that they are enclosed in braces unless otherwise noted.

Single-character pattern components

The metacharacter "*?*" matches any character except newline. Thus the pattern "*T?N*" matches the strings "*TAN*", "*TBN*", "*TCN*", and so on. If the object string is being interpreted as function text (see Section 4.1), then "*?*" matches any character except statement end or line end.

The newline character is represented by the metacharacter "¬", and is used to locate occurrences of a pattern at the extrema of a line. For example, "¬*L*10" matches occurrences of *L*10" which begin a line.

The pattern component "[**ccl**]" specifies any character from the character class represented by **ccl** (see [9]). There are three useful forms of the character class specification: "[**cs**]", "[**~cs**]", and "[**cs1~cs2**]". The first form matches any character from the character set **cs**. The second form matches any character which is not in the set **cs**. The final form matches any character from the set **cs1** which is not in the set **cs2**. A character set is defined by either enumerating its contents, or by specifying a range of characters from the following ordered set:

A-Z Δ *a-z* Δ 0-9

For example, the pattern "*T*[*AEIO*]*N*" matches the strings "*TAN*", "*TEN*", "*TIN*", and "*TON*"; "[*A-Z*67]" matches any letter of the standard alphabet or the digits "6" or "7"; and "[*A-9~0*]" matches any letter of the extended alphabet or any digit, except "0".

Alternation and elision

The metacharacter "|" denotes alternation. For example, the pattern component "*UP*|*DOWN*" matches the string "*UP*" as well as the string "*DOWN*". Parentheses may be used to group compound expressions controlled by "|".

A component suffixed by the metacharacter "-" denotes an optional expression. For example, "*THE* (*OLD*)-*DOG*" matches the string "*THE OLD DOG*" as well as the string "*THE DOG*". Note that, unless the optional phrase is a single-character component, it must be grouped inside parentheses.

Closure

The metacharacter "⋆" specifies Kleene closure, and matches zero or more occurrences of the preceding phrase. For example, the pattern "*THE (OLD) ⋆DOG*" matches the strings "*THE DOG*", "*THE OLD DOG*", "*THE OLD OLD DOG*", and so on. Moreover, "*?⋆*" matches an arbitrary length sequence of non-newline characters. The metacharacter "+" specifies positive closure, and matches one or more occurrences of a phrase. As with elision, non-singleton components must be grouped inside parentheses.

A closure pattern matches the longest possible sequence of characters. For example, with the pattern "*AB⋆*" and the string "*ABBB*", the closure component matches all three "*B*"s, and not just the first one.

Miscellaneous components

Several component types specifically designed to process APL function text are provided.

The metacharacter "◇" is the statement delimiter character, and matches the beginning or end of an APL statement.

The underscore "_" is used as the context delimiter character. A pattern bordered by "_" is treated as a syntactic element [6], and the *context* in which a potential match appears determines if it is an acceptable match. For example, the pattern "*_YES_*" will match the word "*YES*", but not "*YESTERDAY*" or "*EYES*". If "_" appears on only one side of a phrase, only the left or right context of a potential match is checked. The context delimiter works in a similar manner with numeric strings.

The "α" metacharacter matches any identifier, and is functionally equivalent to the pattern "[*A-Δ*][*A-9*]⋆". In a similar manner, "ω" matches any numeric string, and is equivalent to the pattern "[¯0-9][¯.*EJ*0-9]⋆".

3.2 Action templates

An action template specifies what processing is to take place when a locator template encounters a match in an object string. The escaping metacharacters "{", "}", and "‾", and the newline character "-↓", have the same meaning in an action template as they do in a locator template. If an action template contains only literal text, then that text replaces the matched substring in the object string.

It is possible to identify a portion of matched text and to refer to it in an action template. This is accomplished by enclosing the relevant portion of the locator pattern in the metacharacter pair "⊂⊃". These "tagged" strings can then be referenced in the action template by the construct "⊂**n**⊃", where **n** is an integer. Each occurrence of this construct is replaced by the text that matched the **n**th tagged pattern. As an example, with the locator template "{⊂*?⋆*⊃,⊂*?⋆*⊃,⊂*?⋆*⊃}" and the action template "{⊂3⊃,⊂2⊃,⊂1⊃}", the object text "*FIRST, MIDDLE, LAST*" would be transmuted to "*LAST, MIDDLE, FIRST*". Pattern tags may be nested in a locator template, but not in an action template. A particular tag may be referenced any number of times in an action template, or not at all.

In addition to literal and regular expression modes, action templates permit a third option, **evaluated mode**, which is a subclass of the latter. An expression is placed in evaluated mode by enclosing it between occurrences of the metacharacter "∇". A string in evaluated mode must conform to the syntax of a valid APL expression, with two additional constructs being permissible. First, if an assignment

arrow follows the opening del of the expression, then the result of the executed expression is inserted into the replacement template in the position occupied by the evaluated string. This is akin to the macro expansion concepts used in data-driven code generation systems [5]. Second, a string in evaluated mode may contain pattern tags, which are replaced with the matched object text before the string is executed.

Strings in evaluated mode may also reference three template **descriptor variables**, which have the appearance of system variables but denote elementary surrogates that are recognized by the pattern matching processor. These variables are integer scalars which contain information about the relative location of the current match within the object string. The first of them, *⎕LN* (line number), contains the line number on which the match was found. *⎕CP* (cursor position) contains the origin-zero index of the first character of the match relative to the beginning of the **line**. Finally, *⎕CN* (character number) contains the origin-zero index of the first character of the match relative to the beginning of the searched **text**.

4. Implementation

4.1 Functional specifications

The function *PMATCH* implements the regular expression pattern match. *PMATCH* is ambivalent, and returns an explicit result. Its syntax is:

```
Z←['NAME'] PMATCH 'TSPEC'
```

The right argument TSPEC is a character vector containing the locator template and an optional action template, separated by a delimiter. The first character of TSPEC specifies the delimiter, and should be chosen so that it does not occur elsewhere in the argument in unescaped form (literal mode). For example, the argument *'/HONDA'* locates all occurrences of the character string "*HONDA*". *'/HONDA/YAMAHA'* replaces all such matches with the string "*YAMAHA*".

The optional left argument NAME specifies the name of the global variable which contains the text to be processed. If a name is not provided, then the default variable *ΔTXT* is assumed. The left argument may also contain certain search qualifiers which control the behavior of *PMATCH* during the application of the locator template. Permissible qualifiers, which may appear anywhere in the argument, are as follows:

- ∇ Causes *PMATCH* to interpret the text to be searched as function text. This affects the processing of certain locator template components, in particular "*?*".
- ⍝ Inhibits any pattern component *except* "*?*" from matching a character interior to a comment. The lamp symbol beginning a comment may still be matched by the character "⍝".
- ' Inhibits any pattern component *except* "*?*" from matching a character interior to a quoted string. The quote symbols beginning or ending a character constant may still be matched by the character "'".

The formal result z is a two-column non-negative integer matrix, each row of which specifies the origin-zero location of the start of a match in the *original* text, and the number of characters encompassed by the match. For example:

```
      ΔTXT←'FEEL FREE TO SEARCH ME.'
      PMATCH '/{[A-Z]*E[A-Z]*}'
 0  4
 5  4
13  5
20  2
```

4.2 Overview of the algorithm

Typical algorithms for performing regular expression pattern matching can be decomposed into two steps. The first step consists of analyzing the pattern and compiling a well-defined internal representation which is conveniently manipulated. The second step consists of evaluating this internal representation in the context of a specific text vector which is to be searched. This division is both logically modular and expedient. Further, the first step is independent of the searched text and therefore theoretically need be executed only once for a given pattern. This can result in significant cost savings in a system which contains frequent evaluations of a hard-coded template.

This specific implementation consists of a regular expression compiler and an interpreter. The compiler translates a regular expression locator template into an APL function. The interpreter uses this function to locate matches in a given text vector. When the interpreter detects a match, it also honors an action template if one was provided.

The remainder of this section discusses a prototype implementation algorithm in somewhat more detail.

The compilation phase

The compilation algorithm used in this implementation differs from those of typical regular expression processors. The output of the compiler is not a static representation of the input pattern, but rather an APL function. This function, when executed, produces a data structure which represents relationships between the pattern and the specific text being searched. The data structure is sufficiently descriptive to eliminate the need for any further examination of the text vector or template.

The process of compiling the function involves three logical steps: the lexical scan of the pattern, the syntactic decomposition of the regular expressions contained within the pattern into elementary components, and the actual generation of the APL code. Since the regular expression language can be described by a context-free grammar, all three steps may be carried out simultaneously using an LALR(1) table-driven parser. The code generated corresponds to a prefix Polish representation of the decomposed input pattern.

The interpretation phase

The first step in the interpretive portion of the algorithm is executing the relation-building function generated by the compiler, which defines certain key variables. The goal of the remainder of the algorithm is to traverse this internal data structure to determine the final matches.

This process is unfortunately inherently nondeterministic. Most of the syntactic constructs supported by the extended regular expression notation are relatively straightforward to evaluate; the notable exceptions to this are the closure constructs. Recall that closure matches the longest possible substring in cases where there is a choice. This implies an associated iteration on the basic pattern over which closure is being applied until the closure fails. At this point, an attempt is made to match the remainder of the pattern against potential matches in the text. If the pattern fails, however, it cannot be assumed that there are no matches, for a match might still occur if the scope of the closure were shortened by some non-zero multiple of the length of the associated elementary group.

To illustrate this point, consider the pattern "*{AB*BB*B}*" and the text vector "*ABBBBBAB*". An initial evaluation of the first closure in this template would yield a match to the five consecutive "*B*"s, and a second match to the single "*B*". However, both such matches cause the pattern to fail on the next "*B*", so that the length of the first closure must be shortened by one to satisfy the third component of the template. The second closure is immediately satisfied if its length is taken to be zero. However, the final "*B*" in the pattern still causes both matches to fail when it is encountered. It is therefore apparent that the first match succeeds only if the first closure is of length three and the second is of length zero; the second erstwhile match clearly fails. In general, it is neccesary to recursively restrict the right scope of each preceding closure until all such closures are of length zero. Then, and only then, can we be certain that a match has truly failed. This nondeterministic backtracking adds considerable complication and machinery to the algorithm. Alternation likewise introduces recursive backtracking, with an outer product style enumeration of disjoint alternation groups.

If a Boolean relation matrix *REL* is set up so that 1's in a particular row mark the start of a match of the corresponding elementary component, then the pattern match problem can be reduced to one of finding the longest paths from the first row of the matrix to the last. This is, in effect, the dual of the classic least cost graph-theoretic problem [1] with the cost functions being computed according to the lengths of each component of the pattern. In the simplest case, where the pattern does not include closure or alternation, the walk through the matrix can proceed from arbitrary position *REL*[*I*;*J*] if and only if *REL*[*I*;*J*]∧*REL*[*I*+1;*J*+*LA*[*I*]] is *true*, where *LA*[*I*] is the length attribute of elementary component *I*. If the pattern contains closure or alternation components, then closure defaults to the longest possible substring match such that *REL*[*I*+1;*K*] is **true** for some *K*←*LA*[*I*]×*N* where *N* is an integer minimally bounded by 0 for Kleene closure and 1 for positive closure; and alternation defaults to the first alternative in a given group.

If closure fails, then recursive backtracking must be employed, as previously described, until either the match succeeds or all earlier closures are null. If alternation fails, then subsequent alternatives in a group must be considered. If none of the alternatives in a group satisfies the match, then recursive backtracking must be employed until either the match succeeds or all possible combinations of alternatives have been exhausted.

When a match succeeds, the presence of an action template is tested. If one exists, then tagged numbers are replaced by the correspondingly matched strings in the text and any match descriptor variables (□*LN*, □*CP*, and □*CN*) are replaced by the appropriate values. The evaluation of the action template then becomes a straightforward proposition.

5. Pattern Matching Applications

This section presents a number of practical applications of *PMATCH*. The examples are divided into two categories: those which solve problems pertaining to general text processing, and those which deal specifically with APL function text. In all examples, the text to be processed is assumed to be in the variable Δ*TXT*.

5.1 General text processing

Table 1 illustrates some sample text patterns and their effect. Other examples which benefit by more elaborate explanation are presented below.

Table 1. Some simple search patterns.

Syntax and Pattern	Effect
Z←PMATCH '/**pat**'	Locate all occurrences of **pat**
Z←PMATCH '/**pat**/**new**'	Replace all occurrences of **pat** by **new**
Z←PMATCH '/ ¨+/ '	Remove multiple contiguous blanks
Z←PMATCH '/{⊂[.:?!]⊃ *}/{⊂1⊃ }'	Follow punctuation by exactly two blanks
Z←PMATCH '/19{7[6-9]|80}/1981'	Update references to recent years

Document editing

∘ A common grammatical error is the use of the double negative. All occurrences of a specific kind of this language blunder can be detected and corrected as follows:

```
Z←PMATCH '/{CANNOT ⊂[A-Z]*⊃ BUT ⊂[A-Z]*⊃}/{CANNOT ⊂1⊃ ⊂2⊃ING}'
```

For example, the improper construction "cannot help but wonder" would be modified to make use of the present participle of the intransitive verb "wonder".

Monotone series alteration

∘ Each series of contiguous "*A*"s (for example) in a text vector may be increased in length by one by an expression of the form:

```
Z←PMATCH '/{⊂A+⊃}/{⊂1⊃A}'
```

Conversely, the expression below decreases the length of each non-singular series by one:

```
Z←PMATCH '/{⊂A+⊃A}/{⊂1⊃}'
```

If it is desirable that a series be permitted to vanish entirely, then the Kleene closure (*) may be used in place of the positive closure (+).

Context display

∘ All occurrences of a pattern contained in a character vector *PAT* can be displayed in a useful format with:

```
Z←PMATCH '/',PAT,'/{∇((10⍴'' ''),ΔTXT,10⍴'' '')[⎕CN-10-⍳20+⍴PAT],'' '',⍕⎕CN∇}'
```

This displays each occurrence and the ten characters on either side of it, followed by the index of the match within the text vector.

5.2 APL function text processing

The examples in this section illustrate the use of *PMATCH* on function-transformation-related applications. Some of these examples reduce the space, symbol table, and execution overhead of a program, at the expense of its readability. This is justified if an unadulterated maintenance copy of the source is retained for perusal and modification.

Comment massaging

◦ On some APL systems, comments may appear on the same line as, and to the right of, executable statements. The template shown below may be used to remove comments from a function, and to remove the concomitant bracketed line number where the comment fills the entire line:

```
Z←'∇⍵''' PMATCH '/{⊣⍤[⍵⍤] ⋆⍵?⋆|⍵?⋆}/'
```

◦ Frequently, useful results can be obtained by using *PMATCH* to perform multi-pass operations. The example shown in Figure 2 aligns the start of the text of comments so that each begins in the same column. This is done in two passes: the first pass collects information about the location of each comment, while the second pass uses this information to modify the function display.

```
    ∇ ΔTXT←ALIGNCOMMENTS ΔTXT;CNT;POS;SINK
[1]   POS←⍳0 ◇ CNT←0 ⍝ INITIALIZE HIT VECTOR AND COUNTER
[2]   SINK←'∇⍵''' PMATCH '/⍵/{∇POS←POS,⎕CP∇}' ⍝ LOCATE ALL COMMENTS
[3]   POS←(⌈/POS)-POS ⍝ COMPUTE NO. OF BLANKS TO INSERT
[4]   SINK←'∇⍵''' PMATCH '/⍵ ⍤⋆/⍵ {∇←POS[CNT←CNT+1]⍴'' ''∇}' ⍝ INSERT BLANKS
    ∇

      ALIGNCOMMENTS 1 ⎕FD 'ALIGNCOMMENTS'
    ∇ ΔTXT←ALIGNCOMMENTS ΔTXT;CNT;POS;SINK
[1]   POS←⍳0 ◇ CNT←0 ⍝                                  INITIALIZE HIT VECTOR AND COUNTER
[2]   SINK←'∇⍵''' PMATCH '/⍵/{∇POS←POS,⎕CP∇}' ⍝         LOCATE ALL COMMENTS
[3]   POS←(⌈/POS)-POS ⍝                                 COMPUTE NO. OF BLANKS TO INSERT
[4]   SINK←'∇⍵''' PMATCH '/⍵ ⍤⋆/⍵ {∇←POS[CNT←CNT+1]⍴'' ''∇}' ⍝ INSERT BLANKS
    ∇
```

Figure 2. Aligning comments in a function.

Cross-reference kernel

◦ A substantial portion of the work in any cross-reference utility is the extraction of the raw reference information from the source program. All of this data can be collected in a single step using *PMATCH*:

```
IDS← 0 0 ⍴LNS←⍳0
Z←'∇⍵''' PMATCH '/{⊂⍺([:←[])-⊃}/{∇IDS←IDS RCAT ''⊂1⊃'' ◇ LNS←LNS,⎕LN-1∇}'
```

RCAT is a utility function which performs a row-wise catenation of its two arguments, and is used only for clarity here. After calling *PMATCH*, the identifier name and object class information in *IDS*, and the line numbers in *LNS*, can be further massaged and formatted into the desired report.

Compiling branching statements

∘ Many APL programmers use cover functions, such as those illustrated in direct definition form [8] below, to improve the readability of branching statements:

```
          IF:ωρα          UNLESS:ω↓α

Examples: →L10 IF A<B     →DONE UNLESS ×ρVEC
```

To rewrite statements of the form →**label** *IF* **expression** as →(**expression**)ρ**label**, the template below may be used:

```
Z←'∇⍵' PMATCH '/{→⊂α⊃ IF_⊂?*⊃◇}/{→(⊂2⊃)ρ⊂1⊃}'
```

Statements of the form →**label** *UNLESS* **expression** may be altered to →(**expression**)↓**label** as follows:

```
Z←'∇⍵' PMATCH '/{→⊂α⊃ UNLESS_⊂?*⊃◇}/{→(⊂2⊃)↓⊂1⊃}'
```

These transformations can introduce redundant parentheses into a function. Such parentheses can be removed with a straightforward second pass:

```
Z←'∇⍵' PMATCH '/({⊂α|ω|α¨[?*¨]⊃})/{⊂1⊃}'
```

∘ The inverse operation of rewriting simple branching statements to use *IF* and *UNLESS* is also possible:

```
Z←'∇⍵' PMATCH '/{→⊂?*⊃[ρ↑¨/]⊂α|ω⊃◇}/{→⊂2⊃ IF ⊂1⊃}'  or
Z←'∇⍵' PMATCH '/{→⊂?*⊃↓⊂α|ω⊃◇}/{→⊂2⊃ UNLESS ⊂1⊃}'
```

Replacement of constants by variables

∘ The most common numeric constants in APL programs are ¯1, 0, and 1. A template which replaces syntactic occurrences of these constants by the variables *QN*1, *Q*0, and *Q*1, respectively, is given below:

```
Z←'∇⍵' PMATCH '/{_⊂¯1|0|1⊃_}/{Q∇←(⊂1⊃≥0)↓''N'',⍕|⊂1⊃∇}'
```

Note that this does not alter occurrences of these numbers where they appear as part of a longer constant, as in 1 0 */B* or ¯11↑*M*.

Deleting flagged function lines

∘ Language compilers often permit a programmer to specify that certain lines of code are to be conditionally compiled, a facility which can be useful for APL programs as well. For example, if conditional lines are specified by a comment beginning with "⍝X", then the following expression can be used to remove them (retaining a line-label if one exists):

```
Z←'∇⍵''' PMATCH '/{⊂⊣¨[ω¨] *(α:)-⊃(?|◇)*⍝X?*}/{∇←('':''∈''⊂1⊃'')¨/''⊂1⊃''∇}'
```

Renumbering function lines

∘ The template shown below sequentially renumbers the bracketed lines of a function:

```
Z←'∇' PMATCH '/{⊣¨[ω¨]}/{⊣[∇←⎕LN-1∇]}'
```

Diamondizing and undiamondizing lines

∘ Assuming no use of absolute line numbers or ⎕*LC*-relative branches, two consecutive non-header program lines *L*1 and *L*2 can be combined by the use of the diamond separator if and only if *L*1 does not end in a comment and *L*2 is not a labeled line. A template which implements this rule to diamondize a function is:

```
Z←'∇A''' PMATCH '/{⊂~(A?*)⊃⊣¨[ω¨] *⊂~(α:)⊃}/{⊂1⊃◇⊂2⊃}'
```

∘ The dual template which undiamondizes lines, putting each logical statement onto a separate physical line, is given below:

```
Z←'∇A''' PMATCH '/{⊂?*⊃⊂◇⊃⊂?*⊃}/{⊂1⊃∇←(~''A''∈⊂2⊃)↓''A⊂3⊃''∇}'
```

Each newly-introduced line is assigned the arbitrary line number 1; to properly renumber the lines within the function, the template given in the previous section may be employed.

Removing line-labels

∘ The function below removes all line-labels from a program, and replaces references to them with references to absolute line numbers:

```
    ∇ ΔTXT←ZAPLABELS ΔTXT;NOS;PAT;LBLS
[1]   LBLS← 0 0 ρNOS←⍳0
[2]   PAT←'∇A''' PMATCH '/{⊂α:⊃}/{∇←LBLS←LBLS RCAT ¯1↓''⊂1⊃'' ◇ NOS←NOS,⎕LN-1 ◇
      ''''}'
[3]   PAT←1↓,'|',LBLS ◇ PAT←(PAT≠' ')/PAT
[4]   PAT←'∇A''' PMATCH '/{⊂_',PAT,'_⊃}/{∇←NOS[(LBLS∧.=(1↓ρLBLS)↑''⊂1⊃'')⍳1]∇}'
    ∇
```

6. Conclusions

The problem of textual pattern recognition is pervasive throughout the field of computing. Since most computing input and output can be thought of as strings, an effective pattern matching program can immediately be applied to a rich set of text processing problems. Experience with *PMATCH* has shown that it can greatly simplify the solution to many problems which existing programs solve. Whereas previously a solution might have involved a specific, complex algorithm, it can now be expressed in terms of straightforward pattern templates which are easily modified to suit changed specifications.

7. Acknowledgements

We are indebted to Jerry Cudeck, Roland Pesch, Ed Stubbs, and particularly, Dave Markwick, for their careful reading of earlier drafts of this manuscript.

8. References

[1] Aho, Alfred V., J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, Reading, Mass., 1974.

[2] Aho, Alfred V. and M.J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search," *Communications of the ACM*, vol. 18, no. 6 (June 1975), pp. 333-340.

[3] Aho, Alfred V., B.W. Kernighan, and P.J. Weinberger. *Awk—A Pattern Scanning and Processing Language*, second edition, Bell Telephone Laboratories, Murray Hill, N.J., 1978.

[4] Backhouse, Roland C. *Syntax of Programming Languages*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1979.

[5] Dempsey, Mark R. "Using an APL Macroprocessor to Implement Generalized Software Systems," *APL79 Conference Proceedings* and *APL Quote Quad*, vol. 9, no. 4 (June 1979), part 1, pp. 286-293.

[6] Goldsmith, Leslie H. *Workspace Searching Using* 777 *WSSEARCH*, second edition, I.P. Sharp Associates Limited, Toronto, Canada, 1979.

[7] Griswold, Ralph E., J.F. Poage, and I.P. Polonsky. *The SNOBOL4 Programming Language* , second edition, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1971.

[8] Iverson, Kenneth E. *Elementary Analysis*, APL Press, Swarthmore, Pa., 1976.

[9] Kernighan, Brian W. and P.J. Plauger. *Software Tools*, Addison-Wesley Publishing Company, Reading, Mass., 1976.

[10] *The Multics Progammer's Manual*, Massachusetts Institute of Technology, MIT Press, Cambridge, Mass., 1974.

[11] Richards, Martin. "A Compact Function for Regular Expression Pattern Matching," *Software—Practice and Experience*, vol. 9 (1979), pp. 527-534.

[12] Thompson, Ken. "Regular Expression Search Algorithm," *Communications of the ACM*, vol. 11, no. 6 (June 1968), pp. 419-422.

[13] Thompson, Ken and D.M. Ritchie. *The UNIX Programmer's Manual*, Bell Telephone Laboratories, Murrary Hill, N.J., 1974.

Appendix A—Template Metalanguage Summary

A.1 Locator Template Components

rx	Any regular expression
rrx	Restricted regular expression: any regular expression not containing a closure component
cp	Any single-character pattern

Escape Mechanisms

{	Switch to "regular expression" mode
}	Switch to "literal" mode (default)
¨	Complement the mode of the next character

Single-Character Components

c	The character specified by **c**
?	Any character except newline (or statement end if searching function text)
[**c1c2c3**]	Any of the characters **c1 c2 c3**
[**c1-c2**]	Any letter of the alphabet or digit between and including **c1** and **c2** (character range)
[**cs1~cs2**]	All characters in set **cs1** which do not appear in set **cs2** (**cs1** and **cs2** may include a character range)
[**~cs2**]	All characters in the global character set which do not appear in character set **cs2**

Alternation, Elision, and Closure

rx1|**rx2**	Anything which matches either of the regular expressions **rx1** or **rx2** (alternation)
cp- (**rx**)-	Zero or one occurrences (elision)
cp⋆ (**rrx**)⋆	Zero or more occurrences (Kleene closure)
cp+ (**rrx**)+	One or more occurrences (positive closure)

Miscellaneous Components

~	Pattern negation
◇	Statement delimiter
⊣	Line delimiter
_	Context delimiter
α	Any identifier
ω	Any number
⊂**rx**⊃	Pattern tag

A.2 Action Template Components

⊣	Newline
⊂**n**⊃	String matching pattern tag **n** in locator template
∇	Evaluated mode delimiter
←	Replace string with evaluated expression (recognized only when appearing immediately to the right of an odd-parity ∇).

CONTEST 12

John Bogar, Winnipeg

The problem at hand requires you to form a matrix, say TABLE, from two non-negative integer vectors α and ω. The $(+/\alpha)$ is equal to $(+/\omega)$ and if $M \leftarrow \rho\alpha$ and $N \leftarrow \rho\omega$, then TABLE has the following interesting (and useful) properties:

1. The dimensions of TABLE are (M,N)
2. $\alpha = +/$TABLE and $\omega = +\not/$TABLE
3. TABLE contains at most $(M+N-1)$ nonzero entries
4. TABLE[1;1]$\leftarrow MIN\{\alpha[1],\omega[1]\}$

An example probably best illustrates how the actual entries of TABLE are obtained. Suppose $\alpha \leftarrow$5 15 5 and $\omega \leftarrow$7 8 1 9, then we can set up our starting point as:

```
5  *  *  *  5
*  *  *  *  15
*  *  *  *  5
7  8  1  9
```

Now α[1] has been all used up, so the remainder of TABLE[1;] can be filled with zeroes and ω[1] adjusted by $\omega[1] \leftarrow \omega[1]-\alpha[1]$. So our table looks like:

```
5  0  0  0  0
*  *  *  *  15
*  *  *  *  5
2  8  1  9
```

The procedure from here involves working in a diagonal fashion from the northwest corner to the bottom left corner, comparing the $MIN\{\alpha,\omega\}$ in pairs, and making the necessary adjustments. Hence the next pair to consider is $MIN\{2,15\}$ which is 2. The adjustment 15-2=13 and the 2 is all used up so the remainder of its column is filled with zeroes. Now the table is:

```
5  0  0  0  0
2  *  *  *  13
0  *  *  *  5
0  8  1  9
```

This procedure is repeated until the final TABLE looks like:

```
5  0  0  0
2  8  1  4
0  0  0  5
```

So the challenge is to write a dyadic function that will take two non-negative vector arguments and return the matrix formed by 'the northwest corner rule' as an explicit result.

The application of this algorithm is in linear programming where TABLE is an initial basic feasible solution to the classical transportation problem. Suppose in our example that α represents the amount of goods in three warehouses and ω represents the supply of these goods ordered by four retail outlets. Then the table formed by the northwest corner rule is one unique way the goods can be distributed from the warehouses to the retail outlets. Of course this is not the only way, nor is it necessarily the best way, but such an optimal distribution is a further problem of linear programming (and would require additional information on the actual transportation costs) and is not significant to this contest.

As usual, entries will be judged in terms of execution efficiency and space used. $\square$*APPEND* entries as enclosed arrays to file 999 *CONTEST* no later than April 1, 1982. The first element of the enclosed array should contain the name, address and organization affiliation (and mailbox code if applicable), and the second component the $\square$*CR* of the function. Good luck.

A prize of $50 will be awarded to the person who is not an employee of I.P. Sharp submitting the best entry; while the best entry from an employee will receive a book of his or her choice.