# Pattern Matching in Icon*

*Ralph E. Griswold*

TR 80-25

October 1980

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS79-03890.

# Pattern Matching in Icon

## 1. Introduction

Persons who are accustomed to using pattern matching in SNOBOL4 for string processing usually encounter problems in using string scanning in Icon There is a natural tendency to try to formulate patterns in Icon, which is aggravated by the fact that some Icon string scanning functions closely resemble SNOBOL4 patterns However, attempts to translate SNOBOL4 patterns into corresponding Icon string scanning expressions may result in awkward, if not actually incorrect, results

Since SNOBOL4 is a long-established language and Icon is relatively new, it is natural to criticize Icon on this point (especially since it purports to be an improvement over SNOBOL4) As discussed in [11], Icon string scanning is an alternative to pattern matching, not a different kind of pattern matching However, pattern matching is a valuable method for formulating many string analysis operations

This paper shows how SNOBOL4 patterns can be modeled in Icon The result in effect constitutes a precise semantics for pattern matching in SNOBOL4 and also provides a model for more general approaches to pattern matching A knowledge of SNOBOL4 and Icon is assumed, see [13] and [1,12] for details

## 2. SNOBOL4 Patterns and Icon Expressions

Newcomers to Icon usually try to translate SNOBOL4 patterns into Icon expressions at the wrong level While there are rough correspondences between built-in patterns in SNOBOL4 and scanning functions in Icon, SNOBOL4 patterns cannot just be replaced by scanning functions in Icon Consider, for example, the SNOBOL4 pattern

```
        SPAN(LETTER)   WORD
```

An approximately equivalent Icon expression is

```
        word  =  tab(many(letter))
```

To avoid confusion between examples in SNOBOL4 and Icon, upper-case letters are used for SNOBOL4 while lower-case letters are used for Icon

In SNOBOL4 patterns usually are assigned to identifiers and placed in context with other patterns Examples are

```
        MWORD = SPAN(LETTER)   WORD
        GETWORD = BREAK(LETTER) MWORD
```

The assignment of patterns to MWORD and GETWORD might appear in a program preamble and then be used as follows

```
        NEXTW    TEXT ? GETWORD =                 F(ENDW)


                 processing of WORD


                                                   (NEXTW)
```

(The SITBOL [5] explicit pattern-matching operator ? is used here for clarity )

In Icon, the corresponding use might appear, slightly recast, as

```
scan text using
   while tab(upto(letter)) do {
       word = tab(many(letters))


       processing of word


       }
```

If, however, such processing is used in several parts of an Icon program, it may be better to use a procedure

```
scan text using
   while word = getword() do {


       processing of word


       }
```

where getword is

```
procedure getword()
   if tab(upto(letter)) then return tab(many(letter))
   else fail
end
```

Here, getword is a "scanning procedure", since, like scanning functions, it operates in a context in which &subject and &pos are supplied by an external scan-using expression

A somewhat more elegant solution is to formulate getword as a generator

```
procedure getword()
   local word
   while tab(upto(letter)) do {
      word = tab(many(letter))
      suspend word
      }
   fail
end
```

which can be used in context as

```
scan text using
   every word = getword() do {


       processing of word


       }
```

The development of the expressions above leading to getword as a generator is designed to show the level at which SNOBOL4 pattern matching and Icon string scanning meet  SNOBOL4 patterns generally correspond to Icon scanning procedures that are generators

One very useful aspect of patterns in SNOBOL4 is the ease with which they can be combined to form more complex patterns  For example,

```
GETPWORD = ANY(" , ,!?") MWORD
```

might be used to find a word immediately following a punctuation mark, while

```
GETBWORD = " " MWORD " "
```

might be used to find a word that is surrounded by blanks

In such cases, a pattern can be placed in various contexts without having to modify it On the other hand, Icon scanning procedures such as those given above usually cannot be placed in different contexts without modification An examination of the basic concepts of pattern matching shows how to design scanning procedures that can be used like patterns

## 3. Pattern Matching

### 3.1 Pattern Matching in SNOBOL4

Pattern matching in the SNOBOL4 sense has several important properties that allow patterns to be combined and used in various contexts without modification The implicit focus of attention provided by the subject and the cursor allow patterns to be constructed without specific reference to the strings they are to be applied to and without reference to the position at which they are to be applied Most patterns, when applied, change the position of the cursor and all patterns return the substring of the subject between the positions of the subject before and after they are applied (patterns that do not change the cursor return the null string) Furthermore, a pattern that fails to match leaves the cursor unchanged Patterns may be concatenated to form larger patterns as illustrated in the preceding section

The pattern matching process is complex and is discussed detail elsewhere [6,7,14] Briefly stated, patterns are matched starting at the left of the subject It is convenient to characterize a pattern as a sequence (concatenation) of subpatterns During pattern matching, the subpatterns are applied from left to right If a subpattern succeeds, the next subpattern to the right is applied If all subpatterns successfully match, the entire pattern match succeeds If a subpattern fails to match, backtracking occurs to the previous subpattern in the sequence Some patterns have alternative possibilities for matching and thus the previous subpattern may match another substring If so, pattern matching continues to the right with the next subpattern in the sequence If not, the cursor is restored to its position prior to the original match for that subpattern Backtracking to the next subpattern to the left then occurs, and so on The entire pattern match fails if the first subpattern in the sequence fails

### 3.2 Pattern Matching in Icon

Pattern matching in the SNOBOL4 sense can also be performed in Icon provided that certain paradigms and protocols are used

A *matching expression* is defined as follows An expression *e* is a matching expression if and only if

(a) it does not change the value of &subject,

(b) it returns the substring of &subject between the values of &pos before and after it is evaluated, and

(c) it leaves &pos unchanged if it fails (possibly restoring &pos during backtracking)

The two scanning functions tab(i) and move(i) are matching expressions and form the basis for most other matching expressions in Icon Most meaningful matching expressions are composed using other scanning functions A typical example is tab(find(s)) Note that =s, which is an abbreviation for tab(match(s)), is a matching expression

If $e_1$ and $e_2$ are matching expressions, then $e_1$ | $e_2$ and $e_1$ || $e_2$ are matching expressions On the other hand, $e_1$ & $e_2$ is not, in general, a matching expression, since the result of conjunction is the result of evaluating $e_2$, regardless of what $e_1$ matches

There are many other matching expressions that are less obvious For example, if *e* is a matching expression, then x := *e* is a matching expression Note that *any* expression that does not change the value of &subject or &pos is a matching expression, provided that it returns the null string (or, in a matching context, move(0)) Examples in SNOBOL4 are POS(I) and @S

The use of procedures is important in pattern matching Consider the following form of procedure

```
procedure p()
   suspend e
   fail
end
```

If *e* is a matching expression then p() is a matching expression Furthermore, *e* and p() produce the same results Thus p is a procedural encapsulation of *e* This form of encapsulation is used frequently in the examples that follow There are, of course, many other possible forms of matching procedures Note that, in general,

```
procedure p()
   return e
end
```

is not a matching procedure, since *e* cannot be reactivated to restore the value of &pos

## 4. Translating SNOBOL4 Pattern Matching into Icon

Translating SNOBOL4 pattern matching into Icon provides a good illustration of the correspondences between pattern matching in SNOBOL4 and string scanning in Icon As detailed in the following sections, most aspects of pattern matching in SNOBOL4 can be translated into Icon The translation is expressed through transformations on SNOBOL4 statements and expressions The translation is limited to pattern-valued expressions and their arguments, which may be string- or integer-valued expressions The translation of pattern-matching and replacement statements is included also, although only straight-line programs are handled No attempt is made to translate SNOBOL4 control structures, programmer-defined functions, nor the great variety of complex data types and operations of SNOBOL4 that are not related to pattern matching It is assumed that all SNOBOL4 constructions to be translated are semantically correct and that no built-in operations or patterns have been changed

### 4.1 Notation

The transformation $\delta$ converts SNOBOL4 statements into Icon procedure declarations and expressions The transformation $\rho$ converts SNOBOL4 pattern-valued expressions into Icon matching expressions The transformations $\sigma$ and $\iota$ convert SNOBOL4 string- and integer-valued expressions into corresponding Icon string- and integer-valued expressions, respectively *P*, *S*, and *I* denote SNOBOL4 expressions that are pattern-, string-, and integer-valued, respectively, while *X* denotes SNOBOL4 expressions of undetermined type P, S, and I denote SNOBOL4 identifiers that have pattern, string and integer values respectively Similarly, p, s and i denote corresponding Icon identifiers To simplify the notation, *s* and *i* are used to stand for $\sigma(S)$ and $\iota(I)$, respectively

### 4.2 Assignment Statements

The translation of SNOBOL4 assignment statements depends on the data type of the value assigned For string and integer values, the result is a corresponding Icon assignment expression

$\delta$(S = *S*) → s = *s*
$\delta$(I = *I*) → i = *i*

For pattern values, the translation is a procedure declaration

$\delta$(P = *P*) → procedure p(), suspend $\rho$(*P*), fail, end

or, more cosmetically,

```
procedure p()
   suspend ρ(P)
   fail
end
```

Thus SNOBOL4 patterns are translated into Icon procedures that encapsulate the corresponding matching expressions

### 4.3 Binding Times

Since patterns are constructed during program execution in SNOBOL4, but corresponding matching procedures are declared in Icon, there are problems related to binding times

In SNOBOL4, the unevaluated expression operator is used to defer evaluation of expressions until the time that pattern matching occurs, while all other values are bound at the time patterns are constructed. Since expressions in procedure declarations in Icon are not evaluated until the procedures are invoked, their binding time corresponds to unevaluated expressions in SNOBOL4. In Icon, however, procedures cannot be redefined during program execution. Consider, for example, the SNOBOL4 statements

```
N = 1
P = LEN(N)
Q = P "A"
R = *P "B"
P = LEN(*N)
N = N + 1
```

In the first assignment to P, N is bound with its value 1 and this pattern is itself bound in the assignment to Q. In R, however, P is not bound, nor is the value of N bound in the second assignment to P. On completion of executing this sequence of statements, the value of P is equivalent to LEN(2) and the value of R is equivalent to LEN(2) "B", while Q still has the value LEN(1) "A"

Since only straight-line programs are considered here, these problems can be handled by introducing auxiliary identifiers and assignment statements

```
N1 = 1
N = N1
P1 = LEN(N1)
P = P1
Q = P1 "A"
R = *P "B"
P2 = LEN(*N)
N2 = N1 + 1
N = N2
```

This process can be easily mechanized, but the primary concern here is pattern matching, not binding times. Since it is possible to transform straight-line programs to remove binding-time distinctions this matter will not be considered further here. This amounts to the assumption that all identifiers are unbound, and hence

$$\rho(*X) \rightarrow \rho(X)$$
$$\sigma(*X) \rightarrow \sigma(X)$$
$$\iota(*X) \rightarrow \iota(X)$$

In practice, even in SNOBOL4 programs with loops the binding time distinctions can usually be eliminated by making all identifiers in patterns unbound

### 4.4 The Transformation $\rho$

#### 4.4.1 Positional Patterns

Patterns that move the cursor to a specified position in the subject are called positional patterns. The simplest translation is for relative positioning:

$\rho$(LEN(*I*)) → move(*i*)

Because positions in strings are numbered from the left starting at 0 in SNOBOL4 but from 1 in Icon, an adjustment must be made for absolute positioning:

$\rho$(TAB(*I*)) → tab(*i*+1)

Since Icon allows nonpositive specifications for positions relative to right end of a string, RTAB(*I*) is translated as follows:

$\rho$(RTAB(*I*)) → tab(−*i*)

Of course

$\rho$(REM) → tab(0)

The predicates POS(*I*) and RPOS(*I*) have the translations

$\rho$(POS(*I*)) → pos(*i*+1) & move(0)
$\rho$(RPOS(*I*)) → pos(−*i*) & move(0)

The conjunction with move(0) is a device used to discard the value that is returned by pos(i) in Icon.

#### 4.4.2 Lexical Patterns

Patterns that are concerned with sets of characters in the subject are called lexical patterns. As with positional patterns, there are close correspondences between SNOBOL4 and Icon:

$\rho$(ANY(*S*)) → tab(any(*s*))
$\rho$(SPAN(*S*)) → tab(many(*s*))

Since Icon has an operation to form character set complements, NOTANY(*S*) is translated as follows:

$\rho$(NOTANY(*S*)) → tab(any(~*s*))

For example

$\rho$(NOTANY("aeiou")) → tab(any(~"aeiou"))

The SPITBOL [2] pattern BREAKX(*S*) also has an easy translation:

$\rho$(BREAKX(*S*)) → tab(upto(*s*))

The standard SNOBOL4 pattern BREAK(*S*) introduces a subtlety, since upto is a generator, but BREAK, unlike BREAKX, does not advance the cursor beyond the first instance of a character in *S*. Thus, control backtracking must be inhibited to prevent generation of alternatives by upto. This can be done by using braces:

$\rho$(BREAK(*S*)) → tab({upto(*s*)})

#### 4.4.3 Combining Patterns

Alternation and concatenation in SNOBOL4 combine patterns and represent the control regimes "or" and "then", together with their associated backtracking. The translation of these two SNOBOL4 operators into Icon illustrates the relationship between the SNOBOL4 pattern matching algorithm and the more general goal-directed evaluation mechanism of Icon.

Alternation has the translation

$\rho(P_1 \mid P_2) \rightarrow \rho(P_1) \mid \rho(P_2)$

For example

$\rho$(LEN(3) | RTAB(0)) → move(3) | tab(0)

It is interesting to note that alternation is a pattern-valued operator in SNOBOL4, while alternation is a control structure in Icon [10]

The translation of pattern concatenation is similarly straightforward

$\rho(P_1 \quad P_2) \rightarrow \rho(P_1) \parallel \rho(P_2)$

For example

$\rho$(SPAN(" ")  LEN(3)) → tab(many(" ")) || move(3)

Note that conjunction cannot be used in place of concatenation  Although the expression

tab(many(" ")) & move(3)

sets &pos to the same value as the concatenation above, it only returns the substring matched by move(3), not the substring matched by both expressions

The operands of alternation and concatenation may be strings  Their translation is

$\rho(S) \rightarrow =\sigma(S)$

That is,

$\rho(S) \rightarrow =s$

Since SNOBOL4 and Icon both have automatic type coercion of strings and integers,

$\rho(I) \rightarrow =i$

Identifiers present an interesting problem, since generally it is not possible to determine the type of the value of an identifier even in straight-line SNOBOL4 program by static analysis (because of value assignments that may be done during pattern matching)  The translation of a SNOBOL4 identifier that appears in alternation or concatenation depends, however, on its type  It is assumed here that the type of the value of an identifier can be determined by some means, either statically or through some auxiliary information (such as enforcing naming conventions), since dynamic analysis of SNOBOL4 programs is beyond the scope of this paper  The translations are

$\rho$(S) → =s
$\rho$(I) → =i
$\rho$(P) → p()

Here s and i have global scope in Icon (issues of scope in SNOBOL4 defined functions are not treated here)

Some unevaluated expressions that are used to defer computation in patterns involve SNOBOL4 predicates  The translation of such expressions must assure that the result is a matching expression  For example,

$\rho$(*GT($I_1$,$I_2$)) → ($i_1$ > $i_2$) & move(0)

where move(0) discards the value of the comparison (see Section 4 4 1)

### 4.4.4 Patterns with Alternatives

Patterns produced by alternation may match in more than one way  The SPITBOL function BREAKX also produces patterns with alternatives  Other patterns with this property are discussed in this section

ARBNO(P) matches an arbitrary (zero or more) occurrences of what P matches  Its translation is

$\rho$(ARBNO(P)) → arbno(p)

Where the procedure arbno is

```
procedure arbno(p)
   suspend move(0) | (p() || arbno(p))
   fail
end
```

This procedure first matches the null string. If reactivated, it matches what p matches, concatenated with an arbitrary number of things that p matches. Thus the sequence of values generated by arbno(p) is "", p(), p() || p(), p() || p() || p(), . . .

ARBNO differs from other SNOBOL4 pattern-valued functions in that its argument is a pattern. If its argument is a pattern-valued identifier, then the translation given above can be used directly. If, however, the argument of ARBNO is a pattern-valued expression (such as LEN(1)), then ARBNO(*P*) can be converted to ARBNO(Ṕ) by adding an assignment to an auxiliary identifier Ṕ:

```
Ṕ = P
```

The actual name of this auxiliary identifier must, of course, not conflict with the name of any other identifier. For example,

```
ρ(ARBNO(LEN(3) | ANY(" ,")) → arbno(ṕ)
```

given the procedure declaration

```
procedure ṕ()
   suspend move(3) | tab(any(" ,"))
   fail
end
```

The built-in pattern ARB, which is equivalent to ARBNO(LEN(1)), can be translated several ways. One is

```
ρ(ARB) → tab(&pos to size(&subject)+1)
```

Note that the following translation is *incorrect*:

```
ρ(ARB) → tab(&pos to 0)
```

Although the end of &subject can be referred to by the nonpositive specification 0, the generator to treats 0 in its usual arithmetic interpretation.

BAL presents a greater problem since there is no Icon scanning function that corresponds closely to it. bal is essentially a constrained form of upto, and despite the similarity of names, bal does not provide an easy way of implementing BAL. One method is to use procedures

```
ρ(BAL) → bbal() || arbno(bbal)
```

where the procedure bbal embodies the essential elements of BAL:

```
procedure bbal()
   suspend (="(" || arbno(bbal) || =")") | tab(any(~"()"))
   fail
end
```

The first alternative in the suspend corresponds to matching a balanced string enclosed in parentheses. The second alternative corresponds to matching any single character that is not a parenthesis. The translation of BAL into

```
bbal() || arbno(bbal)
```

ensures that at least one balanced substring is matched (an "idiosyncrasy" of BAL).

### 4.4.5 Value Assignment

Immediate value assignment in SNOBOL4 has the translation

```
ρ(P $ S) → s  =  ρ(P)
```

Note that S, by context, is necessarily a string-valued identifier

Conditional value assignment in SNOBOL4 can only be approximated in Icon, unless a very complicated superstructure is added  In most uses, the following translation is adequate

```
ρ(P   S) → s <- ρ(P)
```

This translation preserves the "conditional" aspect, in the sense that the value of s is restored if pattern matching fails, but it does not preserve the "deferred" aspect (another idiosyncrasy of SNOBOL4), since the assignment is made during pattern matching rather than after it is complete

The translation of cursor position assignment is

```
ρ(@S) → (s  = &pos-1) & move(0)
```

### 4.4.6 Control Patterns

SNOBOL4 has a number of built-in patterns that are designed to "control" pattern matching  The translations of FAIL and SUCCEED are

```
ρ(FAIL) → sfail()
ρ(SUCCEED) → succeed()
```

with the procedures

```
procedure sfail()
   fail
end

procedure succeed()
   repeat suspend move(0)
end
```

Approximations to ABORT and FENCE are given by

```
ρ(FENCE) → move(0) | fail
ρ(ABORT) → fail
```

These translations are only approximations, since in nested procedure calls they only terminate matching to one level and do not cause termination of the entire matching process  For example, the statements

```
P1 = "*" ABORT | LEN(1)
P2 = P1 | RTAB(1)
```

produce the declarations

```
procedure p1()
   suspend ="*" fail | move(1)
   fail
end

procedure p2()
   suspend p1() | tab(-1)
   fail
end
```

Thus p1() used alone fails to match if a * is encountered, but does not prevent p2() from matching its alternative  It should be noted that this "one-level" failure is nonetheless useful

### 4.5 The Transformations $\sigma$ and $\iota$

The transformation $\sigma$ is not a primary concern here and is relatively straightforward, since Icon includes all the string-valued functions of SNOBOL4, albeit with different names Thus

$\sigma$(TRIM($S$)) $\rightarrow$ trim($s$)
$\sigma$(DUPL($S$,$I$)) $\rightarrow$ repl($s$,$i$)

and so forth Of course, concatenation requires a syntactic change

$\sigma(S_1 \quad S_2) \rightarrow \sigma(S_1) \parallel \sigma(S_2)$

The transformation $\iota$ is similarly straightforward

Since SNOBOL4 and Icon have equivalent facilities for the automatic type conversion in context, it is sufficient that

$\sigma(I) \rightarrow \iota(I)$
$\iota(S) \rightarrow \sigma(S)$

### 4.6 Pattern-Matching and Replacement Statements

It remains to produce translations for SNOBOL4 pattern-matching and replacement statements Here Icon scanning procedures are needed The translations are

$\delta$($S$ ? $P$) $\rightarrow$ smatch($s$,$\acute{\text{p}}$)
$\delta$(S ? $P$ = $S$) $\rightarrow$ s = srepl(s,$\acute{\text{p}}$,$s$)

Where $\acute{\text{p}}$ is an auxiliary identifier for a matching procedure that encapsulates $\rho(P)$ as was done for ARBNO Of course, if $P$ is a procedure-valued identifier (P), this is unnecessary

The procedure smatch is

```
procedure smatch(s,p)
   return scan s using p()
end
```

Note that p is invoked within smatch If p() succeeds, the value it matches is returned, while if p() fails smatch(s,p) fails Replacement is slightly more complicated

```
procedure srepl(s1,p,s2)
   if transform s1 using p() & (tab(1) = s2)
      then return s1 else fail
end
```

The modification of &subject is effected by returning a scanned substring, tab(1), to which assignment is made in the transform-using expression in srepl Note that tab(1) moves &pos from the the value assigned to it by p() back to the beginning of &subject

### 4.7 Anchored Pattern Matching

The smatch and srepl procedures given above correspond to anchored pattern matching in SNOBOL4, in which the match must occur at the beginning of the string Anchoring may be controlled by

$\delta$(&ANCHOR = $I$) $\rightarrow$ anchor($i$)

where the procedure anchor is

```
procedure anchor(i)
   if i = 0 then init = arb else init = mnull
end
```

Thus the value assigned to init is either arb or mnull The procedure arb is an encapsulation of the translation for ARB given earlier

```
procedure arb()
   suspend tab(&pos to size(&subject)+1)
   fail
end
```

The procedure mnull is a procedure that simply matches the null string

```
procedure mnull()
   suspend move(0)
end
```

The procedures smatch and srepl then become

```
procedure smatch(s,p)
   return scan s using init() & p()
end
```

and

```
procedure srepl(s1,p,s2)
   local i
   if transform s1 using init() & (i := &pos) & p() & (tab(i) := s2)
      then return s1 else fail
end
```

Note that i is the value of &pos after the initial substring matched by init() so that s2 replaces only the substring matched by p(), between the values of &pos before and after p() is evaluated

### 4.8 Limitations

The preceding sections illustrate how most SNOBOL4 patterns can be translated into Icon matching expressions With the exception of conditional value assignment, FENCE, and FAIL, the translations are faithful to the semantics of SNOBOL4 The pattern-matching heuristics of SNOBOL4 are another matter, however

The heuristics used in pattern matching in SNOBOL4 in an attempt to increase the speed of pattern matching and to control left recursion have been the source of considerable controversy [9] The heuristics are poorly understood, they vary from implementation to implementation [2,5,13], and in fact are absent from MACRO SPITBOL [3]

The scanning procedures given above make no attempt to implement any pattern-matching heuristics, nor is it easy to modify them for such heuristics Since MACRO SPITBOL uses no heuristics without apparent problems for users, the inability to translate them into Icon does not seem to be a serious shortcoming

### 4.9 An Example

An example, taken from [8], illustrates the results of translating a segment of a typical SNOBOL4 program The SNOBOL4 code is

```
&ANCHOR = 1
NEXTBL = BREAK(" ") @L SPAN(" ") @M *NEXTBL
LOCBL = POS(0) @L @M NEXTBL | *GT(M,0)
TEXT = "Locate the last blank in this sentence"
TEXT ? LOCBL
```

The resulting Icon code is

```
global init
global text, l, m

procedure main()
   anchor(1)
   text = "Locate the last blank in this sentence"
   smatch(text,locbl)


end

procedure locbl()
   suspend (pos(1) & move(0)) || ((l = &pos-1) & move(0)) ||
      (m = &pos-1) & move(0)) || nextbl() | ((m > 0) & move(0))
   fail
end

procedure nextbl()
   suspend tab({upto(" ")}) || ((l = &pos-1) & move(0)) ||
      tab(many(" ")) || ((m = &pos-1) & move(0)) || nextbl()
   fail
end
```

**4.10 Encapsulating the Transformation $\rho$**

As the preceding example illustrates, the translation of complex patterns produces matching expressions that are lengthy and difficult to read This problem can be reduced by recasting the transformation $\rho$ to use procedures, where possible, instead of direct translation into Icon scanning functions For example, instead of the translation

$\rho$(BREAK($S$)) → tab({upto($s$)})

the transformation can be encapsulated in a procedure sbreak

$\rho$(BREAK($S$)) → sbreak($s$)

with the declaration

```
procedure sbreak(s)
   suspend tab({upto(s)})
   fail
end
```

It is also convenient to use mnull to return a null string rather than using conjunction with move(0) so that for example

$\rho$(@S) → mnull(s = &pos 1)

Matching expressions can be encapsulated for all patterns except alternation, value assignment, ABORT, and FENCE Alternation cannot be encapsulated in a procedure, since in Icon alternation is a control structure whose arguments are not evaluated as they are for functions [10] Encapsulation cannot be used for value assignment, since Icon has no call-by-reference facility ABORT and FENCE cannot be encapsulated, since they include fail The complete transformation $\rho$ and the corresponding Icon procedures are given in Appendices A and B

Using encapsulation, the matching procedures in example above become

```
procedure locbl()
   suspend spos(0) || mnull(l  = &pos-1) || mnull(m  = &pos-1) ||
      nextbl() | mnull(m > 0)
   fail
end

procedure nextbl()
   suspend sbreak(" ") || mnull(l  = &pos-1) || span(" ") ||
      mnull(m  = &pos-1) || nextbl()
   fail
end

procedure mnull()
   suspend move(0)
   fail
end

procedure sbreak(s)
   suspend tab({upto(s)})
   fail
end

procedure span(s)
   suspend tab(many(s))
   fail
end

procedure spos(i)
   suspend mnull(pos(i+1))
   fail
end
```

Concatenation can be encapsulated also, but the operator notation is left for clarity

## 5. Patterns in Icon

The translation of SNOBOL4 patterns into equivalent Icon matching procedures illustrates a paradigm for pattern matching in Icon This translation does not begin, however, to reveal the potential of Icon

As discussed in [11], one of SNOBOL4's serious shortcomings is its lack of a facility for programmer-defined matching procedures In SNOBOL4, patterns are limited to those that can be composed by combining the built-in ones In Icon, as manifested in the translations given above, such matching procedures are easy to write Furthermore, they have possibilities that are not touched on in the examples given so far Simply adding arguments leads to natural generalizations For example,

```
procedure arb(i)
   suspend tab(&pos to size(&subject)+1 by i)
   fail
end
```

is a version of arb that matches in increments of i rather than 1 Similarly,

```
procedure try(i)
   every 1 to i do suspend move(0)
   fail
end
```

is a constrained form of succeed Parameterizing bbal with the characters that are used to determine balancing is another easy modification

A major facility in Icon matching procedures that is not available in SNOBOL4 is the possibility for synthesis that is concomitant with analysis A limited form of this facility is built into Version 3 of Icon [1] in the form of insert(s), which inserts s into &subject at &pos It can be coded as a source-language matching procedure as

```
procedure insert(s)
   suspend move(0) := s
   fail
end
```

A similar procedure to delete i characters starting at &pos is

```
procedure delete(i)
   suspend move(i) := move(0)
   fail
end
```

These procedures are not matching procedures, since they change the value of &subject However, it is natural to define a more general *transforming* expression, in which the value of &subject is restored if the procedure fails Since assignment to scanned substrings is a reversible effect [1], these procedures meet that criterion

There are, of course, a host of other possibilities Since such procedures are written at the source-language level, it is easy to experiment

## 6. Conclusions

The concept of matching expression allows SNOBOL4-style pattern matching to be carried out in Icon The translation of SNOBOL4 patterns into Icon matching procedures not only illustrates the capabilities of Icon, but also provides an explication of pattern matching in SNOBOL4 That is, given an understanding of Icon, the translation provides an unambiguous description of SNOBOL4 patterns and its pattern-matching process More significantly, it provides a basis for describing and implementing other pattern-matching languages

Thus the translation of SNOBOL4 patterns into Icon matching expressions is a special case of a much more general model As illustrated above, matching expressions are a special case of transforming expressions For transforming expressions, it is but a step to a language with pattern-directed transformations

Other possible generalizations include operations on a subject that is not a string, but perhaps a structure While the design of primitive operations for matching and transforming structures is a challenge, the coding protocols and procedural paradigms for strings should apply to structures as well

**Acknowledgement**

I am indebted to Cary Coutant, Dave Hanson, and Steve Wampler for a number of illuminating discussions about string scanning and pattern matching

## References

1 Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language; Version 3*. Technical Report TR 80-2, Department of Computer Science, The University of Arizona. May 1980.

2 Dewar, Robert B. K. *SPITBOL Version 2.0*. Technical Report S4D23, Illinois Institute of Technology. February 1971.

3 Dewar, Robert B. K. and Anthony P. McCann. "MACRO SPITBOL — A SNOBOL4 Compiler", *Software — Practice and Experience*, Vol. 7 (1977). pp. 95-113.

4 Druseikis, Frederick C. and John N. Doyle. "A Procedural Approach to Pattern Matching in SNOBOL4", *Proceedings of the ACM Annual Conference*, 1974. pp. 311-317.

5 Gimpel, James F. "A Theory of Discrete Patterns and Their Implementation in SNOBOL4", *Communications of the ACM*, Vol. 16, No. 2 (February 1973). pp. 91-100.

6 Gimpel, James F. *SITBOL Version 3.0*. Technical Report S4D30b, Bell Telephone Laboratories. Holmdel, New Jersey. June 1973.

7 Gimpel, James F. "Nonlinear Pattern Theory", *Acta Informatica*, Vol. 4 (1975). pp. 213-229.

8 Griswold, Ralph E. *String and List Processing in SNOBOL4, Techniques and Applications*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1975. p. 14.

9 Griswold, Ralph E. "A History of the SNOBOL Programming Languages", *SIGPLAN Notices*, Vol. 13, No. 8 (August 1978). pp. 275-308.

10 Griswold, Ralph E. *Expression Evaluation in Icon*. Technical Report TR 80-21, Department of Computer Science, The University of Arizona. August 1980.

11 Griswold, Ralph E. and David R. Hanson. "An Alternative to the Use of Patterns in String Processing", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2 (April 1980), pp. 153-172.

12 Griswold, Ralph E., David R. Hanson, and John T. Korb. "The Icon Programming Language, An Overview", *SIGPLAN Notices*, Vol. 14, No. 4 (April 1979). pp. 18-31.

13 Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language*. Second Edition. Prentice-Hall, Inc. Englewood Cliffs, New Jersey. 1971.

14 Tennent, R. D. "Mathematical Semantics of SNOBOL4", *Proceedings of the ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, October 1973. pp. 95-107.

## Appendix A — The Transformation $\rho$

The details of the transformation $\rho$ for translating SNOBOL4 patterns into Icon matching procedures are given below To the extent possible, the matching expressions are encapsulated in procedures See Appendix B for the procedure declarations

*identifiers*

$\rho$(S) → mstring(s)  
$\rho$(I) → mstring(i)  
$\rho$(P) → p()

*string- and integer-valued expressions*

$\rho(S)$ → mstring($s$)  
$\rho(I)$ → mstring($i$)

*pattern-valued functions*

$\rho$(ARBNO($P$)) → arbno($\acute{p}$)  
$\rho$(ANY($S$)) → sany($s$)  
$\rho$(BREAK($S$)) → sbreak($s$)  
$\rho$(BREAKX($S$)) → breakx($s$)  
$\rho$(LEN($I$)) → len($i$)  
$\rho$(NOTANY($S$)) → notany($s$)  
$\rho$(POS($I$)) → spos($i$)  
$\rho$(RPOS($I$)) → rpos($i$)  
$\rho$(RTAB($I$)) → rtab($i$)  
$\rho$(SPAN($S$)) → span($s$)  
$\rho$(TAB($I$)) → stab($i$)

*pattern-combining operators*

$\rho(P_1 \mid P_2) \rightarrow \rho(P_1) \mid \rho(P_2)$  
$\rho(P_1 \quad P_2) \rightarrow$ cat$(\rho(P_1),\rho(P_2))$

*built-in patterns*

$\rho$(ARB) → arb()  
$\rho$(ABORT) → fail  
$\rho$(BAL) → sbal()  
$\rho$(FAIL) → sfail()  
$\rho$(FENCE) → move(0) | fail  
$\rho$(REM) → rtab(0)  
$\rho$(SUCCEED) → succeed()

*value assignment*

$\rho(P$ \$ S) → s = $\rho(P)$  
$\rho(P$  S) → s <- $\rho(P)$  
$\rho$(@S) → mnull(s = &pos-1)

*unevaluated expressions*

$\rho(*S) \rightarrow \text{mstring}(s)$
$\rho(*I) \rightarrow \text{mstring}(i)$
$\rho(*P) \rightarrow \text{p}()$

## Appendix B — Icon Matching Procedures for SNOBOL4 Patterns

```
procedure arb()
   suspend tab(&pos to size(&subject)+1)
   fail
end

procedure arbno(p)
   suspend move(0) | (p() || arbno(p))
   fail
end

procedure bbal()
   suspend (="(" || arbno(bbal) || =")") | notany("()")
   fail
end

procedure breakx(s)
   suspend tab(upto(s))
   fail
end

procedure cat(p1,p2)
   suspend p1() || p2()
   fail
end

procedure len(i)
   suspend move(i)
   fail
end

procedure mnull()
   suspend move(0)
   fail
end

procedure mstring(s)
   suspend =s
   fail
end

procedure notany(s)
   suspend sany(~s)
   fail
end

procedure rpos(i)
   suspend mnull(pos(-i))
   fail
end
```

```
procedure rtab(i)
   suspend tab(-i)
   fail
end

procedure sany(s)
   suspend tab(any(s))
   fail
end

procedure sbal()
   suspend bbal() || arbno(bbal)
   fail
end

procedure sbreak(s)
   suspend tab({upto(s)})
   fail
end

procedure sfail()
   fail
end

procedure span(s)
   suspend tab(many(s))
   fail
end

procedure spos(i)
   suspend mnull(pos(i+1))
   fail
end

procedure stab(i)
   suspend tab(i+1)
   fail
end

procedure succeed()
   repeat suspend move(0)
end
```