An Alternative to the Concept of ''Pattern'' in String Processing

Ralph E. Griswold

TR 78-4

April 10, 1978

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

An Alternative to the Concept of "Pattern" in String Processing

My life is made of Patterns
That can scarcely be controlled.
--- Paul Simon

1. Introduction

SNOBOL4 is certainly best known for its pattern-matching facilities [1]. Among readily available high-level languages, it is virtually unique in providing powerful facilities for string analysis. Proposals have been made for extending the pattern-matching facilities of SNOBOL4 to include synthesis as well as analysis [2], and procedural mechanisms for implementing patterns are a central issue in SL5 [3-6] and subsequent work [7]. A number of other languages have incorporated patterns in a style similar to SNOBOL4 or have proposed such facilities in language variants [8-9]. AI languages in particular have increasingly included patterns and pattern matching as central facilities [10-11].

Considering the importance attributed to patterns, it is worthwhile to make a critical evaluation of their characteristics, their advantageous and disadvantageous attributes, and the degree to which they are essential as a mechanism for embodying search and backtrack facilities. This paper considers SNOBOL4 in particular and suggests an alternative to patterns that provides most of their advantages without the associated disadvantages.

2. Patterns in SNOBOL4

In SNOBOL4, patterns are data objects constructed during program execution. There is a repertoire of pattern construction functions and operators that provide for a variety of patterns and relationships among them. During pattern matching, a focus of attention (cursor position) is maintained in the string being examined (the subject). As pattern components successfully match, the cursor is advanced and subsequent pattern components are applied. If a pattern component fails, alternative components are applied. If no alternative succeeds, backtracking to an earlier state is attempted to seek alternatives to a formerly successful match. For descriptions of the matching process, see References 1, 12, 13, and 14.

2.1 Advantages of the Pattern Approach

The richness of the SNOBOL4 pattern facility is illustrated by the following list of pattern-constructing operations and the corresponding processes that occur during pattern matching.

LEN(N)	match N characters
POS(N)	match if cursor is at position N
RPOS(N)	match if cursor is at position N from right end
RTAB(N)	move cursor to position N from right end

TAB(N)	move cursor to position N
ANY(S)	match any character in S
BREAK(S)	match string up to any character in S
NOTANY(S)	match any character not in S
SPAN(S)	match string through characters in S
P $ V	assign substring matched by P to V
P . V	assign substring matched by P to V if entire match succeeds
@V	assign cursor value to V
*X	evaluate X during pattern matching
P1 | P2	apply P1 or apply P2
P1 P2	apply P1 then apply P2
ARBNO(P)	apply pattern P an arbitrary number of times

In addition, there are seven built-in patterns and a number of modes of matching under programmer control. In all, pattern matching in SNOBOL4 provides a powerful facility for string analysis.

Patterns also provide an abstraction mechanism. For example

```
        PUNCT = ANY(",.;:!?")
```

allows PUNCT to be used as an abstraction for the punctuation characters.

A large part of the usefulness of pattern matching lies in the automatic bookkeeping that is provided. A focus of attention in the subject is maintained as matching progresses without the need for explicit specification by the user. While the value of this automatic bookkeeping may appear to be minor, it has the practical effect of freeing the programmer from one of the most error-prone aspects of programming, complex nested indexing. An important consequence of automatic bookkeeping lies in the suppression of notational detail. Since each pattern match applies only to a single subject and the cursor changes automatically, neither of these variables has to be specified in the pattern. Thus complex operations can be expressed with considerable conciseness.

One of the special aspects of patterns lies in their ability to characterize properties of strings in a manner similar to the way that production grammars characterize context-free languages. Patterns viewed in this way provide an easy method for emulating static grammatical characterizations and, for example, constructing recognizers without the need to know how the recognition process is carried out. This is possible because the process of pattern matching, i.e. the application of a pattern to a string, embodies a powerful search and backtrack algorithm that the programmer need not thoroughly understand, much less implement. The algorithm includes the maintenance of state information and the reversal of effects during backtracking.

An advantage of treating patterns as data objects is that complex patterns can be composed from simpler ones using construction operators that parallel the grammatical concepts of subsequent and alternate. Thus recognizers for complex grammars can be built in a bottom-up fashion, starting with simple components and fashioning more complex ones. The almost direct mapping between productions of a grammar and corresponding SNOBOL4 patterns is particularly appealing. A simple example is given by the grammar

```
        <var>::=x|y|z
        <addop>::=+|-
        <mulop>::=*|/
        <term>::=<var>|(<exp>)|<term><mulop><var>
        <exp>::=<term>|<exp><addop><term>
```

for which the corresponding SNOBOL4 patterns are

```
        VAR = "x" | "y" | "z"
        ADDOP = "+" | "-"
        MULOP = "*" | "/"
        TERM = VAR | "(" *EXP ")" | *TERM MULOP VAR
        EXP = TERM | *EXP ADDOP TERM
```

Note the use of deferred evaluation to handle the forward ("recursive") references to TERM and EXP. Since a pattern is a data object, the effect of a loop is obtained by deferring reference to these components until after the pattern is constructed. The self-references are constructed as a result of evaluating TERM and EXP during pattern matching.

In fact, a direct translation between production grammars and patterns can be made by deferring evaluation of all patterns [15]. Using this device for the example above, the patterns are:

```
        VAR = "x" | "y" | "z"
        ADDOP = "+" | "-"
        MULOP = "*" | "/"
        TERM = *VAR | "(" *EXP ")" | *TERM *MULOP *VAR
        EXP = *TERM | *EXP *ADDOP *TERM
```

It is interesting to note that deferred evaluation saves space by avoiding the copying of patterns during construction at the expense of the time required to reference them during pattern matching.

Patterns can also be constructed in a top-down fashion, although this technique is less frequently used. For the example above, this amounts to reversing the order of construction and the use of deferred evaluation for "forward references":

```
        EXP = *TERM | *EXP *ADDOP *TERM
        TERM = *VAR | "(" EXP ")" | *TERM *MULOP *VAR
        MULOP = "*" | "/"
        ADDOP = "+" | "-"
        VAR = "x" | "y" | "z"
```

SNOBOL4 allows greater expressive power than most production grammar systems, of course. Thus

```
        VAR = ANY("xyz")
        ADDOP = ANY("+-")
        MULOP = ANY("*/")
```

are both more concise and more efficient than the alternation of individual characters.

2.2 Disadvantages of the Pattern Approach

The problems with patterns are closely related to their virtues. While the pattern-matching facility of SNOBOL4 has a richness of expressive power, it also has a corresponding verbosity. The large vocabulary of pattern-construction operations, built-in patterns, and matching modes presents the programmer with a formidable repertoire to master.

Similarly, while the implicit pattern-matching algorithm is helpful in formulating complex string analysis, its hidden intricacies may baffle the programmer trying to find the source of a bug. In circumstances where knowledge of the details of pattern matching is necessary, the programmer must master an arcane discipline. Some aspects of pattern matching are so obscure that even the designers and implementors of the language are forced to resort to listings of the system for answers (for example, if a pattern contains a component of the form P . *V, when is *V evaluated and what happens if its evaluation results in failure?).

Less obvious to the programmer is the unnecessary processing that may result because of the exhaustive search-and-backtrack algorithm. While the programmer benefits from the built-in algorithm, the lack of control over this algorithm may result in hidden but substantial inefficiencies in processing. This issue has, of course, been of considerable concern in AI languages [16].

One of the most difficult concepts for the beginning SNOBOL4 programmer to grasp is that pattern construction and pattern matching are separate and distinct processes. Furthermore, since patterns can be constructed at their site of use, the existence of the two processes is not always evident. For example, in

```
LOOP    LIST   BREAK(",") . K LEN(1)    =         :F(DONE)
```

the two processes are not apparent, although both occur. However, in

```
        ITEM = BREAK(",") . K LEN(1)
                  .
                  .
                  .
LOOP    LIST   ITEM   =                           :F(DONE)
```

the first statement clearly constructs a pattern, while the last statement just as clearly applies this pattern. The sophisticated SNOBOL4 programmer knows that the second approach is more efficient in most implementations of SNOBOL4, since the pattern is constructed only once, while the first approach requires that the pattern be constructed for each execution of the statement labeled LOOP. It should be noted that pattern construction uses two resources -- time and space. In the first approach above, time and space are used for each construction of the pattern. After the execution of this statement, this pattern is no longer accessible. Most SNOBOL4 systems eventually "garbage collect" such transient objects to reclaim the space, but since this takes time as well, creation of transient objects eventually imposes an additional time penalty. (It should be noted that some implementations of SNOBOL4 treat constant in-line patterns separately, placing them out of the line of actual program execution.)

From the point of view of program structure, an in-line pattern provides evidence of its function at the site of use, whereas an out-of-line pattern, being physically separated from its site of use, must be located to determine its actual function. Well-chosen mnemonics help, but can hardly substitute for the pattern itself. This tends to defeat the use of patterns as an abstraction mechanism. Furthermore, patterns, unlike functions, cannot be given arguments. This frequently results in the use of a number of similar, but distinct patterns. Again, unlike functions, patterns have no local identifiers and hence must operate by side effects on global variables, as illustrated in the example above. If a pattern is not constructed at its site of use, the difficulty with side effects is aggravated.

One of the most serious linguistic problems with pattern matching in SNOBOL4 is the fact that the pattern-matching facility constitutes an essentially distinct sublanguage imbedded in SNOBOL4. The kinds of operations that occur during pattern matching are significantly different from those that occur outside pattern matching. While there are patterns such as ANY(S) that have no counterpart outside of pattern matching, there are similar, but significantly different, parallels inside and outside of pattern matching. Thus, while SNOBOL4 has a standard assignment operation, pattern matching has three forms of assignment (P $ V, P . V, and @V). Similarly expressions are executed sequentially outside of pattern matching, while inside pattern matching the sequence P1 P2 results in sequential application of P1 and P2, but with search for alternatives and backtracking.

In a very real sense, SNOBOL4 is composed of two languages, a basic language, L, and a pattern-matching language, P. This linguistic dichotomy produces a total vocabulary that is large, forces the programmer to think differently in the two languages, to use different approaches and phraseology, to decide which language to use to accomplish a particular task, and to change frames of reference frequently. The effect is a "linguistic schism".

The dichotomy is particularly troublesome because there is little facility for communication between L and P. In L, patterns for P are constructed. When a pattern match occurs in L, control is transferred to P, where the matching procedures for the pattern are then executed. Thus L has the operations necessary for describing programs in P (but not for carrying out their actions). Pattern construction is essentially the compilation of such programs for P. In typical SNOBOL4 programs, programs for P are continually compiled and executed. Note that the vocabulary of L is increased by having to describe programs in P and that compilation of programs for P during the execution of L is an inherently expensive process.

Pattern matching is not extensible in the same fashion that the rest of the language is. While SNOBOL4 has a facility for programmer-defined functions and datatypes in L, there is no facility for programmer-defined matching procedures, i.e. procedures in P. While complex patterns can be composed from simpler ones, there is no mechanism for introducing new methods of matching.

In P, operations of L are inaccessible except through the interface of unevaluated expressions. This interface is awkward at best. Consider, for example, the problem of determining whether the first comma in a string is at least K characters from the beginning. Numerical computation is part of L, but not P. On the other hand, L has no facilities for locating characters in strings. There are several possible approaches to this problem (the existence of such alternatives is, in itself, indicative of a difficulty). If this problem is given to a typical SNOBOL4 programmer, the most likely type of solution is:

```
        S   BREAK(",") . T                      :F(NO)
        GE(SIZE(T),K)                           :S(YES)F(NO)
```

Here, the solution is divided into two parts. One part is performed in P to get the substring up to the first comma. The second part is performed in L to test the length of this substring.

The more sophisticated (or involutionally minded) SNOBOL4 programmer might produce the following solution:

```
        S   BREAK(",") $ T *GE(SIZE(T),K)       :F(NO)S(YES)
```

Here the solution is accomplished in one statement (a doubtful virtue) by having P interface L through an unevaluated expression to perform the necessary numerical computation. A better solution along these lines is

```
        S   BREAK(",") @N *GE(N,K)              :F(NO)S(YES)
```

The advantage of this solution is that the formation of the substring T is avoided. However, all of these solutions have evident problems. Each of them requires assignment to a global variable as a side effect in P in order to have the information necessary to do a simple computation in L.

The real problem here is that there are frequently times when both *L* and *P* are inadequate, individually. In such cases, the typical result is obscure, refractory, and poorly structured.

2.3 Patterns in Perspective

To summarize the preceding sections, patterns have number of valuable aspects:

1. Powerful facilities for string analysis.

2. An abstraction mechanism.

3. Automatic bookkeeping.

4. A built-in search and backtrack algorithm.

5. Natural characterization of languages.

On the other hand, patterns present many problems:

1. An excessively large vocabulary.

2. Complexity of the pattern-matching algorithm.

3. Unnecessary backtracking and lack of control over the pattern-matching algorithm.

4. Confusion between pattern construction and pattern matching.

5. Difficulties with program structuring.

6. Dichotomy of languages, with a further increase in total vocabulary and a linguistic schism.

7. Inherent inefficiency of runtime construction of patterns.

8. Lack of mechanism for defining matching procedures.

A number of attempts have been made to solve these problems by extending *P*. Suggestions have been made for adding string synthesis facilities [2], for adding programmer-defined matching procedures [17], and for providing more control over the matching algorithm [2]. These proposals provide much of the basis for SL5 [3-6]. Expanding the *P* component has hardly eliminated the need for the *L* component. In fact, the *L* component of SL5 is larger. It includes, among other things, functions for performing simple string analysis in cases where complex search and backtracking are not needed. The dichotomy in SL5 is increased, not reduced, and the vocabulary is, of course, also increased. The linguistic schism is just as deep in SL5 as it is in SNOBOL4.

The fundamental question is whether such a dichotomy is necessary. It is the thesis of this paper that most of the virtues of pattern matching in SNOBOL4 and related languages can be retained in a language without such a dichotomy and, in fact, without patterns.

3. A New Approach to String Processing

The new approach is to augment the more traditional L component and eliminate P. The major additions to the L component necessary to achieve the advantages of pattern matching without actually having patterns are a facility for automatic bookkeeping and search and backtrack mechanisms. The following sections describe the major features of this approach.

3.1 A Brief Overview

The programming language that contains this new approach to string processing is called Icon [18]. Icon resembles SL5 more than SNOBOL4. It has a reserved word syntax with traditional control structures as well as some novel ones. The evaluation of an expression in Icon produces a result consisting of a value and a signal as in SL5. The value portion of the result serves the traditional computational role. Success and failure signals drive control structures in a manner similar to SL5.

Icon lacks the P component of SL5, has a less general procedure mechanism than SL5, but adds new control structures and evaluation concepts that are described in subsequent sections.

An extensive description of Icon is beyond the scope of this paper and is not necessary for understanding the basic thesis. Examples taken from Icon should be clear by context, at least in their general aspects, if not in all details.

3.2 Automatic Bookkeeping

In Icon automatic bookkeeping is accomplished in a manner that appears to be similar to SNOBOL4 but simply bypasses the construction of patterns. The expression

```
s ? e
```

called "scanning", establishes a global subject, s, to which string processing operations in e apply. The expression e, which can include any operation, but typically includes string processing operations, is then evaluated. String processing operations that apply to the subject are called "scanning operations". The result returned by s ? e is the result returned by e.

A typical scanning operation is upto(c), which returns the position in the subject of the first occurrence of a character in c (note the similarity of this operation to the pattern BREAK(c) in SNOBOL4). Thus

```
        s ? (j := upto("aeiou"))
```

assigns to j the position of the first vowel in s (failing if there is no vowel).

This simple example illustrates several important points. As in SNOBOL4, the string operated on by upto(c) is implicit and does not have to be specified as an argument. Unlike SNOBOL4, upto(c) does not construct a pattern, but rather simply carries out the analysis. In SNOBOL4, BREAK(c) constructs a pattern, which, when applied, carries out the analysis. (Note that the precise action is different; upto(c) returns a position, while BREAK(c) returns the substring matched. This difference is inessential.) Another important point is that the expression e in s ? e can contain any Icon operation. In the example above, the standard form of Icon assignment is used to assign the desired position. In SNOBOL4 the equivalent statement would be

```
        s   BREAK("aeiou") @j
```

In Icon, the focus of attention in the subject is maintained as an implicit cursor, similar to the method used in the P component of SNOBOL4. When the subject is established, the cursor is set to 1 (Icon strings are indexed beginning at 1, not 0 as in SNOBOL4). Some Icon operations move the cursor. Examples are tab(n), which sets the cursor to n, and move(n), which adds n to the current value of the cursor. Again, there are analogies to the SNOBOL4 operations TAB(n) and LEN(n), although tab(n) and move(n) operate directly rather than constructing patterns. An example of the use of such a scanning operation is

```
        s ? write("[" || move(2) || "]")
```

which is equivalent to the SNOBOL4 statements

```
        s   LEN(2) . TWO
        OUTPUT = "[" TWO "]"
```

Note that the linguistic schism evidenced in the SNOBOL4 statements does not exist in Icon. The advantage of the Icon approach is particularly evident where more complicated control structures are useful. An example is

```
        while s := read() do
           s ? repeat write("[" || move(2) || "]")
```

The subject and cursor are directly accessible in Icon as keywords &subject and &cursor. Assigning a value to &subject establishes the subject for string scanning. &cursor is automatically set to 1 when &subject is set. &cursor can be explicitly set to any value in the range of &subject.

Since string processing expressions may be complicated and extensive in scope, it is frequently useful to set &subject explicitly, rather than using scanning expressions. The preceding example is written more concisely as

```
        while &subject := read() do
           repeat write("[" || move(2) || "]")
```

The advantage of a scanning expression is that the current subject and cursor are saved before e is evaluated and restored after e is evaluated. In fact, s ? e is essentially equivalent to

```
        push(&cursor)
        push(&subject)
        &subject := s
        e
        pop(&subject)
        pop(&cursor)
```

where push(x) and pop(x) represent internal stack operations for saving and restoring values. Thus nested string scanning is easily obtained. For example

```
        s ? e1 ? e2
```

applies e2 to the result returned by s ? e1.

3.3 String Scanning Operations

There are eight string scanning operations in Icon. Two, move(n) and tab(n), are positional. The remainder are "lexical" in the sense that they analyze the character structure of the subject.

In move(n), the value of n may be negative, specifying movement of the cursor toward the left end of the subject. In all cases, the value of move(n) is the substring between the previous and new cursor positions (regardless of the direction of cursor movement). The operation fails if the resulting cursor position would not be in the range of the subject.

In tab(n), a nonpositive argument specifies a position relative to the right end of the subject. Thus, tab(0) positions the cursor past the last character of the subject. As with move(n), the value of the operation is the substring between the previous and new cursor positions (regardless of the direction of movement). The operation fails if the resulting cursor position would not be in the range of the subject.

The value of &cursor is always positive. A nonpositive value can be assigned to &cursor to specify a position relative to the right end of the string without having to compute the length of the string. This device suppresses detail and avoids bothersome computation. Thus, if the subject is "portability",

```
        &cursor := 0
```

actually sets &cursor to 12, and subsequently

```
        j := &cursor
```

sets j to 12, not 0.

The lexical scanning operations in Icon are more extensive than those in the P component of SNOBOL4:

```
upto(c)
thru(c)
any(c)
bal(c1,c2,c3)
find(s)
match(s)
```

The scanning operation upto(c) returns the position of the first occurrence of a character of c in the subject, starting at the current cursor position. Thus, if the subject is "portability" and the cursor is 3, the value of upto("aeiou") is 5. The operation fails if no such character exists. Note that upto(c) does not change the cursor; the effect of BREAK(c) in SNOBOL4 is obtained by tab(upto(c)).

The scanning operation thru(c) returns the position after a continuous sequence of characters in c in the subject, starting at the current cursor position. Thus, if the subject is "moonshine" and the cursor is 2, the value of thru("aeiou") is 4. The operation fails if the character of the subject at the current cursor position is not contained in c.

The scanning operation any(c) succeeds if the character at the current cursor position in the subject is contained in c and fails otherwise. The value returned is one greater than the current cursor position. Character sets in Icon may be complemented with respect to the alphabet of all characters. Thus any(~c) succeeds if the character at the current cursor position is not included in c.

The scanning operation bal(c1,c2,c3) is a generalization of the matching procedure for the SNOBOL4 pattern BAL. In SNOBOL4, BAL only matches strings balanced with respect to parentheses. In Icon, c1 and c2 are sets of characters that specify the left and right balancing characters. Furthermore, c3 specifies a set of characters that may follow the balanced string. For example, if the subject is "(a)*[b]-7" and the cursor is 1, the value of bal("[(",")]","+-") is 8. The operation fails if there is not such a balanced string starting at the current cursor position. For convenience, defaults are used if the arguments are null:

c1	"("
c2	")"
c3	any character

If c3 is null, the balanced string may extend through the end of the subject. Thus bal() is equivalent to the matching procedure for BAL.

The scanning operation find(s) returns the position of the first occurrence of the string s in the subject, starting at the current cursor position. Thus, if the subject is "mississippi" and the cursor is 1, the value of find("is") is 2. The operation fails if no such string exists.

The scanning operation match(s) returns the cursor position after the occurrence of s as a substring of the subject starting at the current cursor position. Thus, if the subject is "mississippi" and the cursor is 2, the value of match("is") is 4. The operation fails if s is not a substring of the subject at the current cursor position. Thus, for the subject above, if the cursor were 1, match("is") would fail.

For convenience, =s is equivalent to tab(match(s)). Note that =s corresponds to the pattern component s in SNOBOL4.

3.4 Searching and Backtracking

One of the essential components of high-level string processing is the ability to express alternatives concisely and to have the search for such alternatives carried out automatically.

In Icon, the operation e1 | e2 is equivalent to the the operation performed in SNOBOL4 when the pattern P1 | P2 is evaluated in _P_.

This operation is actually fairly complex and deserves discussion. The most obvious aspect of alternation is that e1 is evaluated first and if that evaluation succeeds, the result is the result of the entire expression. However, if evaluation of e1 fails, e2 is evaluated and its result is the result of the entire expression. The subtlety arises if the value produced by successful evaluation of the alternation is not acceptable in the context in which it occurs. Consider, for example,

```
        tab(10 | 5)
```

(Note that this construction, while clear in its intent, has no direct counterpart in SNOBOL4). The expression 10 | 5 has two literal subexpressions, and of course the first, 10, succeeds. However if the subject is, say, six characters long, tab(10) fails. This results in a "re-evaluation" of the expression 10 | 5 and the alternative value, 5, is returned the second time. Thus, tab(10 | 5) is equivalent to tab(10) | tab(5) as would be expected.

In Icon, operations that have the capacity for producing alternative values as required by the context in which they appear are called _generators_. This capacity for generating alternate values is meaningful for many operations in Icon.

The scanning operation upto(c) is, in fact, a generator. For upto(c), the behaviour is like that for the matching procedure for BREAKX(c) in the SPITBOL dialect of SNOBOL4 [19]. If the value returned by upto(c) does not satisfy the context in which it is used, the next position further on is returned, and so on. Note that upto(c) is a generator with an indefinite number of alternatives that depend on c and the current subject.

The possible need for the second alternative in tab(10 | 5) is clear, but the need for alternatives in upto(c) is not so obvious (Note that tab(upto(c)) necessarily succeeds for any value of upto(c) and move(upto(c)) is somewhat fanciful). There are, however, other control structures that may require alternatives. One of these is e1 & e2, which succeeds only if both e1 and e2 succeed. In requiring this "mutual" success, there is automatic backtracking for alternatives of e1 if e2 fails. This corresponds to the matching procedure for the concatenation of patterns, P1 P2, in SNOBOL4. Suppose, for example, that the subject is "mississippi" and the cursor is 1. In the expression

```
tab(upto("i")) & ="issip"
```

upto("i") first returns the value 2 and tab(upto("i")) moves the cursor to this position. However, ="issip" fails, and the first expression is re-evalauted for an alternative. This time the value of upto("i") is 5, tab(upto("i")) move the cursor correspondingly, and ="issip" succeeds.

Note that tab(upto(c)) is equivalent to matching for the SPITBOL pattern BREAKX(c), so the expression above is equivalent to matching for the SPITBOL pattern

```
BREAKX("i") "issip"
```

The Icon expression is slightly more verbose than the SPITBOL pattern, but in turn, Icon expressions offer more flexibility (there is no easy SPITBOL equivalent to j := upto(c)). This trade-off is typical and works to the advantage of Icon in complex string processing, while the conciseness of SNOBOL4 is an advantage in simple situations.

The other string scanning operations that are generators are thru(c), bal(c1,c2,c3), and find(s). For bal(c1,c2,c3), the alternatives are as in SNOBOL4 BAL -- successively longer balanced strings. For find, the alternatives are positions of s successively further on in the subject. For thru(c), the converse is the case -- for each alternative, a value one less than the previous is returned, until the shortest possible continuous (nonnull) sequence of characters in c is located. For example, if the subject is "moonshine" and the cursor is 2, the expression

```
s := tab(thru("aeiou")) & ="on"
```

succeeds and assigns "o" to s, although "oo" is assigned for the initial evaluation of the expression.

The full range of search and backtracking in SNOBOL4 pattern matching is available in the Icon expressions e1 | e2 and e1 & e2. It is important to note that e1 & e2 does not have to be used unless it is needed (while in SNOBOL4 backtracking in a sequence of pattern components cannot be avoided). For example,

```
        x := tab(upto(c1)) & y := tab(upto(c2))
```

succeeds only if the subject contains a character of c2 in a position at or beyond a character of c1, while in the sequence of expressions

```
        x := tab(upto(c1)); y := tab(upto(c2))
```

this constraint does not apply. A value may be assigned to y even if the subject does not contain a character in c1.

When an instance of move(n), tab(n), or =s is backtracked over (that is when it fails for lack of alternatives), the effects of implicit cursor movement are reversed and the cursor is restored to its position prior to the evaluation of the operation. For example, if the subject is "portability" and the cursor is 1, evaluation of

```
        tab(10) & ="a"
```

first sets the cursor to 10 but then restores it to 1 when ="a" fails.

Other effects are not reversed. In the example above, if the expression were

```
        &cursor := 10 & ="a"
```

the value of the cursor would not be restored, since it was set by assignment, not by a scanning operation.

3.5 Procedures

One of the most severe limitations of pattern matching in SNOBOL4 is the inability to add new matching procedures. Since SNOBOL4 has no such facility, programmers do not miss it *per se* (it is essentially "inconceivable", since, as a language, SNOBOL4 has no construct for expressing such a possibility).

In Icon, procedures allow the construction of programmer-defined generators and hence programmer-defined scanning procedures.

Icon procedures resemble those of SL5 [6], although the SL5 decomposition of procedure activation (and hence coroutine usage) is not available in Icon.

A typical Icon procedure is

```
        procedure max(n,m)
           if m > n then return m else return n
        end
```

Since scanning operations are on a par with all other Icon operations, procedures may be used for scanning in the same way that they are used as abstractions for other purposes. An example is a procedure that behaves like match(s), but is "unanchored" like find(s):

```
procedure fmatch(s) private j
   if j := find(s) then return j + length(s) else fail
end
```

Defined generators are obtained by using suspend, which returns a value like return, but leaves the procedure in a state that it can be resumed for the generation of additional values.

For example, the procedure fmatch(s) defined above is not a generator like find(s). This defect can be remedied by using suspend:

```
procedure fmatch(s) private j
   every j := find(s) do suspend j + length(s)
   fail
end
```

A more esoteric application is in the use of defined generators to characterize languages in a fashion similar to SNOBOL4 patterns. Consider the simple grammar

```
<s>::=a<s>a|b
```

An Icon procedure to "match" sentences from the language generated by this grammar is

```
procedure s
   every ="a" & s() & ="a" | ="b" do suspend
   fail
end
```

This procedure is suspended for every alternative of the expression describing the language. Thus

```
"aabaa" ? s()
```

calls s. The first alternative matches "a" and calls s again (recursively), resulting in the match of the second "a" and another call to s. This time, the first alternative fails and the "b" is matched. Upon successive returns, a trailing "a" is matched each time and the entire expression succeeds. On the other hand, for a subject that is not a sentence in the language, alternatives are eventually exhausted, and the scanning operation fails.

The method used above generalizes for more complex grammars, with a procedure for each nonterminal symbol. The correspondence between production grammars and defined scanning procedures is just as direct as the correspondence between production grammars and SNOBOL4 patterns, if a bit more involved.

4. Conclusion

Not surprisingly, scanning in Icon presents some problems. One problem is the choice of primitive scanning operations. For example, it might be desirable to have a scanning operation that sets the cursor like tab(n) but does not return the substring between the previous and new cursor positions. The advantage of such an operation would be efficiency, since the computation of the substring would be avoided. A similar situation exists for move(n). If these new operations are added, however, the vocabulary of the language is increased, with all the attendant problems. An alternative is to replace tab(n) and move(n) by these new operations and add an additional operation to obtain substrings. At the other extreme, tab(upto(s)) is used so frequently that a single operation that combines these two would be useful.

The bases for such decisions are the usual ones in language design. The problem is aggravated by the relative unfamiliarity of scanning. The historical influence of SNOBOL4 tends to inhibit new views. More experience with scanning should provide insight.

The scope of &subject presents a rather serious problem. In Icon, scanning operations on the same subject tend to be more extensive than in SNOBOL4. This is the reason that setting &subject directly is frequently more useful than the implicit setting of the subject in s ? e. However, it then becomes more likely that the subject or cursor may be changed inadvertently. For example, if a defined procedure is called, it may expect to operate on the subject (as the procedure fmatch(s) given above) or it may establish its own subject. If it does the latter without saving and restoring the prior subject and cursor, the results may be catastrophic.

The generally recognized hazards of global variables are magnified here because of the frequency with which the two globals, &subject and &cursor, are used. This appears to be a dilemma, since much of the virtue of string scanning is derived from the globality of these variables.

The usefulness of string scanning in Icon leads to a number of possibilities and open questions. Once scanning on a single subject is available, situations immediately arise where the simultaneous scanning of two or more subjects would be useful. Again, this is a dilemma, since it is the single focus of attention that leads to the simplifications that make string scanning attractive. Any departure from this single focus of attention introduces complexity and detail that string scanning presently avoids.

Looking in another direction, there is no inherent reason why scanning should be limited to strings. Scanning of data structures, given appropriate primitives, follows by analogy. Such possibilities are particularly seductive.

Acknowledgement

I am indebted to Dave Hanson and Tim Korb for a number of suggestions and ideas that are incorporated in this work. I also owe them, as well as my wife Madge, thanks for critical readings of drafts of this report.

References

1. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. The SNOBOL4 Programming Language, 2nd ed. Prentice-Hall, Englewood Cliffs, New Jersey. 1971.

2. Doyle, John N. A Generalized Facility for the Analysis and Synthesis of Strings and a Procedure-Based Model of an Implementation. Masters Thesis, Department of Computer Science, The University of Arizona, Tucson, Arizona. 1975.

3. Griswold, Ralph E. and David R. Hanson. "An Overview of SL5", SIGPLAN Notices, Vol. 12, No. 4 (April 1977), 40-50.

4. Griswold, Ralph E. String Scanning in SL5. SL5 Project Document S5LD5a, The University of Arizona, Tucson, Arizona. June 17, 1976.

5. Griswold, Ralph E. "String Analysis and Synthesis in SL5", Proceedings of the ACM Annual Conference, October, 1976, pp. 410-414.

6. Hanson, David R. and Ralph E. Griswold, "The SL5 Procedure Mechanism", Communications of the ACM, to appear, May 1978.

7. Garnaat, M. J., et al. The Design and Implementation of New String Transformation Facilities for SL5. Technical Report TR 78-1, Department of Computer Science, The University of Arizona, Tucson, Arizona. January 13, 1978.

8. A. L. Furtado and A. S. Pfeffer. "Pattern Matching for Structured Programming", Proceedings of the Seventh Asilomar Conference on Circuits, Systems, and Computers. Pacific Grove, California. 1973.

9. L. G. Tesler, H. J. Enea, and D. C. Smith. "The LISP70 Pattern Matching System", Proceedings of the Third International Joint Conference on Artificial Intelligence, pp. 671-676. Stanford, California. 1973.

10. L. F. Melli. The 2.Pak Language Primitives for AI Applications. Masters Thesis, Department of Computer Science, University of Toronto. December, 1974.

11. "Proceedings of the Workshop on Pattern-Directed Inference Systems", SIGART Newsletter, No. 63 (June, 1977), pp. 1-84.

12. Griswold, Ralph E. The Macro Implementation of SNOBOL4, A Case Study in Machine-Independent Software Development. W. H. Freeman, San Francisco. 1972.

13. 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.

14. Gimpel, James F. "Nonlinear Pattern Theory", Acta Informatica, Vol. 4 (1975), pp. 91-100.

15. Griswold, Ralph E. String and List Processing in SNOBOL4, Techniques and Applications. Prentice-Hall, Inc. Englewood Cliffs, N.J. 1975. pp. 12, 233-234.

16. Sussman, G. J. and D. V. McDermott. "From PLANNER to CONNIVER -- a Genetic Approach", Proceedings of AFIPS 1972 Fall Joint Computer Conference, Vol. 41, pp. 1171-1179.

17. Griswold, Ralph E. "Extensible Pattern Matching in SNO-BOL4", Proceedings of the ACM Annual Conference, October, 1975, pp. 248-252.

18. Griswold, Ralph E., David R. Hanson, and John T. Korb. The Icon Programming Language; a Preliminary Report. Technical Report TR 78-3, Department of Computer Science, The University of Arizona, Tucson, Arizona. April, 1978.

19. Dewar, Robert B. K. SPITBOL Version 2.0. SNOBOL4 Project Document S4D23. Illinois Institute of Technology, Chicago, Illinois. February 12, 1967.