An Alternative to the Use of Patterns in String Processing

RALPH E. GRISWOLD and DAVID R. HANSON
University of Arizona

SNOBOL4 is best known for its string processing facilities, which are based on patterns as data objects. Despite the demonstrated success of patterns, there are many shortcomings associated with their use. The concept of patterns in SNOBOL4 is examined and problem areas are discussed. An alternative method for high-level string processing is described. This method, implemented in the programming language Icon, employs *generators*, which are capable of producing alternative values. Generators, coupled with a goal-driven method of expression evaluation, provide the string processing facilities of SNOBOL4 without the disadvantages associated with patterns. Comparisons between SNOBOL4 and Icon are included and the broader implications of the new approach are discussed.

Key Words and Phrases: pattern matching, string processing, programming languages, SNOBOL4
CR Categories: 4.2, 4.20, 4.22

1. INTRODUCTION

SNOBOL4 is certainly best known for its pattern-matching facilities [21]. Among readily available high-level languages, SNOBOL4 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 [6], and procedural mechanisms for implementing patterns are a central issue in a subsequent language, SL5 [13, 17, 22].

Patterns in a style similar to SNOBOL4 have been incorporated or proposed in a number of other languages or language variants [27, 32]. Artificial intelligence languages in particular have increasingly included patterns and pattern matching as central facilities [2, 28], and pattern matching has been important in some recent applications of artificial intelligence techniques [8]. Finally, there has also been substantial theoretical interest in string pattern matching [3, 7, 23, 30].

Considering the importance attributed to patterns, and to string patterns in particular [7], it is worthwhile to make a critical evaluation of high-level language facilities for pattern matching. This paper evaluates the pattern-matching facilities of SNOBOL4. In particular, the use of patterns as data objects is considered, concentrating on their characteristics, their advantageous and disadvantageous

This work was supported by the National Science Foundation under Grants MCS75-01307 and MCS79-03890.
Author's address: Department of Computer Science, University of Arizona, Tucson, AZ 85721.

attributes, and the degree to which they are essential as a mechanism for embodying search and backtrack facilities. An alternative to patterns that provides most of their advantages without the associated disadvantages is suggested. While other defects in SNOBOL4, such as lack of traditional control structures and the absence of high-level synthesis facilities, are not considered here, solutions to these problems are provided as a by-product of the new approach.

2. PATTERNS IN SNOBOL4

In SNOBOL4, patterns are data objects constructed during program execution. The repertoire of pattern-construction functions and operators provides a variety of patterns and permits the construction of 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 to match, 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 [10, 11, 13, 15, 21].

2.1 Advantages of the Pattern Approach

The richness of the SNOBOL4 pattern facility is illustrated by the pattern-constructing operations and the corresponding processes that occur during pattern matching. There are 16 operations that construct patterns. Of these, five are concerned with positions in the subject:

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

Four operations construct "lexical" patterns, whose actions depend on the character structure of the subject:

ANY(S)	match any character in S
NOTANY(S)	match any character not in S
BREAK(S)	match up to (but not including) any character in S
SPAN(S)	match through characters in S

There are three operations that construct patterns that perform assignments:

@V	assign cursor value to V
P $ V	assign substring matched by P to V
P . V	assign substring matched by P to V if entire match succeeds

Patterns that control the application of other patterns are constructed by three operations:

P1 | P2	apply P1 or apply P2
P1 P2	apply P1 then apply P2
ARBNO(P)	apply pattern P an arbitrary number of times

These operations allow the composition of patterns, and hence the construction of more complex patterns out of simpler ones.

The unevaluated expression operation, *X, defers the evaluation of the expression X until pattern matching. This feature allows the effect of recursive pattern matching as well as computation during pattern matching.

There are seven built-in patterns. Three are concerned with specific types of matching:

ARB	match an arbitrary string
BAL	match a parenthesis-balanced string
REM	match remainder of subject

Four built-in patterns are concerned with control of the matching process:

ABORT	terminate matching with failure
FAIL	fail to match
FENCE	abort during backtracking
SUCCEED	succeed during backtracking

Finally there are two key words that control global aspects of pattern matching. &ANCHOR determines whether patterns must match at the first character of the subject. &FULLSCAN controls the use of heuristics designed to improve the efficiency of pattern matching.

Patterns as data objects also provide an abstraction mechanism. For example, if the value of LETTERS is a string of all alphabetic characters,

```
GETWORD = BREAK(LETTERS) SPAN(LETTERS) . WORD
```

is a pattern that locates a "word" and assigns the result to WORD. Used in pattern matching, this pattern operates as an abstraction much like a function. For example,

```
TEXT   GETWORD      :S(YES)F(NO)
```

identifies a word in TEXT, if there is one.

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 since the cursor changes automatically, neither of these variables has to be specified in the pattern. Thus complex operations can be expressed concisely. This characteristic is evident in the example above; the specification for GETWORD requires no reference to the string to which it is applied nor to where in this string the first letter is found.

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 understand beyond an abstract functional level, much less implement. The algorithm includes the maintenance of state information and the reversal of effects during backtracking. Thus the programmer can specify a desired construction without having to program the algorithm for applying it. For example,

```
KEY = ("re" | "trans") ("form" | "port") ("er" | "s")
```

is a pattern that matches strings such as "reformer", "reforms", "transforms", and so on. The pattern-matching algorithm assures that the application of this pattern will match any of the eight possibilities, regardless of where the pattern occurs in the subject. In this simple example, a matching algorithm is obvious. However, KEY can be used in more complex contexts such as

```
KEYWORD = (POS(0) | " ") KEY (ANY(",.;:!?") | RPOS(0))
```

and so on. Regardless of the complexity of the pattern, the pattern-matching algorithm exhaustively searches for all alternatives.

One of the special aspects of patterns lies in their ability to characterize properties of strings in a manner similar to the way in which context-free grammars characterize context-free languages. Patterns viewed in this way provide an easy method for emulating static grammatical characterizations and, for example, for constructing recognizers without the need to know how the recognition process is carried out. This use also illustrates the value of patterns as data objects. Patterns can be composed from simpler ones using construction operators that parallel the grammatical concepts of subsequent and alternate. Recognizers for complicated grammars can be built in a bottom-up fashion, starting with simple components and fashioning more complex ones. The almost direct correspondence between productions of a context-free 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 forward references are computed by the evaluation of TERM and EXP during pattern matching.

In fact, a direct translation between context-free grammars and patterns can be made by deferring evaluation of all patterns [9, 14]. 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
```

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 grammar systems. Thus

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

are both more concise and more efficient than the alternation of individual characters. Of course SNOBOL4 allows the specification of context-sensitive constructions and in general provides a much richer expressive facility than context-free grammar systems.

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 [30].

Similarly, while the implicit pattern-matching algorithm is helpful in formulating complex string analysis, programmers frequently lack confidence in the correctness of complicated patterns. Hidden intricacies of the matching algorithm 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 implementers of the language are forced to resort to listings of the system for answers.

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 artificial intelligence languages [31]. A simple example of the problem in SNOBOL4 is illustrated by the application of the pattern KEY to a subject consisting of "reforming". The first and second components of the pattern match "reform", but when "er" and "s" fail to match, the algorithm backtracks to the second component to try the alternative "port" which obviously cannot match, since a different literal string has already been found in this position. While there is an obvious solution in this case and there are classes of related problems that can be treated easily [6], the general difficulty remains. SNOBOL4 does provide a few source-language mechanisms for controlling the matching algorithm, but these are rarely used and tend to aggravate the problem of understanding the processes that go on during matching.

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. 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. (Some implementations of SNOBOL4 recognize constant in-line patterns during compilation and place them out of the line of actual program execution [5].)

Patterns cause many problems in program structuring. The necessity of using side effects is particularly troublesome. The assignment to WORD in the pattern GETWORD given above illustrates this problem.

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. The issue tends to defeat the use of patterns as an abstraction mechanism. Furthermore, patterns, unlike functions, cannot be given arguments at the site of application. The need for parameterization frequently results in the use of a number of similar, but distinct patterns. For example, if words are to be identified at several places in a program, but different identifiers are needed for assignment of the words, the pattern GETWORD given above cannot be used, since the identifier WORD is an integral part of the pattern and cannot be specified when GETWORD is applied. Similarly, 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. Some pattern operations, such as ANY(S), have no counterpart outside pattern matching, while others have similar, but significantly different, parallels inside and outside pattern matching. For example, there are three forms of assignment inside pattern matching and only the standard assignment operation outside. Similarly, expressions are executed sequentially outside pattern matching, while inside pattern matching their execution may involve backtracking. For instance, outside pattern matching P1 P2 denotes string concatenation, while inside it results in sequential application of P1 and P2 with search for alternatives and backtracking.

In a very real sense, SNOBOL4 is composed of two languages, a basic language $\mathscr{L}$ and a pattern-matching language $\mathscr{P}$. This linguistic dichotomy produces a total vocabulary that is large and 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 $\mathscr{L}$ and $\mathscr{P}$. In $\mathscr{L}$, patterns for $\mathscr{P}$ are constructed. When a pattern match is invoked in $\mathscr{L}$, control is transferred to $\mathscr{P}$, where the matching procedures for the pattern are then executed. Thus $\mathscr{L}$ has the operations necessary for describing programs in $\mathscr{P}$, but not for carrying out their actions. Pattern construction is essentially the compilation of such programs for $\mathscr{P}$. In typical SNOBOL4 programs, programs for $\mathscr{P}$ are continually compiled and executed. Note that the vocabulary of $\mathscr{L}$ is increased by having to describe programs in $\mathscr{P}$ and that compilation of programs for $\mathscr{P}$ during the execution of $\mathscr{L}$ is an inherently expensive process.

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

In $\mathscr{P}$, operations of $\mathscr{L}$ 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 $\mathscr{L}$ but not of $\mathscr{P}$. On the other hand, $\mathscr{L}$ 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 $\mathscr{P}$ to get the substring up to the first comma. The second part is performed in $\mathscr{L}$ to test the length of this substring.

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

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

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

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

The advantage of this solution is that the formation of a substring and the computation of its length is avoided. However, all of these solutions have evident problems. Each of them requires assignment to a global variable as a side effect in order to have the information necessary to do a simple computation in $\mathscr{L}$.

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

2.3 SNOBOL4 Patterns in Perspective

To summarize the preceding sections, string patterns as embodied in SNOBOL4 have a number of valuable aspects:

(1) Powerful and extensive 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, especially the necessity for using side effects.
(6) Inefficiency inherent in the run-time construction of patterns.
(7) Dichotomy of languages, with a further increase in total vocabulary and a linguistic schism.
(8) Lack of mechanism for defining matching procedures.

A number of attempts have been made to solve these problems by extending $\mathscr{P}$. Suggestions have been made for adding string synthesis facilities [6], for adding programmer-defined matching procedures [12], and for providing more control over the matching algorithm [6]. These proposals provide much of the basis for SL5 [13, 17, 22]. Expanding the $\mathscr{P}$ component has hardly eliminated the need for the $\mathscr{L}$ component. In fact, the $\mathscr{L}$ 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 $\mathscr{L}$ component and to eliminate $\mathscr{P}$. The major additions to the $\mathscr{L}$ 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 [20]. Icon resembles SL5 more than it does SNOBOL4. It has an expression-oriented 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 $\mathscr{P}$ 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. More comprehensive descriptions of Icon are given in [16, 18, 19, 24] and implementation is discussed in [24].

3.2 Automatic Bookkeeping

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

scan *s* **using** *e*

establishes a global subject *s* to which string processing operations in *e* apply. The expression *e*, which can include any operations 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 the **scan** expression is the result returned by *e*.

Most scanning operations deal with character positions within the subject. In Icon, character positions are between characters and numbered from the left starting at 1. For example, the positions in SUBJECT are

```
  S   U   B   J   E   C   T
↑   ↑   ↑   ↑   ↑   ↑   ↑   ↑
1   2   3   4   5   6   7   8
```

Note that the position after the last character may be specified. It is also useful to specify positions from the right end of a string; nonpositive numbers starting at 0 and continuing with negative values specify positions from the right end toward the left, e.g.,

```
    S     U     B     J     E     C     T
 ↑     ↑     ↑     ↑     ↑     ↑     ↑     ↑
−7    −6    −5    −4    −3    −2    −1     0
```

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

```
scan s using 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. The operation *upto*(*c*) does not construct a pattern, but simply carries out the analysis. In SNOBOL4, a similar operation, 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 to the string processing concepts in Icon, although it has pragmatic importance.)

Another important point is that the expression *e* in the **scan-using** construction 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. 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. Both operations return the substring between the previous and new cursor positions. Again, there are analogies to the SNOBOL4 operations TAB(n) and LEN(n), although *tab*(*n*) and *move*(*n*) operate immediately upon invocation rather than constructing patterns. An example of the use of such a scanning operation is[1]

```
scan s using
  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, with the consequent need for an auxiliary variable, 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
  scan s using
    repeat write ("[" || move(2) || "]")
```

(The **repeat** construct repeatedly invokes *write* until failure occurs because the cursor cannot be advanced two positions.)

The subject and cursor position are directly accessible in Icon as key words

[1] || denotes string concatenation.

&subject and *&pos*. Assigning a value to *&subject* establishes the subject for string scanning. *&pos* is automatically set to 1 when *&subject* is set. *&pos* 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 can be written more concisely as

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

The advantage of the **scan** expression is that the current subject and cursor are saved before *e* is evaluated and restored after *e* is evaluated. In fact,

```
scan s using e
```

is essentially equivalent to

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

where *push*(*x*) and *pop*(*x*) represent internal stack operations for saving and restoring values. Since the **scan** expression suppresses a substantial amount of detail, nested string scanning is easily obtained. For example, the following section of program examines a string of items separated by commas, printing the items that contain the letter "x".

```
scan s using
  repeat {
    scan tab(upto(",")) using {
      if upto("x") then
        write(&subject)
      }
    move(1)
    }
```

The value of *tab*(*upto*(",")) is the subject of an inner scanning operation that prints an item only if it contains an "x". Once this operation is complete, the cursor is advanced one position in the subject of the outer scan.

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.

As previously described, the value of both *move*(*n*) and *tab*(*n*) is the substring between the previous and new cursor positions (regardless of the direction of cursor movement). Both operations fail if the resulting cursor position is not in the range of the subject. As a consequence of the way in which positions are designated, *tab*(0) positions the cursor past the last character of the subject.

The value of *&pos* is always positive. If a nonpositive value is assigned to *&pos* to specify a position relative to the right end of the string without having to compute the length of the string, the conversion to the corresponding positive position is provided automatically. This device suppresses detail and avoids bothersome computation. Thus, if the subject is "portability",

&pos := 0

actually sets *&pos* to 12, and subsequently

j := *&pos*

sets *j* to 12, not to 0.

The lexical scanning operations in Icon are more extensive than those in the $\mathscr{P}$ component of SNOBOL4:

upto(*c*)
many(*c*)
any(*c*)
bal(*c*1, *c*2, *c*3)
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 position 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 position or return a string; the effect of BREAK(c) in SNOBOL4 is obtained by *tab*(*upto*(*c*)).

The scanning operation *many*(*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 position is 2, the value of *many*("aeiou") is 4. The operation fails if the character of the subject at the current 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*. (Character sets have a number of other uses; see [16] for details.)

The scanning operation *bal*(*c*1, *c*2, *c*3) 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, *c*1 and *c*2 are character sets that specify the left and right balancing characters. Furthermore, *c*3 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, the following defaults are used if the arguments are null:

*c*1	"("
*c*2	")"
*c*3	any character

Thus *bal*( ) is similar to the matching procedure for the pattern BAL in SNOBOL4.

The scanning operation *find*(*s*) returns the position just before the first occurrence of the string *s* in the subject, starting at the current cursor position. Thus, if the subject is "mississippi" and the position 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 an initial substring of the subject starting at the current cursor position. Thus, if the subject is "mississippi" and the position is 2, the value of *match*("is") is 4. The operation fails if *s* is not an initial substring of the subject at the current cursor position. Thus, for the subject above, if the position were 1, *match*("is") would fail. For convenience, the expression =*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 *e*1 | *e*2 is equivalent to the operation performed in SNOBOL4 when the pattern constructed by P1 | P2 is evaluated in $\mathscr{P}$.

This operation is actually fairly complex and deserves discussion. The most obvious aspect of alternation is that *e*1 is evaluated first and if that evaluation succeeds, the result is the result of the entire expression. However, if evaluation of *e*1 fails, *e*2 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 "reevaluation" 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 alternative values is meaningful for many operations and is used not just in string scanning but throughout Icon [19].

The scanning operation *upto*(*c*) is, in fact, a generator. For *upto*(*c*), the behavior is like that for the matching procedure for BREAKX(c) in the SPITBOL dialect of SNOBOL4 [4]. 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 *e*1 & *e*2, which succeeds only if both *e*1 and *e*2 succeed. In requiring this "mutual" success, there is automatic backtracking for alternatives of *e*1 if *e*2 fails. This operation 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 position 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 reevaluated for an alternative. This time the value of *upto*("i") is 5, *tab*(*upto*("i")) moves the cursor correspondingly, and ="issip" succeeds.

Since *tab*(*upto*(*c*)) is equivalent to matching for the SPITBOL pattern BREAKX(c), 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 straightforward SPITBOL equivalent to *j* := *upto*(*c*)). This tradeoff 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 *bal*(*c*1, *c*2, *c*3) and *find*(*s*). For *bal*(*c*1, *c*2, *c*3), the alternatives are as in the SNOBOL4 pattern BAL—successively longer balanced strings. For *find*, the alternatives are positions of *s* successively further on in the subject.

The full range of search and backtracking in SNOBOL4 pattern matching is available in the Icon expressions *e*1 | *e*2 and *e*1 & *e*2. It is important to note that *e*1 & *e*2 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*(*c*1)) & *y* := *tab*(*upto*(*c*2))

succeeds only if the subject contains a character of *c*2 in a position at or beyond a character of *c*1, while in the sequence of expressions

x := *tab*(*upto*(*c*1)); *y* := *tab*(*upto*(*c*2))

this constraint does not apply. A value may be assigned to *y* even if the subject does not contain a character in *c*1.

It is not necessary to require the mutual success of two expressions in order to obtain the alternatives of a generator. The control structure

every *e*1 **do** *e*2

causes *e*1 to generate its alternatives in sequence, evaluating *e*2 for each alternative generated by *e*1. An example is

every *j* := *find*(*s*) **do** *write*(*j*)

which prints all the positions at which *s* occurs as a substring of the current subject.

In backtracking over an instance of *move*(*n*), *tab*(*n*), or =*s*, 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 position is 1, evaluation of

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

first sets the position to 10 but then restores it to 1 when ="a" fails. Other effects are not reversed. In the expression

```
&pos := 10 & ="a"
```

the value of the cursor position is not restored, since it is 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.

A typical Icon procedure is

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

As shown in this example, procedures may return values using the expression **return** *e*. If *e* fails, the procedure call fails. The expressions **succeed** *e* and **fail** are similar to **return** but return the indicated signal. Arguments are transmitted by value.

Since scanning operations are on a par with all other operations, procedures may be used for scanning in the same way in which 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)
  local j
  if j := find(s) then
    return j + length(s)
  else fail
end
```

If *s* is found in the subject, the appropriate value is returned. Otherwise, the procedure fails.

Defined generators are obtained by using **suspend**, which returns a value like **return**, but leaves the procedure activation in a state in which 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)
  local j
  every j := find(s) do
    suspend j + length(s)
  fail
end
```

Like built-in generators, different instances of defined generators may be suspended simultaneously without interfering with each other.

A more esoteric application of procedures is in the use of defined generators in a fashion similar to SNOBOL4 patterns to characterize context-free languages. 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

scan "aabaa" **using** *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 complicated grammars with a procedure for each nonterminal symbol. The correspondence between context-free grammars and defined scanning procedures is just as direct as the correspondence between context-free grammars and SNOBOL4 patterns, if a bit lengthier. Perhaps more importantly, other computations, such as the construction of a parse tree, can be added to the appropriate defined scanning procedures. Such computations cannot be done as easily in SNOBOL4 because there is no way to write matching procedures.

4. DISCUSSION

As demonstrated in Section 3, generators in Icon give the programmer more control over the search and backtracking necessary in many string processing applications than can be exercised in SNOBOL4. It is this programmer control and the bookkeeping implicit in the maintenance of a *&subject* and *&pos* that make the Icon facilities preferable to, say, a well-designed library of functions. Although a set of library functions would reduce the size of the language, the loss of the evaluation mechanism of the built-in facilities would require more work on the part of the programmer.

There are cases, however, where simple lexical functions would serve as well as the more general scanning facility. SL5 included a number of lexical functions in addition to its scanning facility, but this approach results in a large vocabulary and deepens the linguistic schism described in Section 2.3. In Icon, this problem is avoided by permitting most of the scanning operations to be used as simple lexical functions if additional arguments are supplied. For example, *find*($s1$) operates as described in Section 3.3, while *find*($s1$, $s2$) operates simply as a lexical function, returning the position of the first occurrence of $s1$ in $s2$. The form *find*($s1$, $s2$, i, j) limits the examination of $s2$ to the substring between positions i and j. This polymorphic property of the scanning operations is a result of having them as built-in functions, so that the compiler can supply the correct defaults for omitted arguments. A more elaborate type system, such as that used in Alphard [29] or CLU [25], would be required to support a similar mechanism for defined procedures. A built-in procedure can of course be replaced by a defined procedure of the same name, but the specialized defaults are not automatically provided for the defined procedure.

Not surprisingly, however, the Icon approach to string scanning does present some interesting problems. One problem is the choice of primitive scanning operations, which is similar to what constitutes a "well-designed" function library for string manipulation. 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, but the automatic reversal of the assignment to *&pos* would still be provided. 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 string scanning. The historical influence of SNOBOL4 tends to inhibit new views. More experience with scanning should provide new insights.

The global nature of *&subject* and *&pos* is another problem. In Icon, scanning operations on the same subject tend to be more extensive than in SNOBOL4 because any language constructs may appear in scanning expressions. This is the reason that setting *&subject* directly is frequently more useful than the implicit setting of the subject in the **scan** expression. However, it then becomes more likely that the subject or cursor position 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 position, the results may be catastrophic.

The generally recognized hazards of global variables [34] are magnified here because of the frequency with which the two globals, *&subject* and *&pos*, are used. This appears to be a dilemma, since much of the virtue of string scanning is derived from the global nature of these variables.

It should be noted with respect to terminology that there are other languages that use "generator" to describe language facilities; Cobol and Algol 68 [33] are two languages in which the use of the term is completely unrelated to its use in Icon. One of the first languages having a concept of generators is IPL-V [26] in which a generator is a subroutine that processes the elements of a data structure by calling another routine for each element. More recently, generators appear in CLU [1, 25] and Alphard [29] where they are used in looping constructs to iterate over the elements of a data structure. Their use is restricted to specific constructs and they lack the evaluation mechanism associated with Icon generators.

5. CONCLUSION

Icon demonstrates that the advantages of pattern matching in SNOBOL4 can be achieved without patterns as data objects and that the linguistic dichotomy of SNOBOL4 is not an essential property of high-level string processing.

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 coordinated scanning of two or more subjects would be useful. 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 attractive.

It is important to note that the search and backtracking facilities of Icon are not limited to string scanning. These facilities allow a more natural expression of some constructions than is possible in most other programming languages. Examples are

if $(x \mid y) = (m \mid n)$ **then** $f(x, y)$
if $(x = n)$ & $(n > y)$ **then** $f(x, y)$
if $x < (n \mid m) < y$ **then** $f(x, y)$
every $(n := f(x))$ & $(n > 0)$ **do** $g(n)$

Such constructions are closer to the way that programmers think in mathematical and natural languages than typical programming languages allow. More experience with the use of such constructions may also lead to the development of new control structures for expressing alternatives, search strategies, and mutually necessary conditions.

ACKNOWLEDGMENTS

We are indebted to Tim Korb for a number of suggestions and ideas that are incorporated in this work, as well as for advice on the presentation. The suggestions of Susan Graham and the referees were also very helpful.

REFERENCES

1. Atkinson, R. Toward more general iteration methods in CLU. CLU Design Note 54, M.I.T., Cambridge, Mass., Sept. 1975.
2. Bobrow, D.G., and Raphael, B. New programming languages for artificial intelligence research. *Comput. Surv. 6*, 3 (Sept. 1974), 153-174.
3. Boyer, R.S., and Moore, J.S. A fast string searching algorithm. *Commun. ACM 20*, 10 (Oct. 1977), 762-772.

4. DEWAR, R.B.K. SPITBOL version 2.0. SNOBOL4 Proj. Doc. S4D23, Illinois Inst. Technology, Chicago, 1971.
5. DEWAR, R.B.K., AND MCCANN, A.P. MACRO SPITBOL—A SNOBOL4 compiler. *Softw., Pract. Exper. 7*, 1 (Jan.-Feb. 1977), 95-114.
6. DOYLE, J.N. A generalized facility for the analysis and synthesis of strings and a procedure-based model of an implementation. M.S. thesis, Univ. Arizona, Tucson, 1975.
7. FLECK, A.C. Formal models for string patterns. In *Current Trends in Programming Methodology, vol.* IV, *Data Structuring*, R. T. Yeh, Ed. Prentice-Hall, Englewood Cliffs, N.J., 1978, pp. 216-240.
8. FRASER, C.W. A compact, machine-independent peephole optimizer. In Conf. Rec. 6th Annu. ACM Symp. Principles Programming Languages, Jan. 1979, pp. 1-6.
9. GIMPEL, J.F. *Algorithms in SNOBOL4.* Wiley, New York, 1976.
10. GIMPEL, J.F. A theory of discrete patterns and their implementation in SNOBOL4. *Commun. ACM 16*, 2 (Feb. 1973), 91-100.
11. GIMPEL, J.F. Nonlinear pattern theory. *Acta Inf. 4*, (1975), 91-100.
12. GRISWOLD, R.E. Extensible pattern matching in SNOBOL4. In Proc. ACM Annu. Conf., Oct. 1975, pp. 248-252.
13. GRISWOLD, R.E. String analysis and synthesis in SL5. In Proc. ACM Annu. Conf., Oct. 1976, pp. 410-414.
14. GRISWOLD, R.E. *String and List Processing in SNOBOL4, Techniques and Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1975, pp. 12, 233-234.
15. GRISWOLD, R.E. *The Macro Implementation of SNOBOL4, A Case Study in Machine-Independent Software Development.* W. H. Freeman, San Francisco, 1972.
16. GRISWOLD, R.E. The use of character sets and character mappings in Icon. *Comput. J.*, to appear.
17. GRISWOLD, R.E., AND HANSON, D.R. An overview of SL5. SIGPLAN Notices *12*, 5 (Apr. 1977), 40-50.
18. GRISWOLD, R.E., AND HANSON, D.R. Reference manual for the Icon programming language. Tech. Rep. TR 79-1, *Dep. Comput. Sci., Univ.* Arizona, Tucson, Jan. 1979.
19. GRISWOLD, R.E., HANSON, D.R., AND KORB, J.T. Generators in Icon. *ACM Trans. Program. Lang.*, submitted for publication.
20. GRISWOLD, R.E., HANSON, D.R., AND KORB, J.T. The Icon programming language: An overview. SIGPLAN Notices *14*, 4 (April 1979), 18-31.
21. GRISWOLD, R.E., POAGE, J.F., AND POLONSKY, I.P. *The SNOBOL4 Programming Language*, 2nd ed. Prentice-Hall, Englewood Cliffs, N.J., 1971.
22. HANSON, D.R., AND GRISWOLD, R.E. The SL5 procedure mechanism. *Commun. ACM 21*, 5 (May 1978), 392-400.
23. KNUTH, D.E., MORRIS, J.H., AND PRATT, V.R. Fast pattern matching in strings. *SIAM J. Comput. 6*, 2 (June 1977), 323-350.
24. KORB, J.T. The design and implementation of a goal-directed programming language. Ph.D. dissertation, Univ. Arizona, Tucson, 1979.
25. LISKOV, B.H., ET AL. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564-576.
26. NEWELL, A., ED. *Information Processing Language-V Manual* (Rand Corp.), Prentice-Hall, Englewood Cliffs, N.J., 1961.
27. PFEFFER, A.S., AND FURTADO, A.L. Pattern matching for structured programming. In Proc. 7th Asilomar Conf. Circuits, Systems, and Computers, Pacific Grove, Calif., 1973, pp. 466-469.
28. Proceedings of the workshop on pattern-directed inference systems. SIGART Newsletter *63* (June 1977), 1-84.
29. SHAW, M., WULF, W.A., AND LONDON, R.L. Abstraction and verification in Alphard: Defining and specifying iteration and generators. *Commun. ACM 20*, 8 (Aug. 1977), 553-564.
30. STEWART, G.F. An algebraic model for string patterns. In Conf. Rec. 2nd Annu. ACM Symp. Principles of Programming Languages, Jan. 1975, pp. 167-184.
31. SUSSMAN, G.J., AND MCDERMOTT, D.V. From PLANNER to CONNIVER—A genetic approach. *Proc. 1972 AFIPS, Fall Jt. Computer Conf.*, vol. 41, AFIPS Press, Arlington, Va., pp. 1171-1179.
32. TESLER, L.G., ENEA, H.J., AND SMITH, D.C. The LISP70 pattern matching system. In Proc. 3rd Int. Jt. Conf. Artificial Intelligence, Stanford, Calif., 1973, pp. 671-676.

33. VAN WIJNGAARDEN, A., ET AL. Revised report on the algorithmic language Algol 68. *Acta Inf.* *5* (Jan. 1976), 1-236.
34. WULF, W., AND SHAW, M. Global variables considered harmful. SIGPLAN Notices *8*, (Feb. 1973), 28-34.

Received May 1978; revised August and December 1979; accepted December 1979