The Control of Searching and Backtracking in String Pattern Matching*

Ralph E. Griswold

TR 82-20

December 28, 1982

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

*This work was supported by the National Science Foundation under Grant MCS81-01916.

The Control of Searching and Backtracking in String Pattern Matching

The efficiency of searching and backtracking has long been a concern of the designers and implementors of high-level facilities for pattern matching on strings. In fact, backtracking was omitted from pattern matching in COMIT [23] because of concerns about its potentially exponential behavior [24].

The SNOBOL4 language [18], on the other hand, supports an exhaustive depth-first algorithm for pattern matching. This approach was motivated by the value of a pattern in characterizing a set of strings in a general and uniform way. If pattern matching is not general and exhaustive, the set of strings matched by a pattern may be difficult to comprehend and the value of the pattern as an abstraction is consequently diminished.

This report describes implementation techniques, design decisions, and experience related to these problems in the SNOBOL4 language. As a comparison, the somewhat different approach taken in the design of Icon [1, 15] is discussed in the latter part of this report.

1. Pattern-Matching Heuristics in SNOBOL4

In the early SNOBOL languages [4, 5], a pattern consisted only of the concatenation (conjunction) of a few primitive pattern components: (1) literal strings, (2) strings of a specified length but arbitrary characters, (3) strings of arbitrary characters and arbitrary length, and (4) strings balanced with respect to parentheses. Patterns were constructed during compilation and did not change during program execution.

In such a simple system, it is easy to determine the minimum length of a string that can be matched by a pattern. Furthermore, during pattern matching, the number of characters necessary for a successful match of the remaining pattern components also can be determined. Such length information was used to implement "heuristics" that terminated pattern matches that would be futile, *a priori*, simply because the string being matched was not long enough.

The additional features of SNOBOL4 complicated this aspect of the implementation. These features include (1) a large repertoire of built-in patterns and pattern-constructing operations, (2) the alternation (disjunction) of patterns, and (3) the run-time construction of patterns.

In SNOBOL4, a pattern is the concatenation (conjunction) of a number of pattern components, which are written in succession:

$$P = P_1\ P_2 \ldots P_n$$

When a pattern match is performed as in

S ? P

the subject string S serves as the focus of attention for the evaluation of P. During pattern matching, a *cursor* identifies the current position in the subject at which matching is taking place. The cursor initially starts at 0, corresponding to the beginning (left end) of the subject. P_1 is evaluated first with the cursor at zero. If P_1 matches, the cursor is advanced past the characters matched and P_2 is evaluated next. In general, if P_i matches, P_{i+1} is matched next. If P_i fails to match, however, the cursor is restored to its previous position and P_{i-1} is *resumed* for a possible alternative match. If P_{i-1} matches, P_i is evaluated again. If P_{i-1} fails to provide an alternative match, P_{i-2} is resumed, and so on. If P_n eventually matches, the entire pattern match is succeeds. If P_1 fails, the cursor is incremented and the pattern matching process starts over beginning at the next character of the subject. If P_1 fails and there are no more characters in the subject, the entire pattern match fails.

Some typical SNOBOL4 patterns that illustrate the problems involved in the control of searching and backtracking are:

s	where s is a string that matches if it occurs at the current cursor position in the subject,
LEN(n)	which matches n characters, provided that there are at least n characters in the subject starting at the current cursor position,
POS(n)	which matches zero characters (the null string) but succeeds if and only if the cursor is at n,
ANY(s)	which matches the character of the subject at the current cursor position, provided that character is contained in the string s,
ARB	which matches the null string first, but extends the substring it matches by one character every time it is resumed,
ARBNO(P)	which matches whatever P matches zero or more times, starting with zero (the null string) and matching P once more each time ARBNO(P) is resumed,
P $ V	which matches whatever P matches and assigns the substring that is matched to the variable V,
P1 | P2	which matches P1 or, if that fails, P2, and
*X	which evaluates the expression X and then matches the pattern X produces.

Note that incrementing the cursor when P fails is equivalent to placing ARB in front of the entire pattern and omitting the automatic incrementing of the cursor for the first pattern component. There is an "anchored" mode of pattern matching that prevents the cursor from being incremented if the first component of the pattern fails.

The match for a string s, as well as for LEN(n), POS(n), and ANY(s) fail if they are resumed. The behavior of ARBNO(P) and P $ V when resumed depends on P. When P1 | P2 is resumed, the pattern that matched is resumed. The pattern that *X produces is resumed during backtracking.

In the original SIL (macro) implementation of SNOBOL4 [9], conjunction and alternation are treated as structural relationships and a pattern is implemented as a tree of pattern nodes. Each node contains information about the length of string required for the current pattern component to match as well as the minimum length for subsequent components.

1.1 The Futility Heuristic

The futility heuristic is designed to prevent unnecessary matching in situations where there are not enough characters to satisfy the requirements of the pattern. This may occur because the pattern, *a priori*, requires more characters than there are in the subject or because at some point during the pattern match there are not enough characters remaining in the subject to satisfy the remaining pattern components. Thus in

```
"abcd"   ?   LEN(4) "d"
```

the match it not attempted, since the pattern requires 5 characters. Consider also

```
"abcd"   ?   (ANY("xy") LEN(3)) | (ANY("cd") LEN(1) ANY("cd"))
```

Here the pattern itself requires only 4 characters, so the pattern match is attempted. However, the first alternative fails to match. The first component of the second alternative would match with the cursor at 2, but there are not enough characters left in the subject to satisfy the remaining two components of the second alternative.

The minimum lengths required for the various patterns are easily determined. Let ℓ(P) represent the minimum number of characters require to match P. Then

ℓ(s) = size(s)
ℓ(LEN(n)) = n
ℓ(POS(n)) = 0
ℓ(ANY(s)) = 1
ℓ(ARB) = 0
ℓ(ARBNO(P)) = 0
ℓ(P1 | P2) = min(ℓ(P1),ℓ(P2))
ℓ(P1 P2) = ℓ(P1) + ℓ(P2)

Length considerations also arise in backtracking. Two kinds of failure are distinguished: match failure and length failure. Match failure occurs when a pattern component simply does not match at the current cursor position. Length failure occurs if there are not enough characters remaining in the subject string to satisfy the remaining components of the pattern. For example, POS(n) produces match failure if the cursor is not at n, while ARB produces length failure if it is resumed and the cursor is at the end of the subject.

When failure occurs, the type of failure is transmitted backward during backtracking and is used to determine whether or not to resume pattern components that previously matched. For example, if ARB is encountered during backtracking because of match failure it is resumed and extends the substring it previously matched by one character, provided the cursor is not at the end of the subject. If ARB is encountered during backtracking because of length failure, however, it is not resumed since ARB can only increase the length of the substring it matches and hence decrease the length available for subsequent pattern components. In this case, backtracking continues to the next previous component.

In effect, length failure suppresses the unnecessary resumption of previously matched pattern components. Length failure does not imply that the entire pattern may not eventually match. For example, in

```
P1 | P2
```

P2 may match a shorter substring than P1 matches.

See [22] for a more extensive discussion of length considerations in pattern matching.

1.2 Handling ARBNO(P)

ARBNO(P) presents a technical problem, since P may match the null string. For example,

```
ARBNO(LEN(0))
```

is a legal, if useless, pattern. When ARBNO(LEN(0)) is first evaluated, it matches the null string, corresponding to zero instances of LEN(0). However, if ARBNO(LEN(0)) subsequently is resumed, it matches LEN(0), which matches the null string and leaves the cursor position unchanged. Thus pattern matching "gets stuck" at this point if subsequent pattern components continue to fail to match.

This problem is handled by a separate heuristic that does not resume ARBNO(P) if P last matched the null string.

1.3 Handling Left Recursion

Unevaluated expressions can be used to produce the effect of a recursive pattern. For example,

```
P = "a" | ("b" *P)
```

creates a pattern that references itself. The unevaluated expression defers the determination of the value of P until matching takes place; otherwise it would be determined when the pattern was built and would refer the its previous value of P, not to itself. Thus P characterizes the set of strings a, ba, bba, bbba, ... On the other hand, the following pattern is also legal:

```
P = (*P "a") | "b"
```

This pattern characterizes the set of strings b, ba, bba, ... However, this pattern is left recursive, since the first component that is matched for P is P itself. Even patterns such as

```
P = *P
```

that do not characterize any set of strings are nonetheless legal in SNOBOL4.

To prevent recursive processing loops that would result from left recursion in patterns, every unevaluated expression is assumed to match at least one character. This has the effect of eventually producing length failure in left-recursive situations, even if the unevaluated expression in fact matches the null string.

1.4 The Linguistic Effect of the Heuristics

The heuristics described above introduce problems of two kinds. In the first place, they may be "visible". Consider the futility heuristic. In the early SNOBOL languages, this heuristic only affected the running speed of programs and its presence did not affect the computations performed by a program. In SNOBOL4, however, value assignment during pattern matching is a side effect that occurs even if the entire pattern match eventually fails. By effectively pruning search paths that would be futile, an assignment may not be made that otherwise would have been made. A simple example is

```
"abcd"  ?  (LEN(3) $ V) LEN(2)
```

In the absence of heuristics, abc is assigned to V, while with the heuristics, the match for the first component is not attempted.

The "one-character" assumption creates a potentially more serious problem, since the assumption may be false and a match may not be attempted even though it would succeed. The same is true of the heuristic for handling ARBNO(P).

In any event, the heuristics tend to conflict with the design goal of a pattern as an abstraction for a set of strings. Even if the potential problems do not arise in practice, they tend to undermine the confidence of programmers in the correctness of their programs and in the implementation itself. This problem is aggravated by the fact that the effects of the heuristics are difficult to understand and that their nature varies from implementation to implementation.

1.5 Heuristics in Other Implementations of SNOBOL4

The SITBOL [6] implementation of SNOBOL4 provides more sophisticated and complicated heuristics than the SIL implementation.

The futility heuristic in SITBOL does not prevent pattern matching simply because there are not enough characters in the subject to satisfy the requirements of the pattern. Instead, it performs pattern matching until there are not enough characters left to satisfy a component. Thus, in SITBOL

```
"abcd"  ?  (LEN(3) $ V) LEN(2)
```

assigns abc to V even though the pattern match subsequently fails.

The recursion–breaking heuristic in SITBOL does not assume that an unevaluated expression will match at least one character, but instead relies on the minimum length required by pattern components that follow an unevaluated expression.

SITBOL has an additional start–up heuristic that attempts to determine whether the first component in a pattern can match at any position in the subject before starting the pattern match. For example, if the first component of a pattern is

```
ANY("ab")
```

the entire pattern cannot possibly match unless the subject contains an a or a b. The start-up heuristic uses the properties of the different kinds of pattern components in this way to avoid pattern matches that would be futile simply on the basis of their first component.

Note that programs run under the SIL and SITBOL implementations may behave differently. The heuristics in SITBOL are designed to be less visible than those in the SIL implementation. See [7] and [8] for a more detailed description of SITBOL heuristics.

Other implementations also treat the heuristics in somewhat different ways. See [2] for a description of the heuristics in SPITBOL 360 and [21] for a description of the heuristics in FASBOL. MACRO SPITBOL [3],

on the other hand, does not implement any pattern-matching heuristics. It is worth noting that MACRO SPITBOL is in wide use and that its lack of heuristics do not seem to cause problems for programmers.

2. Language Features for Controlling Searching and Backtracking in SNOBOL4

2.1 Control over the Heuristics

Concern about the consequences of the visibility of heuristics led to a language feature that lets the programmer turn the heuristics off and on.

Two modes of pattern matching are recognized: "quickscan", in which the heuristics are used, and "fullscan", in which the heuristics are not used. The mode is selected by setting the value of the keyword &FULLSCAN, where

```
&FULLSCAN = 1
```

turns off the heuristics and

```
&FULLSCAN = 0
```

turns on the heuristics. The heuristics are initially on. There is no way to selectively control the different heuristics individually.

2.2 Control Patterns

In addition to the heuristics, SNOBOL4 has a few *control patterns* that are designed to allow the programmer to control searching and backtracking.

The built-in pattern ABORT causes a pattern match to fail at the point that it is encountered. ABORT usually is used to avoid matching a subject string that does not fall into a desired category. For example,

```
S ? (LEN(10) ABORT) | P
```

matches S for P only if S is less than 10 characters long.

A related control pattern in FENCE, which causes a pattern match to fail if it resumed during backtracking. Thus

```
S ? P1 FENCE P2
```

fails if P1 matches but P2 does not match and prevents the resumption of P1 for possible alternative matches. Note that

```
FENCE ≡ NULL | ABORT
```

where NULL is a pattern that matches the null string (and hence always matches, regardless of the subject string).

Two related patterns are FAIL and SUCCEED. FAIL just fails to match and hence forces backtracking. SUCCEED always matches, even when it is resumed. FAIL sometimes if used to force a pattern to be resumed repeatedly to match all the strings it can. For example,

```
S ? (P $ OUTPUT) FAIL
```

assigns to OUTPUT each substring of S that is matched by P. Since assignment to OUTPUT in SNOBOL4 causes the value that is assigned to be written out also, the effect is a trace of all substrings matched by P.

2.3 The SNOBOL4 Pattern Repertoire

Many problems with unnecessary searching and backtracking can be avoided by including language features that handle frequently occurring special cases in an efficient manner. Several of the patterns in the SNOBOL4 repertoire are included for this reason. For example, ANY(s) is included because of the frequency with which one of several characters is to be matched. If ANY(s) were not in SNOBOL4, a match such as

```
S   ?   ANY("aeiou")
```

would have to be phrased as

```
S   ?   ("a" | "e" | "i" | "o" | "u")
```

The second formulation is not only less efficient (unless the implementation recognizes and handles such patterns in a special way), but it is also less compact.

Another example is BREAK(s), which advances the cursor up to the first character in the subject that is contained in s. Thus

```
S   ?   "d"
```

and

```
S   ?   BREAK("d")
```

both succeed if S contains a d. The second pattern is considerably faster, since the location of the d is done in the BREAK pattern, which is specially designed to stream through characters. In the first case, a match for d is attempted at each position of the subject, with the cursor being advanced by the pattern-matcher. In both cases, the time taken for the match is linear in the number of characters that precede the desired one, but the constant is considerably smaller when BREAK is used. For example, if the first d is 10 characters from the beginning of the second formulation is about twice as fast as the first.

Patterns such as ANY(s) and BREAK(s) are widely used and undoubtedly increase the speed of pattern matching. Such patterns tend, however, to focus on the process of pattern matching and to diminish the value of a pattern as a characterization of a set of strings.

3. Experience with SNOBOL4

3.1 The Effectiveness of the Heuristics

It is, of course, possible to construct examples for which the heuristics produce an arbitrarily large improvement in the performance of pattern matching. One type of example is typified by

```
"aaaaaaaaaaaaaaa"   ?   ARB ARB ARB ARB ARB ARB ARB ARB ARB ANY("bc")
```

This pattern match eventually fails, simply because the subject does not contain a b or c. In the fullscan mode the subject is "divided up" in all possible ways among the ARBs until the first ARB fails starting at the end of the subject. In the quickscan mode, failure occurs as soon as the last ARB moves the cursor to the end of the subject. Without the heuristics, the time required for the pattern match is exponential in the number of ARBs in the pattern. With the heuristics, the time required is a linear combination of the number of characters in the subject and the number of ARBs in the pattern.

This example suggests other possible heuristics, but it is difficult to imagine where to stop in such a process. The more important issue is whether such situations occur in practice.

The other aspect of this issue is that the heuristics take time to apply, require space for the storage of heuristic information, and complicate the pattern-matching algorithm. Yet a large percentage of pattern matches that occur in real programs do not benefit from the heuristics. For example, in

```
"abcd"   ?   ANY("cd") $ V
```

there is necessarily some overhead for processing the heuristics, although they are not relevant in this case. Again, once can imagine increasingly sophisticated implementation techniques to avoid such unnecessary processing, but there is no clear solution for the general case.

Since there is great diversity in the applications of SNOBOL4 [10] and programming styles and data vary enormously, an experimental approach to an evaluation of the heuristics is the only practical one. In one study [13], 10 SNOBOL4 programs were selected from a variety of applications ranging from theorem proving to flowchart generation. These programs were timed on representative data in both quickscan and fullscan modes and on both SIL SNOBOL4 and SITBOL. Interestingly, all programs produced the same results in

both modes and under both implementations, even though all were designed to run in the quickscan mode and some were written to run under the SIL implementation, while others were written to run under SITBOL.

Running under SIL, 9 of the 10 programs showed an increase in running speed when the heuristics were used. The largest increase in speed was 37.7%, while in two cases the speed was reduced by about 2% when the heuristics were used. The average improvement obtained was 7.6%.

The results obtained when running under SITBOL were more surprising: the largest increase obtained from using the heuristics was only 14.5% with an average improvement of only 2.9%. The running speed of three programs was not affected by the heuristics and two programs ran more slowly with the heuristics than without them. The probable cause of the degradation of efficiency by the heuristics is the overhead involved in applying them. The overall lower effectiveness of the SITBOL heuristics may be due to the use of a weaker form of the futility heuristic.

One interesting observation of this study is that the programs that benefitted the most from the heuristics generally were ones that used styles of pattern matching that are known to be inefficient [14]. In other words, if these programs were rewritten to take advantage of techniques that most SNOBOL4 programmers know, the heuristics would be less helpful.

3.2 Programmer Use of Heuristics and Control Patterns

In the same study, 400 SNOBOL4 programs, selected from a wide variety of applications and environments, were examined to determine the extent to which programmers exercise control over the heuristics and how extensively control patterns are used.

In this sample of 400 programs, there were only three references to &FULLSCAN, all using it to turn off the heuristics. Of the two control patterns specifically designed to prevent unnecessary searching and backtracking, FENCE appeared only 43 times and ABORT appeared only 56 times. In both cases, the uses were concentrated in 22 of the 400 programs.

The very infrequent use of &FULLSCAN strongly suggests that most programmers are not bothered by the possible side effects of the heuristics.

There are several possible explanations for the relatively infrequent use of FENCE and ABORT. One possibility is that they are not useful in practice. Another possibility is that programmers are not sufficiently concerned about the efficiency of pattern matching to use control patterns. Yet another possibility is the most programmers do not understand how to use control patterns or appreciate the benefits they can produce.

4. Design Alternatives

4.1 Control Structures in Pattern Matching

Aside from the control patterns, the only control structures available to the programmer are concatenation and alternation (conjunction and disjunction). Consider a traditional control structure that might be added to this limited repertoire:

```
if P1 then P2
```

The interpretation of this control structure is that the match for P2 is attempted only if the match for P1 succeeds and that if P2 subsequently fails, P1 is not resumed. This control structure usually could be used in place of

```
P1 FENCE P2
```

The potential advantage of the **if-then** control structure is that it expresses the desired behavior in a straightforward way and lends itself to understandable combinations more easily than FENCE does. In effect, **if-then** expresses the desired behavior positively, while FENCE expresses it negatively. Furthermore, **if-then** allows backtracking to be inhibited without resulting in failure of the entire pattern match.

Other conventional control structures have similar interpretations in pattern matching. For example

```
while P1 do P2
```

could be used to express iteration. With the limited control structures in SNOBOL4, recursion is needed to formulate loops in pattern matching.

Conversely, the goal-directed kind of evaluation that is inherent in pattern matching often is useful in other kinds of computations. In SNOBOL4, the language features for conventional kinds of computation and sharply divided from the features for pattern matching [16]. The unification of these features is the primary foundation for the Icon programming language [1, 15].

4.2 The Icon Programming Language

Expressions in Icon are capable of producing a sequence of values [17], just as ARB in SNOBOL4 is capable of matching a sequence of substrings. An example is the Icon function

```
find(s1, s2)
```

which produces the position in s2 at which s1 occurs as a substring. If s1 does not occur in s2, this function fails just as a pattern match may fail in SNOBOL4. If s1 occurs as a substring in s2, that position is returned. Thus

```
find("abc", "aabc")
```

produces the value 2. (Positions in strings are numbered starting at 1 in Icon, while they start at 0 in SNOBOL4.)

However, s1 may occur at several positions in s2. An example is

```
find("abc", "aabcabcaabc")
```

in which the positions are 2, 5, and 9.

Expression evaluation in Icon is *goal-directed* just as pattern matching is in SNOBOL4. If one value produced by an expression does not result in successful evaluation in the surrounding context, the expression is resumed to produce another value. Thus

```
find("abc", "aabcabcaabc") > 5
```

succeeds, even though the first two values produced by find do not satisfy the comparison operation. Note that this expression has nothing to do with pattern matching. In fact, a similar expression can be written using the PL/1 function INDEX, although the comparison in PL/1 is not successful, since INDEX produces only one value.

Traditional control structures in Icon, such as **if-then-else**, use the success or failure of a control expression in place of the Boolean values used in Algol-style languages. For example,

```
if find(s1, s2) then expr1 else expr2
```

evaluates $expr_1$ if s1 occurs as a substring of s2, but evaluates $expr_2$ otherwise.

Goal-directed evaluation causes expressions to be resumed implicitly. Expressions also can be resumed explicitly to produce all their values. This is done with the **every-do** control structure, as in

```
every i := find(s1, s2) do write(i)
```

which writes all the positions at which s1 occurs as a substring of s2. Note that **every-do** repeatedly *resumes* an expression, while **while-do** repeatedly *evaluates* an expression (that is, **while-do** produces the first value of an expression repeatedly).

One interesting control structure in Icon limits the number of values that an expression can produce:

```
expr \ i
```

This expression limits *expr* to at most i values. Thus

```
every i := (find(s1, s2) \ 10) do write(i)
```

writes at most the first 10 positions at which s1 occurs as a substring of s2.

Note that Icon allows the programmer to use each value in the sequence that is produced by an expression. In SNOBOL4, the individual substrings matched by a pattern are not readily available to the programmer. Icon focusses on the sequence of values produced by an expression, while SNOBOL4 focusses on the last value matched by a pattern.

Icon also provides for the alternation and conjunction of expressions:

$expr_1$ | $expr_2$

produces the values of $expr_1$ followed by the values of $expr_2$, analogous to the sequence of strings matched by

```
P1 | P2
```

in SNOBOL4. The expression

$expr_1$ & $expr_2$

succeeds if and only if both $expr_1$ and $expr_2$ succeed, analogous to the way that

```
P1 P2
```

matches in SNOBOL4. Conjunction in Icon is not a control structure but merely a natural consequence of goal-directed evaluation. Other operations behave the same way. For example, the concatenation operation

$expr_1$ || $expr_2$

succeeds if and only if both $expr_1$ and $expr_2$ succeed and produces the concatenation of the values produced by $expr_1$ and $expr_2$. If $expr_1$ succeeds but $expr_2$ fails, $expr_1$ is resumed to produce another value.

5. Comparison of SNOBOL4 and Icon

The sketch of some of the basic features of Icon in the preceding section suggests the potential value of the inclusion of a larger repertoire of control structures in pattern matching. Consider the following hypothetical patterns in a SNOBOL-like language:

```
if P1 then P2
(P1 \ 1) & P2
```

The first expression was compared earlier to SNOBOL4's FENCE. The second expression accomplishes the same thing by combining conjunction and limitation. There are many more possibilities, such as

```
(P1 \ 2) & P2
```

which is virtually impossible to formulate in SNOBOL4.

It is clear that if SNOBOL4 programmers had more control over the pattern matching process, patterns could be formulated to express desired relationships in a more straightforward way and, consequently, limit unnecessary searching and backtracking.

In fact, SNOBOL4-style pattern matching can be modelled in Icon [11, 12]. When this is done, the pattern matching process can be examined by tracing the pattern-matching procedures (SNOBOL4 has no facility for tracing pattern matching or defining matching procedures).

In one experiment, a language-preprocessor written in SNOBOL4 was mechanically translated into Icon. The resulting program was faithful to the semantics of pattern matching in SNOBOL4 and did not use any features of Icon that are not also in SNOBOL4. When the resulting program was run with tracing, a great deal of evidently unnecessary searching and backtracking was observed. To test the usefulness of this information, the SNOBOL4 program was modified by adding FENCEs at points where backtracking would evidently lead to unnecessary processing. When the resulting program was debugged (the insertion of appropriate FENCEs proved to be difficult), the SNOBOL4 program ran nearly twice as fast as before.

In a further experiment, SNOBOL4-style patterns were systematically replaced by more straightforward control structures in the Icon program. The resulting program ran nearly 10 times as fast as it had with only SNOBOL4-style patterns!

6. Conclusions

The potential for inefficiency in string pattern matching is high. Implementation techniques, such as the heuristics in SNOBOL4, may improve the running speed of some programs substantially, but are of little help in other programs. This does not imply, however, that the efficiency of pattern matching in these other programs cannot be improved.

Most SNOBOL4 programmers use the heuristics by default. Despite the fact that these heuristics may interfere with expected program behavior, this does not appear to be a significant problem in practice.

Control patterns can significantly improve the speed of pattern matching, but these patterns are not widely used. Special-purpose patterns, such as ANY(s) and BREAK(s), also can increase the speed of pattern matching and are used by most SNOBOL4 programmers.

Such features allow the construction of patterns that run efficiently, but they increase the vocabulary of SNOBOL4 and tend to diminish the value of a pattern as an abstract characterization of a set of strings.

One fundamental source of inefficiency in pattern matching lies in the limited control structures that are available to the programmer. A wider range of control structures, especially ones that allow pattern matching to be cast in positive, straightforward terms, can substantially increase the speed of pattern matching, as has been shown in the use of Icon.

A quite different approach is to exclude language features so that a more efficient pattern-matching algorithm can be used. See [19] and [20] for the potential of this approach.

Thus implementation techniques can be contrasted with language design as methods to improve the efficiency of pattern matching. Implementation techniques are inherently limited, since the implementation cannot know what a programmer intended. Conversely the programmer is limited by language features in specifying what a pattern is to match and how the match is to be carried out. In language design, efficiency is in basic conflict with linguistic simplicity and conciseness.

One promising area of investigation remains largely unexplored: the measurement and analysis of pattern matching. The studies reported here are based on crude overall observations of running speeds. Since SNOBOL4 provides no tracing facilities for pattern matching, serious inefficiencies may go unnoticed, being reflected only in the running speed of the entire program.

The tracing of matching procedures in the Icon model of string pattern matching suggests that instrumentation for performance measurement could be an important tool for both implementors and programmers.

Acknowledgement

Dave Hanson provided several helpful suggestions concerning the presentation of the material in this report.

References

1. Coutant, Cary A.; Griswold, Ralph E.; and Wampler, Stephen B. *Reference Manual for the Icon Programming Language; Version 5 (C Implementation for UNIX).* Technical Report TR 81-4a, December 1981.
2. Dewar, Robert B. K. *SPITBOL Version 2.0.* Technical Report S4D23, Illinois Institute of Technology, Chicago, Illinois. February 12, 1971.
3. Dewar, Robert B. K. and McCann, Anthony P. "MACRO SPITBOL—A SNOBOL4 Compiler", *Software — Practice and Experience*, Vol. 7 (1977). pp. 95-113.
4. Farber, David J.; Griswold, Ralph E.; and Polonsky, Ivan P. "SNOBOL, A String Manipulation Language", *Journal of the ACM*, Vol. 11, No. 1 (January 1964). pp. 21-30.
5. Farber, David J.; Griswold, Ralph E.; and Polonsky, Ivan P. "The SNOBOL3 Programming Language", *The Bell System Technical Journal*, Vol. XLV, No. 6 (July-August 1966). pp. 895-944.

6. Gimpel, James F. *SITBOL; Version 3.0.* Technical Report S4D30b, Bell Telephone Laboratories, Inc., Murray Hill, New Jersey. June 1, 1973.

7. 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.

8. Gimpel, James F. *Algorithms in SNOBOL4.* John Wiley & Sons, New York, New York. 1976.

9. Griswold, Ralph E. *The Macro Implementation of SNOBOL4; A Case Study of Machine-Independent Software Development.* W. H. Freeman and Company, San Francisco, California. 1972.

10. Griswold, Ralph E. *Bibliography of Documents Related to the SNOBOL Programming Languages.* Technical Report TR 78-18a, Department of Computer Science, The University of Arizona, Tucson, Arizona. September 28, 1979.

11. Griswold, Ralph E. *Pattern Matching in Icon.* Technical Report TR 80-25, Department of Computer Science, The University of Arizona, Tucson, Arizona. October 1980.

12. Griswold, Ralph E. *Models of String Pattern Matching.* Technical Report TR 81-6, Department of Computer Science, The University of Arizona, Tucson, Arizona. May 1981.

13. Griswold, Ralph E. *An Empirical Study of the Effectiveness of Pattern-Matching Heuristics in SNOBOL4.* Technical report, Department of Computer Science, The University of Arizona, Tucson, Arizona. December 22, 1982.

14. Griswold, Ralph E. and Griswold, Madge T. *A SNOBOL4 Primer.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1973.

15. Griswold, Ralph E. and Griswold, Madge T. *The Icon Programming Language.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. In press.

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

17. Griswold, Ralph E.; Hanson, David R.; and Korb, John T. "Generators in Icon", *ACM Transactions on Programming Languages and Systems*, Vol. 3, No. 2 (April 1981). pp. 144-161.

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

19. Liu, Ken-Chih. *An Efficient Algorithm for String Pattern Matching.* Doctoral dissertation, Department of Computer Science, The University of Iowa. Iowa City, Iowa. July 1977.

20. Liu, Ken-Chih and Fleck, Arthur C. "String Pattern Matching in Polynomial Time", *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas. January 29-31, 1979. pp. 222-225.

21. Santos, Paul Joseph Jr. *FASBOL, A SNOBOL4 Compiler.* Memorandum No. ERL-M134, Electronics Research Laboratory, University of California, Berkeley, California. December, 1971.

22. Waite, W. M. *Implementing Software for Non-Numeric Applications.* Prentice-Hall, Inc. Englewood Cliffs, New Jersey. 1973.

23. Yngve, V. H. "A Programming Language for Mechanical Translation", *Mechanical Translation*, Vol. 5, No. 1 (1958). pp. 25-41.

24. Yngve, V. H. "COMIT". Oral presentation, Bell Telephone Laboratories, Inc., Murray Hill, New Jersey. 1964.