Models of String Pattern Matching*

Ralph E. Griswold

TR 81-6

May 1981

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

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

Models of String Pattern Matching

1. Introduction

In an earlier report [1], a model for string pattern matching in the style of SNOBOL4 was developed using Icon. That report showed how most of the pattern-matching mechanism of SNOBOL4 could be translated into Icon and, in effect, provided a model for implementing pattern matching in the style of SNOBOL4.

This report examines the issue of string pattern matching in more depth and detail. Two models for implementing SNOBOL4 string pattern matching are presented. The first model is concerned only with the side effects of pattern matching, while the second model uses strings that are returned by the matching processes. Next a more general model is developed in which various proposals for extensions to pattern matching in SNOBOL4 can be implemented. Finally, some of the major issues of pattern matching are discussed and a proposal for a pattern-matching facility based on Icon is presented.

An understanding of pattern matching in SNOBOL4 and a working knowledge of Version 3 of Icon [2] are prerequisite to understanding the material that follows. Version 4 of Icon is used in this report, but significant differences between Versions 3 and 4 are noted.

2. Basic Concepts

In SNOBOL4, pattern matching involves a *subject*, which is examined during pattern matching, and a *cursor*, which is the position in the subject at which the examination takes place. In the Icon models of pattern matching that follow, the subject and cursor are represented by two global identifiers, declared as

```
global subject, cursor
```

Patterns are embodied in Icon procedures that obey a protocol that assures that the matching process works properly. Such procedures, called *matching procedures* [1], have the form

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

Such a procedure is said to *encapsulate* the expression *e* and the evaluation of p() produces the same outcome as the evaluation of *e*. (In Version 4 of Icon, an implicit procedure return produces no result, so the implicit return following the suspend produces no additional result.)

The protocol for matching procedures therefore amounts to rules that expressions must obey or, stated differently, determines the class of *matching expressions*.

Since only the expression *e* varies from matching procedure to matching procedure, an abbreviated syntax for matching procedures is useful. In this paper, the notation

```
p ::= e
```

stands for

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

The ::= "operator" represents a compile-time declaration. (The Icon structure assignment operator ::= is used for structure assignment in Version 3 of Icon, but is absent from Version 4. That operator is used here because of its familiar connotation of definition.) Some expressions are parameterized with identifiers whose values are passed in calls of the corresponding matching procedures The notation above is naturally extended so that, for example

```
p(x,y)  = e
```

stands for

```
procedure p(x y)
   suspend e
end
```

Finally, local identifiers may be provided so that, for example

```
p(x,y)s,t  = e
```

stands for

```
procedure p(x,y)
   local s,t
   suspend e
end
```

In SNOBOL4, patterns are data objects that are constructed at run time Their status as data objects allows them to be used in computations (the construction of other patterns) and, more importantly, to be transmitted through a program as the value of variables and arguments of functions. Thus a pattern may be used in many *places* as well as at many *times*

Icon procedures are also data objects The important difference between patterns in SNOBOL4 and matching procedures in Icon is that patterns include an additional level of structure In fact, SNOBOL4 patterns contain (or reference) matching procedures [3] These SNOBOL4 matching procedures are internal to the implementation of SNOBOL4 and are inaccessible to the SNOBOL4 programmer The significance of this additional "layer" is discussed in Section 9

3. The Cursor Model

In order to provide matching procedures for the patterns of SNOBOL4, one sufficient protocol is given by the following three conditions

(1) Evaluation of *e* must not result in a change to the value of subject

(2) Evaluation of *e* must leave the value of cursor between its value prior to the evaluation of *e* and the end of subject, inclusive

(3) If *e* fails it must leave cursor unchanged

The nondecreasing aspect of Condition 2 reflects an idiosyncrasy of SNOBOL4 and is in no way essential to the underlying concepts of pattern matching Condition 3 allows *e* to change cursor as specified in Condition 2, but requires cursor to be restored if *e* fails

Note that matching procedures are used only for their side effects — changing the value of the global identifier cursor It lends some uniformity to the model, however, if matching procedures produce a uniform result, even though this result is not used Hence in the following sections, matching procedures also return the value of cursor

This protocol for matching procedures defines the *cursor model* for pattern matching, so called since it involves changing the value of cursor This model corresponds closely to the way that pattern matching is actually implemented in SNOBOL4 [3]

To demonstrate that the protocol described above is sufficient to implement pattern matching requires only the specification of matching procedures for the patterns of SNOBOL4 and a procedure that corresponds to the SNOBOL4 pattern-matching statement

It simplifies the coding of matching procedures if Condition 3 is incorporated in the shell of matching procedures Thus in this model

```
p ::= e
```

stands for

```
procedure p()
   local tcursor
   tcursor := cursor
   suspend e
   cursor := tcursor
end
```

Thus *e* itself need not restore cursor.

The following sections detail the matching procedures for the cursor model. The prefix c_ is used to distinguish procedures in the cursor model. Upper-case letters are used for SNOBOL4 identifiers, while lower-case letters are used for corresponding identifiers in Icon.

3.1 Positional Patterns

Positional patterns are those that change or test the value of the cursor without regard for the actual characters in the subject. There are six kinds of positional patterns: five that are constructed by the pattern-valued functions LEN(I), TAB(I), POS(I), RTAB(I), and RPOS(I) as well as the built-in pattern ARB.

LEN(I) adds I to the value of the cursor, provided that I is positive and that the resulting value is not greater that the length of the subject. A corresponding matching procedure is

```
c_len(i) ::= if 0 <= i <= *subject+1-cursor then cursor := cursor+i
```

(In Version 4 of Icon, *s is the length of s and if *e1* then *e2* fails if *e1* fails.) Note that it is possible to set the cursor at the end of the subject. This reflects the fact that cursor positions in SNOBOL4 (and Icon) are between characters. A value of cursor equal to *subject+1 corresponds to a position after the last character of the subject. Note also that the result of c_len(i) is the value of cursor as specified earlier. If i is too great, cursor is not changed and c_len(i) fails. Otherwise cursor is incremented. The restoration of cursor is done by the shell of the matching procedure.

TAB(I) is similar to LEN(I), but sets the cursor to the value of I. The matching procedure is

```
c_tab(i) ::= if cursor-1 <= i <= *subject then cursor := i+1
```

Here the value assigned to cursor is one greater than i, since SNOBOL4 character positions start at 0 but Icon positions start at 1. The argument i to c_tab is in terms of SNOBOL4 indexing, while the value assigned to cursor is in terms of Icon indexing (the position 0 in Icon is a specification for the position after the end of a string). It is important that the value actually assigned to cursor be in terms of Icon indexing to avoid complicating subsequent procedures.

POS(I) tests that the value of the cursor is equal to I and has the matching procedure

```
c_pos(i) ::= (cursor = i+1)
```

RTAB(I) and RPOS(I) are similar to TAB(I) and RPOS(I), except that positions are determined relative to the right end of the subject. Their matching procedures are included in Appendix A.

ARB is a built-in pattern that has "alternatives". It first does nothing. However, if the context in which it is used requires an alternative action, it increments the cursor by one, then by two, and so on. For example, the pattern

```
ABPAT = "A" ARB "B"
```

matches any string that contains an A followed (not necessarily immediately) by a B. Once an A is matched, ARB increments the cursor by 0, 1, 2, ... as needed until a B is reached.

The matching procedure corresponding to ARB uses a simple Icon generator:

```
c_arb ::= (cursor := (cursor to *subject+1))
```

The standard Icon syntax makes the operation of the matching procedure clearer:

```
procedure c_arb()
   local tcursor
   tcursor := cursor
   suspend (cursor := (cursor to *subject+1))
   cursor := tcursor
end
```

Each time c_arb() is reactivated, cursor is incremented by one.

3.2 Lexical Patterns

Lexical patterns are those that change the cursor depending on the characters contained in the subject. There are seven kinds of lexical patterns: matches of specific strings, five kinds that are constructed by the pattern-valued functions ANY(S), NOTANY(S), SPAN(S), BREAKX(S), and BREAK(S) as well as the built-in pattern BAL. The matching for these patterns is analogous to the lexical functions in Icon.

In SNOBOL4, the match of a specific string is indicated by simply including the string in the pattern — there is no visible syntax. Patterns for string matches are constructed as a byproduct of automatic type coercion. For example, in the pattern

```
ABPAT = "A" ARB "B"
```

The literals "A" and "B" are coerced into patterns that match the strings A and B.

The corresponding matching procedure for string matches is

```
c_match(s) ::= (cursor := match(s,subject,cursor))
```

ANY(S), which adds one to the cursor provided the character following the cursor is contained in the string S, has the matching procedure

```
c_any(s) ::= (cursor := any(s,subject,cursor))
```

NOTANY(S) is the converse of ANY(S), adding one to the cursor if the character following the cursor is not contained in the string S:

```
c_notany(s) ::= (cursor := any(~s,subject,cursor))
```

SPAN(S) is similar to ANY(S), but increments the cursor by the number of consecutive characters in S that occur following the cursor. Its matching procedure is

```
c_span(s) ::= (cursor := many(s,subject,cursor))
```

BREAKX(S) increments the cursor by the number of characters following the cursor up to a character in S. Its matching procedure is

```
c_breakx(s) ::= (cursor := upto(s,subject,cursor))
```

BREAKX(S), which is an extension to SNOBOL4 first introduced in SPITBOL [4], has alternatives like ARB and may set the cursor to alternative positions if required by the context in which it is used. For example, the pattern

```
LOCTHE = BREAKX("t") "th"
```

sets the cursor to the position of the first t followed by an h, even if there are previous ts in the subject, such as

```
two tanks rammed the wall
```

in which LOCTHE would set the cursor to the third t after setting it to the first and second, only to have th fail to match in those cases.

Since upto is an analogous generator in Icon, the matching procedure given above behaves similarly. The standard SNOBOL4 pattern BREAK(S), however, does not set the cursor to alternative values. Its matching procedure therefore requires that upto be limited to a single result:

```
c_break(s) ::= (cursor := upto(s,subject,cursor) \ 1)
```

(In Version 4 of Icon, *e1* \ *e2* limits *e1* to at most *e2* results.)

The built-in pattern BAL sets the cursor to the position after a string that is balanced with respect to parentheses. An idiosyncrasy of BAL is that it must increment the cursor by at least one. This idiosyncrasy makes its formulation in Icon more complex than it would be otherwise. (A function BAL(S) that matched a balanced string up to a character in S would be a constrained form of BREAKX(S), just as bal(s) is a constrained form of upto(s) in Icon.) Here it is convenient to start with a more elementary procedure analogous to c_upto:

```
c_balu ::= (cursor := bal(")",,,subject,cursor))
```

and an auxiliary procedure that embodies the idiosyncrasies of BAL:

```
c_bbal ::= (c_match("(") & c_balu() & c_len(1)) | c_notany("()")
```

Then

```
c_bal ::= c_bbal() & c_arbno(c_bbal)
```

c_arbno applies c_bbal an arbitrary number of times and is given in the next section. Note the use of the previously defined matching procedures c_match, c_len, and c_notany.

3.3 Applicative Patterns

The SNOBOL4 operation P1 P2 constructs a pattern that first applies P1 and then applies P2. The corresponding matching procedure is

```
c_cat(p1,p2) ::= p1() & p2()
```

Note that p1 and p2 are procedure-valued arguments and that these procedures are invoked in the procedure c_cat. The standard Icon syntax makes the situation clearer:

```
procedure c_cat(p1,p2)
   local tcursor
   tcursor := cursor
   suspend p1() & p2()
   cursor := tcursor
end
```

The conjunction operator, &, is used to provide mutual goal-directed evaluation between invocation of the matching procedures p1 and p2, and provides the backtracking that occurs in pattern matching in SNOBOL4. Since p1 and p2 are matching procedures, saving and restoring cursor in the procedure above is redundant.

The SNOBOL4 operation P1 | P2 constructs a pattern that matches P1 but then matches P2 if P1 fails to match in context. The corresponding matching procedure is

```
c_alt(p1,p2) ::= p1() | p2()
```

The SNOBOL4 function ARBNO(P) matches P repeatedly, but as few times as are needed in context. Its matching procedure is recursive:

```
c_arbno(p) ::= cursor | (p() & c_arbno(p))
```

The first operand of the alternation could be any value, since ARBNO(P) first does nothing (i.e. it matches P zero times). The value of cursor is provided to conform to the uniform value constraint given earlier.

3.4 Assignment

There are three patterns that assign values to identifiers during pattern matching: assignment of the cursor value and immediate and deferred assignment of "matched substrings". Since Icon has neither call-by-reference nor pointers, these patterns cannot be represented by matching procedures in which the identifier is an argument. However, there are corresponding matching expressions.

Assignment of the cursor value, @V, is simple:

```
v := cursor
```

(In cases where patterns involve assignments and hence cannot be represented by matching procedures, the corresponding matching expressions can simply be substituted where calls to matching procedures would otherwise occur.)

The term "matched substring" refers to the substring of the subject between the values of the cursor before and after a pattern is applied. Thus in immediate substring assignment, P $ V, the value assigned to V is the substring of the subject between the values of the cursor before and after P matches. The corresponding matching expression is

```
v := substr(p)
```

where substr is a procedure that returns the substring matched by p and is defined as follows:

```
procedure substr(p)
   local tcursor
   return (tcursor := cursor) & p() & subject[tcursor:cursor]
end
```

This procedure may be coded more compactly by the use of somewhat arcane techniques:

```
procedure substr(p)
   return subject[.cursor:(p() & cursor)]
end
```

Here cursor must be explicitly dereferenced (.cursor), since Versions 3 and 4 of Icon do not dereference identifiers in argument lists until all the arguments are evaluated. The conjunction in the second part of the range specification produces the value of cursor after p() is invoked, where p is a matching expression corresponding to the pattern P. In order to conform to the uniform result constraint, the matching expression that performs substring assignment must be augmented so as to produce the value of cursor:

```
(v := substr(p)) & cursor
```

Conditional substring assignment can be approximated in the model here by the use of reversible assignment:

```
(v <- substr(p)) & cursor
```

The deferred aspect of conditional substring assignment cannot be incorporated in this model because Icon lacks pointers. In any event, the way conditional substring assignment works in SNOBOL4 is an idiosyncratic carry-over from earlier versions of SNOBOL [5].

3.5 Applying Patterns

Patterns in SNOBOL4 are applied in a pattern-matching statement that has the form

```
S ? P
```

(The explicit pattern-matching operator of SPITBOL is used for clarity here as opposed to the implicit operator in standard SNOBOL4.)

In such a statement, S provides the subject on which P is matched. The statement may either succeed (P matches) or fail (P does not match), although values may also be assigned to variables as side effects of the matching process.

A simple Icon procedure that corresponds to a pattern-matching statement is

```
procedure c_apply(s,p)
   subject := s
   cursor := 1
   return p()
end
```

The first two assignment expressions establish the subject and the initial cursor value. Then the procedure p, passed by value to c_apply, is invoked. If p() succeeds, the pattern match is successful and c_apply succeeds. If p() fails, c_apply fails (return fails if evaluation of its argument fails).

SNOBOL4 has two modes of matching: anchored and unanchored. In the anchored mode, the pattern must match beginning at the first character of the subject. In the unanchored mode, the pattern may match anywhere in the subject; a match is first attempted beginning at the first character. If this fails, a match is attempted at the second character, and so on. Unanchored matching is equivalent to placing ARB at the beginning of the pattern.

These two pattern-matching modes can be incorporated in the model here by defining two matching procedures:

```
c_anchor ::= cursor
```

and

```
c_float ::= (cursor := (1 to *subject+1))
```

or simply

```
c_float := c_arb
```

The pattern matching procedure above then can be generalized as follows:

```
procedure c_apply(s,p)
   subject := s
   cursor := 1
   return (c_mode() & p())
end
```

where

```
c_mode := c_anchor
```

establishes the anchored mode and

```
c_mode := c_float
```

establishes the unanchored mode. The procedure c_anchor returns cursor to conform to the uniform result constraint established earlier. The significant aspect of c_anchor is that it does not allow the cursor to be changed as c_float does.

The pattern-matching *expression* S ? P in SITBOL [6] returns the substring matched by P as opposed to just succeeding or failing as in the pattern-matching statement in standard SNOBOL4. This extension is easy to accommodate:

```
procedure c_apply(s,p)
   subject := s
   cursor := 1
   return (c_mode() & substr(p))
end
```

One further SNOBOL4 statement remains — the replacement statement, which has the form

```
S1 ? P = S2
```

in which the subject S1 is matched by the pattern P and if P matches, the substring it matches is replaced by S2, thus modifying S1.

Frequently the replacement string is the result of evaluating an expression that contains identifiers to which values are assigned during pattern matching. An example is

```
S ? BREAK("t") . TPART = "[" TPART "]"
```

Replacement is essentially the concatenation of three strings: (1) the *head* portion of the subject prior to where the pattern matches, (2) the replacement string, and (3) the *tail* portion of the subject following the substring matched by the pattern.

Given global identifiers head and tail, a procedure c_repl can be written to assign values to the head and tail portions:

```
procedure c_repl(s,p)
   local mid
   subject := s
   cursor := 1
   return {
      (head := substr(c_mode)) &
      (mid := substr(p)) &
      (tail := subject[cursor:0]) &
      mid
      }
end
```

The auxiliary identifier mid is the last element in the conjunction and provides the value matched by p() as the value returned by c_repl.

Then the SNOBOL4 replacement statement

S ? P = *S*

where *S* is, in general, a string-valued expression, is represented in the cursor model as

s := head || (c_repl(p,s) & *s*) || tail

Note that head and tail are not dereferenced until after c_apply is evaluated. Note also that the value returned by c_repl is discarded, since its conjunction with *s* produces only the result of evaluating *s*.

4. The Substring Model

The cursor model of pattern matching described in Section 3 concentrates on the minimum facilities needed to implement SNOBOL4-style pattern matching. That is, changing cursor position is the basic result of applying most patterns. The "substring matched" by a pattern is only used in value assignment and the SITBOL-style pattern matching expression. For these cases, the substring matched can be obtained by separate mechanisms as it is, in fact, in actual implementations of SNOBOL4.

In an earlier report [1], however, a more powerful model was used in which matching expressions returned the substring matched in addition to changing the cursor. While this *substring model* is more powerful (and inefficient) than is necessary to implement SNOBOL4, it provides the basis for a number of extensions that are not possible in the cursor model as well as even more powerful generalizations.

In the substring model, the following condition is added:

(4) If *e* succeeds, the value it returns must be the substring of subject between the values of cursor before and after the evaluation of *e*.

This additional condition requires revisions in the details of matching procedures given in Section 3, but it offers no substantial difficulties. Some examples of the changes required are given below. The prefix s_ is used to distinguish procedures in the substring model. A complete list of matching procedures for the substring model is given in Appendix B.

4.1 Positional Patterns

The heart of the changes lies in the positional patterns. LEN(I) and TAB(I) have matching procedures

```
s_len(i) ::= if 0 <= i then subject[.cursor:cursor := cursor+i]
```

and

```
s_tab(i) ::= if cursor-1 <= i then subject[.cursor:cursor := i+1]
```

The range tests are included only to assure SNOBOL4 constraints are met and to avoid incorrect results due to the different indexing schemes in SNOBOL4 and Icon. As mentioned earlier cursor must be dereferenced in the range specifications. In each case, the substring matched is the result. Note that the range tests are simplified, since Icon substring specifications perform range checks.

4.2 Lexical Patterns

The matching procedures for lexical patterns are similarly straightforward adaptation of those of the cursor model. For example, the matching procedure for BREAKX(S) in the substring model is

```
s_breakx(s) ::= subject[.cursor:cursor := upto(s,subject,cursor)]
```

4.3 Applicative Patterns

Patterns that apply patterns are the same in the cursor and substring models, except for the use of concatenation, not conjunction, since the substring matched by concatenated patterns is the concatenation of the substrings matched by the patterns. Thus

```
s_cat(p1,p2) ::= p1() || p2()
```

and

```
s_arbno(p) ::= "" | (p() || s_arbno(p))
```

Note that s_arbno first returns the empty string — the substring matched by zero applications of p.

4.4 Assignment

Immediate and conditional substring assignment are straightforward in the substring model:

```
v := p()
```

and

```
v <- p()
```

Cursor position assignment in the substring model requires the kind of augmentation that immediate and conditional substring assignment require in the cursor model:

```
(v := cursor) & ""
```

4.5 Applying Patterns

In the substring model, the procedure for applying patterns is similar to that for the cursor model:

```
procedure s_apply(s,p)
   subject := s
   cursor := 1
   return (s_mode() & p())
end
```

where s_mode is either s_anchor or s_float as before, with

```
s_anchor ::= ""
```

and

```
s_float ::= subject[1:(cursor := (1 to *subject+1))]
```

Since p() returns the substring matched, this procedure also serves for the SITBOL pattern-matching expression.

The procedure used for replacement is also similar to that used in the cursor model. See Appendix B.

5. A Generalized String Model

In the substring model, matching procedures return the substring they match. A natural consequence of this model is that an entire pattern matches the substrings of its (concatenated) components.

The substring matched is frequently useful, but it is unnecessarily restrictive. Doyle earlier proposed extending pattern matching to allow, among other things, contributions other than matched substrings as well as the suppression of unwanted substrings [7]. In this model, pattern matching is a process in which analysis and synthesis are concomitant, and they produce a result that may or may not contain components matched in the subject.

Such extensions are easily added to the substring model (but not the cursor model). Condition 4 can simply be relaxed as follows.

(4*) The value returned by *e* may be any string.

Condition 4* defines the *generalized string model* of pattern matching. The prefix g_ is used to distinguish procedures in this model.

5.1 Concatenation

In the substring model, Condition 4 requires the value returned by a matching expression to be the substring of the subject between the values of the cursor before and after the expression is evaluated. Hence the value returned by the concatenation of two patterns is required to be the concatenation of the substrings matched by the two patterns. In the generalized string model, Condition 4* imposes no such requirement. It is, however, most natural to define the concatenation of patterns to produce the concatenation of their values:

```
g_cat(p1,p2) ::= p1() || p2()
```

In other words

```
g_cat := s_cat
```

Similarly

```
g_arbno := s_arbno
```

since ARBNO(P) is effectively concatenation an indefinite number of times. It also follows that

```
g_apply := s_apply
```

5.2 Indigenous and Exogenous Contributions

Doyle used the term "indigenous contribution" to refer to matched substrings and the term "exogenous contribution" to refer to strings not obtained by matching the subject.

Indigenous contribution is essentially defined by the substring model and all s_ matching procedures make indigenous contributions (some of them, such as s_pos(i), contribute the empty string).

Doyle's pattern /P matches P but produces the empty string (i.e., the value produced by P is discarded). A corresponding matching procedure is

```
g_discard(p)  = p() & ""
```

Doyle's pattern \S produces S, subject is ignored and cursor is not changed A corresponding matching procedure is

```
g_exog(s)  = s
```

Doyle's replacement pattern P ≡ S matches P but produces S (it is equivalent to the concatenation /P \S). A corresponding matching procedure is

```
g_replace(p,s)  = p() & s
```

5.3 Transformational Synthesis

Indigenous and exogenous contribution are merely special cases of a more general operation, *transformational synthesis*, in which the matched substring is transformed by some procedure The matching procedure is

```
g_xform(p1,p2)  = p2(p1())
```

where p2 is some (string-valued) procedure that is applied to the result of evaluating p1() A simple example of transformational synthesis is

```
rev(p)  = g_xform(p,reverse)
```

which produces the reversal of the string produced by p

The transforming procedure need not, of course, be built in For example,

```
procedure period(s)
   local c, t
   t = ""
   every c - !s do t ||= c || "."
   return t
end
```

when used in

```
dform(p)  = g_xform(p,period)
```

inserts periods after every character of the string produced by p

6. Extensions to Pattern Matching

The cursor and substring models are two models for SNOBOL4-style pattern matching The generalized string model allows extensions to the SNOBOL4 repertoire, primarily in the area of synthesis In any of these models, there are various other extensions that are difficult or impossible to formulate in the framework of SNOBOL4 pattern matching but that would nonetheless significantly enhance it The extensions that follow are based on the substring and generalized string models, although several apply, with obvious modifications, to the cursor model The prefix x_ is used to distinguish procedures for such extensions

6.1 Generalizations of Existing SNOBOL4 Patterns

SNOBOL4 requires that all patterns change the cursor in a nondecreasing fashion Thus in LEN(I), I must be nonnegative and in TAB(I), I must be greater than or equal to the value of the cursor These restrictions are unnecessary Furthermore the definition of "substring matched" need not be changed — it can still refer to the substring of the subject between the cursor before and after the pattern is applied Condition 2 must be revised as follows to allow the value of the cursor to be decreased

(2*) Evaluation of *e* must leave the value of cursor between 0 and the end of subject, inclusive

For LEN(I) and TAB(I), the procedures are actually somewhat simpler than before

```
x_len(i) ::= subject[.cursor:cursor := cursor+i]
x_tab(i) ::= if 0 <= i then subject[.cursor:cursor := i+1]
```

The range test in x_tab(i) prevents negative position specifications from being incorrectly interpreted.

A "contracting pattern" similar to ARB, but starting by setting the cursor to end of the subject and working backward would often be useful:

```
x_marb ::= (cursor := (*subject+1 to cursor by -1))
```

(The arguments of to are evaluated, only once, prior to generation of the sequence.)

Several SNOBOL4 patterns can be generalized in a useful manner simply by parameterizing them. Thus ARB can be viewed as a special case of ARB(I), which increments the cursor by successive values of I.

The matching procedure is

```
x_arb(i) ::= (cursor := (cursor to *subject+1 by i))
```

In a further generalization, a negative value of I could be interpreted as decreasing the cursor and a zero value of I leaving the cursor unchanged (equivalent to the infrequently used pattern SUCCEED). The matching procedure for this interpretation is more complicated but basically straightforward.

Another example of a useful parameterization would be BAL(S1,S2), where S1 and S2 specify strings of characters with respect to which the balancing is determined. This generalization is incorporated in Icon's lexical analysis function bal.

6.2 Limiting Goal-Directed Evaluation

One of the most insidious hazards in pattern matching is the unnecessary search for alternatives, which may assume exponential proportions. This usually occurs when the application of a pattern is destined to fail. In fact, backtracking during pattern matching was disallowed in COMIT because of the fear of the inefficiencies it would introduce [8]. In practice, unnecessary backtracking in SNOBOL4 does not produce catastrophic effects. Its effect on program performance is generally unknown, however, and the programmer has no way of observing it, since pattern matching procedures cannot be traced in SNOBOL4. (The problem is painfully evident when SNOBOL4 patterns are translated into Icon, whose matching procedures can be traced.)

If unnecessary backtracking is known or suspected to be a possibility, the pattern FENCE can be used to abort pattern matching on backtracking. In practice, FENCE is used infrequently, partly because its potential value is imponderable. Furthermore, the effects of FENCE are drastic — it is not possible to inhibit backtracking at one site without aborting pattern matching altogether.

A more attractive approach is to limit the goal-directed evaluation in a specific pattern.

```
x_limit(p,i) ::= (p() \ i)
```

In use i normally would be 1, although the generalization does no harm.

Using limit, BREAK(S) is subsumed by BREAKX(S):

```
x_break(s) ::= x_limit(s_breakx,1)
```

6.3 Pattern Location

The unanchored mode of pattern matching in SNOBOL4 provides a convenient, if limited, means of locating the first position at which a pattern matches. This can be cast in a more general way by

```
x_locate(p) ::= (cursor := (cursor to *subject+1)) & p()
```

Note that the substring matched by locate in the substring model is the substring matched by p, while in the cursor model it includes the substring starting at the original value of cursor. This discrepancy can be eliminated by using concatenation instead of conjunction, but the substring matched by p probably is the more desirable choice.

Location of a string occurs frequently, as evidenced by the common occurrence of SPITBOL patterns such as

```
    BEGIN = BREAKX("b") "begin"
```

A specific pattern for this purpose, based on Icon's lexical analysis function find is worthwhile:

```
    x_find(s) ::= s_tab(find(s,subject,cursor)+1)
```

Note that the substring matched by x_find(s) is the substring up to s.

6.4 Nested Pattern Matching

As mentioned earlier, pattern matching in SITBOL is an expression, not a statement. Thus it is possible to write

```
    (S ? P1) ? P2
```

This expression first applies P1 to S. If that match succeeds, P2 is then applied to substring matched by S ? P1. If P2 fails, however, an alternative match is not attempted for P1.

In order to allow mutual goal-directed evaluation in pattern-matching expressions, significant modifications must be made to the application procedure. The procedure that follows is named x_apply to indicate that it is an extension to standard SNOBOL4 pattern matching. The procedure given here is based on the substring model; it can be adapted easily to the cursor model.

In the definitions for c_apply and s_apply given earlier, subject and cursor are simply set on entry. In order for nested pattern matching to work properly, x_apply must restore subject and cursor to their former values when it returns. In addition, x_apply itself must be a matching procedure, capable of performing a sequence of matches. Such a procedure follows.

```
    procedure x_apply(s,p)
       local tsubject, tcursor, value
       suspend {
          (tsubject := subject) &
          (tcursor := cursor) &
          (subject <- s) &
          (cursor <- 1) &
          (mode() & (value := p())) &
          (subject <- tsubject) &
          (cursor <- tcursor) &
          value
          }
    end
```

The heart of x_apply is a suspend whose argument is a complex conjunction of assignment expressions. tsubject and tcursor first are used to save the values of subject and cursor so that they can be restored on exit from the procedure. subject and cursor are then set to their initial values. Reversible assignment is used so that their prior values are restored should p() fail. If p() succeeds, its value is saved and subject and cursor are restored to their values prior to invocation of x_apply (note that p() may change cursor but not subject under Condition 1 of the matching expression protocol). The final expression in the conjunction is value, which is the result returned by x_apply.

Should x_apply be reactivated for an alternative match for p(), the expressions in the conjunction are reactivated in reverse order. (In Version 4 of Icon, enclosing braces do not inhibit goal-directed evaluation.) Thus subject and cursor are restored to the values they had when p() was previously activated. p() is then activated again. Should p() succeed, x_apply proceeds as before, suspending with the new value. Should p() fail, subject and cursor are restored as before and x_apply fails.

Note that the argument of suspend is a matching expression in either model: although subject may be changed temporarily during its evaluation, it is restored before completion. Similarly x_apply is a matching procedure:

```
x_apply(s,p)tsubject,value ::= {
   (tsubject := subject) &
   (subject <- s) &
   (cursor <- 1) &
   (mode() & (value := p())) &
   (subject <- tsubject) &
   (cursor <- tcursor) &
   value
   }
```

Since x_apply is a matching procedure, it corresponds to a pattern and can be used like any other matching procedure. Note that tcursor is local by virtue of its inclusion in the "shell" of all matching procedures.

7. Limitations of the Models

None of the procedural models described in the preceding sections can handle all the features of pattern matching in SNOBOL4. This section discusses the limitations of the procedural models and assesses their importance.

7.1 Binding Time

Patterns in SNOBOL4 are constructed at run time, while matching procedures are defined at compile time. Consequently identifiers in matching procedures are bound when the procedures are invoked, while identifiers in patterns are normally bound when the patterns are constructed. This early binding time in patterns can be avoided by use of the unevaluated expression operator, which is, in fact, necessary to obtain recursive references and defer evaluation of expressions until patterns are applied. Recursion in Icon matching procedures of course follows naturally.

In most cases, all identifiers in SNOBOL4 patterns could be unevaluated and the patterns would produce the same results. This is usually not done because extra notation is required, not because the early binding is needed. Essential use of early binding is very rare in SNOBOL4 programs. In fact, a number of SNOBOL4 programs containing many complex patterns have been translated into Icon programs with matching procedures such as those given in this report with no instance of a problem due to the difference in binding times.

Since binding time is not optional in matching procedures, but is a concern and problem in patterns, matching procedures may, in fact, offer an advantage over patterns in this regard.

7.2 Conditional Substring Assignment

The deferred aspect of conditional substring assignment, which cannot be handled in the models, was an efficiency consideration, since the original implementation of SNOBOL4 creates physically distinct copies of every string created [3]. Since many tentative substrings may be created during pattern matching, a list of names and positions is kept so that strings are actually formed only when the pattern matching process is successfully completed. Originally this technique, as opposed to immediate but reversible assignment, was largely transparent to executing programs. As features were added, the situation became clouded. Very few SNOBOL4 programmers know, for example, what happens in a statement such as

```
S ? P . *A[I]
```

When is the unevaluated expression *A[I] evaluated? What happens if it fails because I is out of range?

Immediate substring assignment was introduced so that the values of substrings matched could be used subsequently in the same pattern match. Quixotically, immediate substring assignment is faster than conditional substring assignment in some implementations of SNOBOL4 [4,6].

In any event, the idiosyncrasies of conditional substring assignment are a consequence of implementation considerations, and there is little evidence that they offer any linguistic advantages over reversible assignment.

7.3 Aborting Pattern Matching

SNOBOL4 has two built-in patterns, ABORT and FENCE, that abort pattern matching altogether, causing the entire pattern match to fail ABORT causes failure when encountered in matching, while FENCE causes failure during backtracking (ABORT and FENCE can be formulated in terms of each other in conjunction with the built-in patterns FAIL, which simply fails to match and NULL, which always matches; there is only one concept involved FAIL and NULL of course can be represented easily by matching procedures)

The ability to abort a pattern match at an arbitrary point is not easy to add to the procedural models. It is possible, however, by addition of a distinctive "aborting" value, with a check for this value in all matching procedures that invoke other matching procedures This technique is actually employed in the original implementation of SNOBOL4 [3] It is, however, cumbersome and would greatly complicate the models given here. An alternative is adding more power to the procedural model In SL5 [9], for example, a pattern match could be aborted by resumption of the environment that serves as the root of the matching process. The semantics needed to implement a procedural model of pattern matching are discussed in Section 8. On the other hand, while the ability to abort pattern matching (even successfully [7]) is intellectually appealing, most actual uses of such a facility can be accomplished other ways

7.4 Pattern-Matching Heuristics

Different dialects of SNOBOL4 employ various heuristics to avoid "futile" attempts to match and hence to speed the process [4,6] Heuristics are also used to avoid left-recursive plunges during the matching process

When the heuristics were first conceived, they were motivated entirely by concerns of efficiency and were transparent to the running program except as they affected the time required for program execution. As language features were added, they became linguistically significant, since components of a pattern that might have side effects might not be applied at all as a consequence of the heuristics

In retrospect, the heuristics appear to be a language design mistake Most programs actually run faster without use of the heuristics (checking takes longer than the "futile" matching that is saved) Few if any SNOBOL4 programmers fully understand the heuristics The Macro SPITBOL dialect of SNOBOL4 [10], which does not implement any heuristics, is used without apparent difficulty, even with respect to left recursion

Since the heuristics are neither well defined nor consistently interpreted in different implementations, nor apparently necessary or useful, the fact that they cannot be incorporated easily in the procedural models is of little consequence

8. Facilities Required to Implement Pattern Matching

The pattern-matching facilities described above are given in terms of Icon expressions and procedures. That is to say, the facilities of Icon are adequate to implement such pattern-matching facilities with the exceptions discussed in the preceding section Not all of the facilities of Icon are needed, however

Before going on to consider a possible new pattern-matching facility, it is worth examining the minimum set of features that are needed

(1) Goal-directed evaluation

(2) Reversible assignment

(3) The alternation control structure

(4) Procedures that are data objects

(5) A procedure mechanism capable of suspending activation with the return of a value and subsequent reactivation

(6) Elementary string operations comparison, concatenation, and substring specification

Goal-directed evaluation underlies much of the pattern-matching mechanism and is essential to the proper functioning of the models Furthermore, it must be a general aspect of expression evaluation

Reversible assignment is essential to the limited form of data backtracking needed (more general data backtracking could be used, but is unnecessary).

Alternation is a very important control structure. Unlike conjunction, which is an essentially vacuous operation that facilitates mutual goal-directed evaluation among expressions, alternation controls the order in which the results of generators are produced.

Procedures must be data objects so that they can be passed as arguments to other procedures and invoked at the proper time.

The suspend mechanism in Icon procedures is about the minimum procedural facility that is needed. A more general coroutine mechanism with the proper characteristics would also do, but that generality is not necessary. Note that all built-in Icon generators, such as to-by, can be written as Icon procedures.

Elementary operations on strings are obviously needed in a string pattern-matching facility. Icon offers a wide variety of such features as well as syntactic devices, such as range specifications, that make string processing concise and easy to formulate. All of these features can be composed from more elementary operations, however.

9. Design of a New Pattern-Matching Facility

There is ample evidence that pattern matching in the style of SNOBOL4 is a valuable programming language facility. Some of the faults of pattern matching in SNOBOL4 can be attributed to idiosyncrasies and specific design flaws. Other faults are a consequence of the basic structure of SNOBOL4 on which the pattern-matching facilities are built. This section discusses a possible new pattern-matching facility that could be incorporated into a more suitable base language.

9.1 Adequacy of a Procedural Model

The procedural models described in Sections 3 through 5 have the capability of representing most of the features of pattern matching in SNOBOL4. None of the limitations described in Section 7, with the possible exception of the ability to abort pattern matching at an arbitrary point, are significant. In fact, the procedural models are, if anything, more naturally consistent and uniform than SNOBOL4's pattern-matching facility. Furthermore, the procedural models offer clear opportunities for extensions, unifications, and generalizations that are needed in any pattern-matching facility that would be an improvement over SNOBOL4.

9.2 Choice of a Procedural Model

Of the three models presented earlier, the generalized string model offers the greatest possibilities. While the production of matched substrings is inherently more expensive in time and space than the change of the cursor alone, this problem can be reduced by lazy evaluation and other clever implementation techniques. In any event, the use of pattern matching is often motivated more by a need for problem solving power than by execution efficiency. Consequently the proposal here is based on the most general of the three models.

The protocol in the generalized string model, revised to include possible extensions discussed in Section 6, is:

(1) Evaluation of *e* must not result in a change to the value of subject.

(2) Evaluation of *e* must leave the value of cursor between 0 and the end of subject, inclusive.

(3) If *e* fails, it must not leave cursor changed.

(4) If *e* succeeds, the value it returns may be any string.

This model imposes few restrictions on matching expressions — it even allows them to change subject temporarily. Condition 1 could be lifted entirely, although this raises a number of sticky issues. Condition 3 also could be lifted, but experience and a little experimentation suggest that it serves a valuable purpose in imposing structure on pattern matching.

9.3 Design Criteria

The many and conflicting considerations in programming language design are well known and extensively discussed in the literature. In the limited scope of this paper, the considerations are confined to the area of pattern matching. In the final say, decisions among competing factors are made on the basis of philosophy, taste, and intuition. The following design criteria are suggested:

(1) The pattern-matching facility should be integrated into the base language. This integration will avoid the proliferation of similar but distinct facilities (four types of assignment in SNOBOL4 for example) and will avoid the linguistic schism that plagues SNOBOL4 [11]

(2) A capability for programmer-defined matching procedures is essential. The lack of such a facility in SNOBOL4 is its most significant weakness and contributes to the large size and idiosyncratic nature of its pattern-matching facilities.

(3) The cursor and subject should be accessible to the programmer so as not to limit the power of defined matching procedures. Specifically, it should be possible to write any built-in matching procedure as a defined matching procedure. The programmer should not have to refer to the cursor and subject to perform basic pattern matching, however.

(4) A set of built-in features should be selected as a compromise between simplicity and conciseness on the one hand and facility as evidenced by need on the other.

This report does not attempt to propose a final design. However, suggestions for built-in features, which may serve as a point of discussion, are given in the next section. No prefix is used to distinguish matching procedures corresponding to these facilities. New mnemonics are introduced to improve terminology, but analogies to earlier features should be clear.

9.4 Proposed Built-In Features

Much of Icon's model for string scanning provides a good basis for the proposed new facility. Specifically, its system for specifying character positions, which allows nonpositive specification relative to the right end of the string, is superior to SNOBOL4's system for specifying character positions.

Icon's move(i) and tab(i), equivalent to x_len(i) and x_tab(i) adapted to Icon's character position specification, are proven matching procedures: both relative and absolute change of cursor position are commonly needed.

Icon's underlying lexical functions seem well chosen and provide the basis for a set of matching procedures:

```
taba(s) ::= subject[.cursor,cursor := any(s,subject,cursor)]
tabb(s1,s2,s3) ::= subject[.cursor,cursor := bal(s1,s2,s3,subject,cursor)]
tabf(s) ::= subject[.cursor,cursor := find(s,subject,cursor)]
tabm(s) ::= subject[.cursor,cursor := many(s,subject,cursor)]
tabs(s) ::= subject[.cursor,cursor := match(s,subject,cursor)]
tabu(s) ::= subject[.cursor,cursor := upto(s,subject,cursor)]
```

(Better names are needed. The natural choices, those of Icon's lexical analysis functions, are not used here to avoid confusion.)

In addition, testing the value of cursor is often useful:

```
pos(i) ::= (cursor = poseq(i)) & ""
```

where poseq(i) returns the positive equivalent of i:

```
procedure poseq(i)
   if i < 1 then i := *subject+1+i
   if 1 <= i <= *subject+1 then return i else fail
end
```

A new matching procedure that seems useful simply checks for the existence of a substring in subject.

```
    look(s) ::= find(s,subject,cursor) & ""
```

look(s) can be used to avoid much unnecessary processing.

9.5 Icon as a Base Language

Since Icon is evidently adequate for implementing a pattern-matching facility at the source level, it is natural to consider Icon as a base language for a built-in pattern-matching facility.

Icon's major advantages for such a base language are its goal-directed evaluation and related control structures. These exist in no other programming language in such a general form and are specially suited to needs of pattern matching. Icon also has an adequate procedure mechanism (although a call-by-reference facility, or even something more powerful would be helpful).

Icon is, however, already a "large" language, both syntactically and semantically. Adding a pattern-matching facility on top of Icon would produce an overbearing result. Fortunately, there are a number of features in Icon that could be removed to provide a smaller base language.

The entire string-scanning facility of Icon could be replaced by the one described here. Expunging string scanning involves removing the scan-using and transform-using control structures, all scanning functions. One side effect of removing these facilities is the elimination of the somewhat awkward concept of a "scanned substring" to which assignment can be made to change the subject.

A more radical possibility is removing the lexical functions such as find(s) and upto(s). The rationale for this proposal is that such functions are unnecessary and lead to confusing programming techniques that mix high-level and low-level string processing. This change has the additional advantage of decreasing the linguistic size of the language.

9.6 Programmer-Defined Matching Procedures

The existing procedure mechanism of Icon is adequate for programmer-defined matching procedures. There are two issues that need to be addressed, however:

(1) Should there be built-in support for common aspects of matching procedures?

(2) Should the protocol for matching expressions be enforced as part of the matching procedure mechanism?

Some syntactic support for matching procedures is a practical necessity. Much of the form of a matching procedure is stylized. No programmer would want to have to enter standard information repetitiously — just as a more compact notation was used in this paper to avoid constantly repeating the procedural shell that is common to all matching procedures.

Enforcing the protocol on matching expressions comes down to two matters: (1) ensuring that values assigned to the cursor are in the range of the subject, and (2) restoring the cursor when matching procedures fail.

Range enforcement as it is presently done with &pos in Icon seems to be a clean method: assignment to &pos simply fails if the value would be out of the range of &subject. This internal checking avoids many of the explicit tests that a programmer would otherwise have to perform (review c_len(i) and c_tab(i) given in Section 3).

The great advantage that programmer-defined matching procedures provide is the elimination from the built-in repertoire of procedures that are not needed frequently and that can be easily written by the programmer. Most of the matching procedures discussed in Sections 5 and 6 fall into this category. Other examples are given in Section 10.

9.7 Patterns Versus Matching Procedures

In SNOBOL4, LEN(I) is a pattern-valued function, not a pattern. This detail is often overlooked, since patterns are constructed at run time and pattern-valued expressions can be written in-line in pattern-matching statements. Consider a pattern-valued expression such as

```
        P = LEN(3) | (TAB(5) BREAK("x")
```

Each function in this expression constructs a pattern. Two are combined by the concatenation operation which forms another pattern. This pattern, in turn, is combined by the alternation operation to form the final pattern assigned to P. Using matching procedures as described in preceding sections, there would be a procedure declaration for each pattern:

```
        p1 ::= len(3)
        p2 ::= tab(5)
        p3 ::= break("x")
        p4 ::= cat(p2,p3)
        p ::= alt(p1,p4)
```

If this is not clear, recall that cat and alt are called with procedure-valued arguments; these procedures are not invoked until after cat and alt are called. On the other hand

```
        alt(len(3),cat(tab(5),break("x")))
```

would evaluate the procedures prior to invocation of alt and cat — with quite different results!

Fortunately, it is not necessary to encapsulate alternation and concatenation in matching procedures — they can simply be written using the corresponding Icon operations:

```
        p ::= len(3) | (tab(5) || break("x"))
```

This reflects the fundamental difference between patterns and matching procedures. In SNOBOL4, matching procedures are contained in patterns, and they cannot be used directly with other language operations. The direct use of matching procedures eliminates the need for a repertoire of pattern-building and application operations that mimic other operations in the language. Thus the SNOBOL4 substring assignment operation

```
        P $ S
```

builds a pattern containing a matching procedure to perform assignment. With matching procedures, this level is eliminated and is simply

```
        s := p()
```

This is the essential advantage of matching procedures over patterns. In SNOBOL4, operations that are invoked during pattern matching are hidden in the implementation. In the Icon model, however, matching procedures are not contained in patterns, and they can be invoked directly using Icon control structures and operations.

9.8 Applying Matching Procedures

One disadvantage of the application procedures, such as x_apply(s,p), is that p must be a procedure, which requires a separate declaration for even the simplest case. Thus the SNOBOL4 statement

```
        S ? LEN(3)
```

becomes

```
        move3 ::= move(3)
```

and

```
        apply(s,move3)
```

A better solution is a control structure in the style of Icon's scan-using, but perhaps cast in operator syntax

e1 ? *e2*

The example above then becomes

```
    s ? move(3)
```

The main difference between scan *e1* using *e2* and *e1* ? *e2* based on x_apply is that the expressions *e1* and *e2* are restricted to a single result in scan-using, while *e1* ? *e2* allows goal-directed evaluation of *e1* and *e2*.

Another syntactic convenience is "augmented pattern application":

e1 ?:= *e2*

which is equivalent to

e1 := (*e1* ? *e2*)

except that *e1* is only evaluated once. The result of evaluating *e1* must be a variable, of course.

10. Comparison of String Processing Facilities

In this section, solutions of several "typical" string processing problems are given in SNOBOL4, Icon, and the new facility proposed in Section 9. These solutions provide a basis for comparison, discussion, and evaluation.

10.1 Examples

Reformatting Lines

One of the simplest and frequently occurring string processing problems is reformatting lines of data. A simple example is given in Reference 12, pages 42-43. In this example the data consists of a list of congressmen from New Jersey, for which information is formatted in specific columns as shown below.

```
    4                           30   36
    ↓                           ↓    ↓
  1 William T. Cahill           Rep  Collingswood
  2 Thomas C. McGrath, Jr.      Dem  Margate City
  3 James J. Howard             Dem  Wall
                            .
                            .
                            .
 14 Dominick V. Daniels         Dem  Jersey City
 15 Edward J. Patton            Dem  Perth Amboy
```

The problem is to print a new set of data with only the names left justified at column 1 and the address right justified at column 44. The desired output is

```
1                                          44
↓                                           ↓
William T. Cahill                Collingswood
Thomas C. McGrath, Jr.           Margate City
James J. Howard                          Wall
                            .
                            .
                            .
Dominick V. Daniels               Jersey City
Edward J. Patton                  Perth Amboy
```

A SNOBOL4 program, adapted from Reference 12 follows.

```
          &ANCHOR = 1
          INFO = TAB(3) TAB(29) . NAME TAB(35) REM . PO

READ      LINE = INPUT                                       :F(END)
          LINE ? INFO                                        :F(ERROR)
          NAME = TRIM(NAME)
          OUTPUT = NAME DUPL(" ",44 - (SIZE(NAME) + SIZE(PO))) PO
+                                                            :(READ)

END
```

This solution is sufficiently obvious that it needs no discussion.

A straightforward Icon solution follows.

```
procedure main()
   local line, name, po
   while line := read() do {
      scan line using {
        tab(4)
        name := tab(30)
        tab(36)
        po := tab(0)
        }
      write(left(trim(name),30),right(po,14))
      }
end
```

Again no explanation is needed — the correspondence to the SNOBOL4 solution is evident.

A solution using the new facility is:

```
info ::= write(tab(4) & left(trim(tab(30)),30),(tab(36) & right(tab(0),14)))

procedure main()
   local line
   while line := read() do
      (line ? info())
end
```

This solution is a bit more "ingenious" and takes advantage of the ability to use the entire function repertoire during pattern matching. Note that no auxiliary identifiers are required. This is, however, not a feature of the new facility, *per se*. An alternative Icon solution, motivated by this technique, is

```
procedure main()
   local line
   while line := read() do
      scan line using
         write(
            tab(4) & left(trim(tab(30)),30),
            tab(36) & right(tab(0),14)
            )
end
```

This problem is simple enough that low-level string processing is really more appropriate. Such a solution in Icon provides a contrast with the high-level solutions:

```
procedure main()
   local line
   while line := read() do
      write(left(line[4:30],30),right(line[36:0],4))
end
```

Line Justification

Another standard example is line justification — adding blanks between words in paragraphed text to get an even right margin. There are many approaches to this problem. One is discussed in Reference 13, pages 176-178, from which the following SNOBOL4 solution is adapted:

```
            &ANCHOR = 1
            BLANKS = BREAK(" ") . HEAD SPAN(" ") . SEP
            DEFINE("JUST(JUST,LENGTH)LINE,HEAD,SEP,DIFF")

READ        LINE = INPUT                                 :F(END)
            OUTPUT = JUST(LINE,60)                       :(READ)

JUST        DIFF = LENGTH - SIZE(JUST)
            LE(DIFF,0)                                   :S(RETURN)
            JUST   BLANKS =                              :F(RETURN)S(JUST2)
JUST1       JUST   BLANKS =                              :F(JUST3)
JUST2       LINE = LINE HEAD SEP " "
            DIFF = GT(DIFF,1) DIFF - 1                   :S(JUST1)F(JUST4)
JUST3       JUST = LINE JUST
            LINE =                                       :(JUST1)
JUST4       JUST = LINE JUST                             :(RETURN)

END
```

This solution simply grinds through text, looking for blanks and adding one at each occurrence. Since more than one blank may need to be added at each possible place, the code at JUST3 starts the loop over if necessary. A solution that avoids "rivers" by adding blanks from the right and left alternatively is given on page 178 of Reference 13 and involves reversing the text on alternative lines before processing.

For Icon, a "natural" approach to this problem is to use transform-using rather than scan-using:

```
procedure just(line,length)
   local diff
   if not upto(" ",line) then return line
   diff := length - *line
   while diff > 0 do
      transform line using {
         while tab(upto(" ")) do {
            tab(many(" "))
            insert(" ")
            diff -:= 1
            if diff = 0 then break
            }
         tab(0)
         }
   return line
end
```

In the new facility, the more natural approach is to synthesize the desired result by concatenation in a matching procedure:

```
global newline

expand ::= (newline ||:= tabu(" ") || tabm(" ") || " ") & tab(0)

procedure just(line,length)
   local diff
   if not (line ? look(" ")) then return line
   diff := length - *line
   newline := ""
   while diff > 0 do
     if (line ?:= expand()) then diff -:= 1
     else {line := newline || line; newline := ""}
   return newline || line
end
```

This approach, could, of course, be taken in Icon — it merely is not obvious without the "discipline" implied by the new facilities. Note that expand produces two results — one assigned to newline and the other (tab(0)) returned to be assigned to line.

Again, a low-level approach is easily formulated and provides a contrast with the high-level solutions:

```
procedure just(line,length)
   local diff, i
   if not upto(" ",line) then return line
   diff := length - *line
   while diff > 0 do {
      i := 1
      while i := upto(" ",line,i) do {
         line[i] ||:= " "
         i := many(" ",line,i+1)
         diff -:= 1
         if diff = 0 then break
         }
      }
   return line
end
```

Infix-to-Prefix Conversion

This problem involves converting arithmetic expressions from infix form to fully parenthesized prefix form. Examples of the desired conversions are:

x	x
x+1	+(x,1)
((x+(1)))	+(x,1)
x-y-z	-(-(x,y),z)
3*delta+2	+(*(3,delta),2)
2^2^n	^(2,^(2,n))
(x^n)/(z+1)	/(^(x,n),+(z,1))

Only those operators shown above are considered here, and have their usual associativities and precedences.

One problem in the conversion is the removal of possibly superfluous parentheses in the infix form. Handling precedence and associativity is the other main problem. Precedence can be taken care of by the order of processing. Left-associative operators, such as +, present the greatest problem, since all three string-processing facilities operate in an essentially left-to-right manner, while transforming a left-associative operator requires finding its right-most occurrence.

The following SNOBOL4 solution is adapted from Reference 13, page 110:

```
                &ANCHOR = 1
                DEFINE("PREFIX(PREFIX)L,R,OP,M")
                STRIP = "(" BAL . PREFIX ")" RPOS(0)
                ASSIGN = *GT(M,0) TAB(*(M - 1)) . L LEN(1) . OP REM . R
                PMOP = (BAL ANY("+-") @M FAIL) | ASSIGN
                SSOP = (BAL ANY("*/") @M FAIL) | ASSIGN
                LASSOC = PMOP | SSOP
                RASSOC = BAL . L "^" . OP REM . R

READ            LINE = INPUT                                    :F(END)
                OUTPUT = PREFIX(LINE)                           :(READ)

PREFIX          PREFIX ? STRIP                                  :S(PREFIX)
                PREFIX ? LASSOC                                 :S(FORM)
                PREFIX ? RASSOC                                 :F(RETURN)
FORM            PREFIX = OP "(" PREFIX(L) "," PREFIX(R) ")"     :(RETURN)

END
```

An Icon solution from Reference 2, page 80, with slight modifications, is as follows.

```
procedure main()
   local line
   while line := read() do
      write(prefix(line))
end

procedure prefix(s)
   s := strip(s)
   return lassoc(s,"+-"|"*/") | rassoc(s,"^") | s
end

procedure strip(s)
   local t
   while scan s using ="(" & t := tab(bal(")")) & pos(-1) do
      s := t
   return s
end

procedure lassoc(s,c)
   local j
   j := 0
   scan s using every j := bal(c)
   if j = 0 then fail else return form(s,j)
end
```

```
procedure rassoc(s,c)
   local j
   scan s using j := bal(c) | fail
   return form(s,j)
end

procedure form(s,k)
   local a1, a2, op
   scan s using {
      a1 := tab(k)
      op := move(1)
      a2 := tab(0)
      }
   return op || "(" || prefix(a1) || "," || prefix(a2) || ")"
end
```

This solution illustrates the value of goal-directed evaluation — especially in the call of lassoc. The procedure lassoc itself demonstrates how the right-most left-associative operator is found using an iterative approach as opposed to failure-motivated pattern matching.

Finally, a solution using the proposed new facility is

```
strip ::= (tabs("(") & (tabb(")") ? strip()) || pos(-1)) | tab(0)
prefix ::= (strip() ? (lassoc("+-"|"*/") | rassoc("^") | tab(0)))
lassoc(s)t ::= (every t := tabb(s)) & form(tabs(\t))
rassoc(s) ::= form(tabb(s))
form(s) ::= move(1) || "(" || (s ? prefix()) || "," || (tab(0) ? prefix()) || ")"

procedure main()
   local line
   while line := read() do
      write(line ? prefix())
end
```

This solution deserves more explanation. The matching procedure strip illustrates nested matching, and can be compared directly to the iterative processing in the SNOBOL4 and Icon. The solution in the new facility is essentially recursive — the expression

```
tabb(")") ? strip()
```

recursively applies strip to a string enclosed in parentheses. The tab(0) alternative is an escape in case there are no parentheses. The technique used in lassoc(s) to locate the right-most left-associative operator is similar to that used in the Icon solution. The expression \t, a feature of Version 4 of Icon, fails if t is null, providing an escape similar to the test j = 0 in the Icon solution. Compare the recursive matching of prefix in form(s) to the recursive procedure calls in SNOBOL4 and Icon.

In retrospect, both the SNOBOL4 and Icon solutions can be adapted to resemble the new solution more closely (and conversely). The SNOBOL4 and Icon solutions are essentially as they appear in the literature and hence represent the "best" solutions of the authors at the time they were written. The new solution undoubtedly was influenced by the other solutions (especially the Icon solution).

10.2 Sentence Recognition

Finally, the close equivalence of SNOBOL4, Icon, and the new facility in characterizing languages is illustrated by the problem of recognizing sentences from the language characterized by the following grammar:

```
<s> ::= a <s> | <t> b | c
<t> ::= d <s> a | e | f
```

where <s> is the goal.

A SNOBOL4 solution is:

```
              S = "a" *S | *T "b" | "c"
              T = "d" *S "d" | "e" | "f"
              GOAL = POS(0) S RPOS(0)

READ          LINE = INPUT                                       :F(END)
              LINE ? GOAL                                        :F(NO)
              OUTPUT = "accepted"                                :(READ)
NO            OUTPUT = "rejected"                                :(READ)

END
```

An Icon solution is:

```
procedure main()
   local line
   while line := read() do
      if recogn(s,line) then write("accepted") else write("rejected")
end

procedure recogn(goal,text)
   return scan text using
      goal() & (&pos = pos(0))
end

procedure s()
   suspend (="a" || s()) | (t() || ="b") | ="c"
end

procedure t()
   suspend (="d" || s() || ="d") | ="e" | ="f"
end
```

Finally, a solution using the new facility is:

```
s ::= (tabs("a") || s()) | (t() || tabs("b")) | tabs("c")
t ::= (tabs("d") || s() || tabs("d")) | tabs("e") | tabs("f")
goal() ::= s() & pos(0)

procedure main()
   local line
   while line := read() do
      if (line ? goal()) then write("accepted") else write("rejected")
end
```

10.3 Discussion of Examples

The problem of reformatting lines is sufficiently simple that a low-level approach is easy to formulate and to understand. This is to be expected — for sufficiently simple string processing, low-level facilities are certain to be superior to high-level ones in programming ease and clarity. Since the motivation here is to develop facilities suitable for complex string processing problems, the question is more how much burden such high-level facilities place on the solution of simple problems.

For the problem of reformatting lines, the differences among the various solutions are not substantial, although the Icon string scanning solutions appear to be unnecessarily overbearing by comparison. The solution using the new facility is similar, in many respects, to the SNOBOL4 solution, although the advantages of being able to use the function repertoire in matching procedures is evident when compared to the necessity of separating string analysis and synthesis into separate statements in the SNOBOL4 solution.

In the case of line justification, the algorithm used is itself somewhat awkward and this difficulty is reflected in all the solutions. As such, it serves to show how such problems must be dealt with. Here the low-level solution is comparable in complexity to the Icon solution — the one case where transform-using shows to advantage. The low-level solution does use some non-obvious coding techniques (consider the first two lines of the inner while loop). The solutions in SNOBOL4 and in the new facility are comparable in many respects. The main difference in the new facility is that the developing result is augmented within the matching procedure.

In infix-to-prefix conversion, the SNOBOL4 method of finding the right-most left-associative operator is particularly contorted and illustrates the frequent need in SNOBOL4 to use "failure-motivated" patterns [7]. The Icon solution uses goal-directed evaluation to advantage (see the second line of prefix), but it is also quite lengthy. (The ability to match from right to left would be very helpful here, but that raises several thorny issues [7].) Here the new facility is more impressive and the potential for matching procedures to be "descriptive" is evident. This solution also exhibits some coding techniques in the new facility that are discussed in the next section.

In the sentence recognition problem, all the solutions are essentially isomorphic. The SNOBOL4 solution is somewhat more concise than the others. This conciseness is partly due to the way that patterns are formed and used and partly due to idiosyncrasies in SNOBOL4 syntax, such as the absence of an explicit operator for concatenation. (This aspect of SNOBOL4 causes its share of troubles also [5].) Some of the "busyness" of the new solution could be avoided by syntactic sugar, such as the use of =s for tabs(s), as in Icon. Note the conciseness obtained by procedural encapsulation also — another instance of syntactic sugar.

10.4 Coding Techniques Using Matching Procedures

The result of evaluating *e1* ? *e2* is the result of evaluating *e2*. This result generally depends on the value of *e1*, of course. In most circumstances, the result is a concatenation of strings. Some of these strings, using Doyle's terminology, are indigenous, resulting from the application of matching functions such as tab(i) or programmer-defined matching procedures. Other strings are exogenous. Exogenous strings are frequently literals, but may result from any string-valued operation. As mentioned in Section 5, indigenous and exogenous strings are simply special cases of transformations. In the line reformatting solution

```
left(trim(tab(30)),30)
```

is an example of such a transformation. Another example from the infix-to-prefix solution is

```
form(tabb(s))
```

Transformation is a useful paradigm. Transformation in this sense is simply a normal computational technique, as opposed to the transform-using construct in Icon, which returns the modified value of &subject.

In developing a result by concatenation, it is frequently necessary to discard results of matching functions or procedures. This problem, discussed in Section 5, is illustrated in the solution to the line reformatting problem by

```
tab(4) & left(trim(tab(30)),30)
```

and by

```
tabs("(") & (tabb(")") ? strip())
```

in the solution to the infix-to-prefix conversion problem.

In the first case, the first four characters of the subject are simply irrelevant. In the second case, the literal left parenthesis is required, but it is not a desired part of the result. As illustrated, conjunction serves to omit its first operand from the result. The use of conjunction in this fashion amounts to an idiom. For readability, a different syntax for this situation might be desirable. Thus

e1 -> e2

would more clearly indicate that the result of evaluating *e1* is "replaced" by the result of evaluating *e2*. (It is interesting to consider replacing the operator & by -> in Icon to indicate "goal-directed evaluation in sequence". The mental trick lies in the "mnemonic" implication of the symbol ->.)

The application of matching procedures within matching procedures frequently proves valuable, as is illustrated in the solution to the infix-to-prefix problem. The interesting point is that in *e1* ? *e2*, either *e1* or *e2* may, themselves, be matching procedures. The "normal" case is for *e2* to be a matching procedure — that is the implication of *e1* ? *e2*. On the other hand, if *e1* is a matching procedure, it produces a result based on the current subject. This is illustrated by

```
tabb(")") ? strip()
```

in which the matching procedure strip is applied to the balanced string up to a right parenthesis.

In some cases, it is useful to "pass along" a string to another procedure. This is illustrated in the solution to the infix-to-prefix conversion problem by

```
rassoc(s) ::= form(tabb(s))
form(s) ::= move(1) || "(" || (s ? prefix()) || "," || (tab(0) ? prefix ()) || ")"
```

Here the first operand of an infix expression is produced by tabb(s). However, this operand is not needed in the desired result until the operator following it and an exogenous left parenthesis have been concatenated. In the first attempt to write a matching procedure to do this, a natural approach is to use an auxiliary identifier:

```
rassoc(s)t ::= (t := tabb(s)) & move(1) || "(" || (t ? prefix()) . . .
```

Use of the matching procedure form(s) avoids the auxiliary identifier t, since the value matched by tabb(s) is passed through the argument s of form. As a consequence, the "match and discard" construction

```
(t := tabb(s)) & move(1)
```

is avoided as well.

In complicated string processing, this "pass along" idiom is naturally suggested when the result of a matching procedure is needed, but not at the time it is produced. The solution is simply to introduce another matching procedure at this point, where the new matching procedure takes the string in question as an argument and supplies the remainder of the result that is needed.

For the programmer who is accustomed to using patterns in the style of SNOBOL4, the opportunity to use a variety of control structures in matching procedures may be difficult to perceive. The examples in Section 10.1 do not make much use of such possibilities, but one instance from the solution to the infix-to-prefix conversion problem is suggestive:

```
every t := tabb(s)
```

All this expression does is assign to t the longest balanced string up to the operator s. Compare this to the possible techniques for accomplishing this in SNOBOL4 — or printing all the parenthesis-balanced substrings of a string in SNOBOL4:

```
S ? BAL $ OUTPUT FAIL
```

In this new facility, the natural method is

```
every write(s ? tabb())
```

There is, of course, a difference between the semantics of BAL in SNOBOL4 and tabb() in the new facility, but the point remains the same.

More conventional control structures have their uses also. Consider the procedures for c_tab(i) and c_len(i) in Section 3.

One rather serious problem with the new facility is illustrated by the matching procedure used in the solution of the line justification problem:

```
expand ::= (newline ||:= tabu(" ") || tabm(" ") || " ") & tab(0)
```

In this solution, expand essentially produces two results. As a side effect, the value of newline is augmented, while the value returned by expand is tab(0), which is assigned to line. The need to produce multiple results is common, especially in complex problems. In the solution to the line-justification problem, the second "result" is accomplished by assignment to newline. Such side-effect approaches are common and necessary in SNOBOL4 and are accepted because they are necessary. While side-effect approaches are more uncomfortable in a language like Icon, the more serious underlying problem is the scope of identifiers. In the solution given, newline must be declared global. The necessity for such declarations is annoying, but could be avoided if Icon interpreted undeclared identifiers as global rather than local (this was done in an early version of Icon). Such a solution does not solve the basic problem however. Suppose expand needed to assign a value to an identifier, such as line, that was local in the context in which expand is called.

Call-by-reference or pointers could be used to solve this problem. Another solution would be a dynamic scoping facility, such as is used in SL5 [9]. A more general solution to the problem of multiple results would be preferable, but it is difficult to see how such a solution could be formulated without major changes in the base language.

11. Conclusions

The cursor and substring models given in Sections 3 and 4 illustrate two ways that matching procedures can be used to implement pattern matching in the style of SNOBOL4. While there are a few aspects of pattern matching in SNOBOL4 that these models cannot accommodate, none of these exceptions limits the general aspects of pattern matching nor detracts from its facilities in any essential way. In fact, the procedural models are in many respects more general than pattern matching in SNOBOL4 and certainly allow many extensions and generalizations.

The procedural models strip away a level of indirection that is present in SNOBOL4. Instead of using patterns that reference matching procedures, the procedural models use procedures directly. Consequently, matching procedures can be used in combination with all of the other features of the base language. Neither is the repertoire of operations that can be used in pattern matching limited, nor is it necessary to have a separate set of pattern-construction operations that parallels the control structures and operations of the base language.

The penalty for removing this level of indirection is a somewhat less concise syntax. In addition, the availability of all base-language operations during pattern matching may lure programmers into awkward programming constructions — "a discipline of pattern matching" is not imposed by the language, but must be observed by programmers.

Icon has most of the facilities needed to support matching procedures. The adaptation of Icon to this discipline is illustrated by the model given in Section 9 and supported by the programming examples of Section 10.

There are, of course, remaining problems — especially with the scoping of identifiers. Nor does the proposal of Section 9 even begin to address more basic problems, such as the recurrent need for producing multiple results from pattern matching in a structured fashion. The proposal does, however, suggest a possible compromise between the linguistic schism that patterns impose on SNOBOL4 and the inadequate capacity for abstraction offered by string scanning in Icon.

Acknowledgement

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

References

1. Griswold, Ralph E. *Pattern Matching in Icon.* Technical Report TR 80-25, Department of Computer Science, The University of Arizona. October 1980.
2. Coutant, Cary A., Ralph E. Griswold, and Stephen B. Wampler. *Reference Manual for the Icon Programming Language; Version 3.* Technical Report TR 80-2, Department of Computer Science, The University of Arizona. May 1980.
3. 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.
4. Dewar, Robert B. K. *SPITBOL Version 2.0.* Technical Report S4D23, Illinois Institute of Technology. February 1971.
5. Griswold, Ralph E. "A History of the SNOBOL Programming Languages", *SIGPLAN Notices*, Vol. 13, No. 8 (August 1978). pp. 275-308.
6. Gimpel, James F. *SITBOL Version 3.0.* Technical Report S4D30b, Bell Telephone Laboratories, Holmdel, New Jersey. June 1973.
7. Doyle, John N. *A Generalized Facility for the Analysis and Synthesis of Strings, and a Procedure-Based Model of an Implementation.* Technical Report S4D48, Department of Computer Science, The University of Arizona. February 1975.
8. Yngve, Victor H. Private communication. 1964.
9. Hanson, David R. and Ralph E. Griswold. "The SL5 Procedure Mechanism", *Communications of the ACM*, Vol. 21, No. 5 (May 1978). pp. 392-400.
10. Dewar, Robert B. K. and Anthony P McCann. "MACRO SPITBOL — A SNOBOL4 Compiler", *Software — Practice and Experience*, Vol. 7 (1977). pp. 95-113.
11. Griswold, Ralph E. and David R. Hanson. "An Alternative to the Use of Patterns in String Processing", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2 (April 1980), pp. 153-172.
12. Griswold, Ralph E., James F. Poage, and Ivan P. Polonsky. *The SNOBOL4 Programming Language.* Second Edition. Prentice-Hall, Inc. Englewood Cliffs, New Jersey. 1971.
13. Griswold, Ralph E. *String and List Processing in SNOBOL4; Techniques and Applications.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1975. p. 14.

Appendix A — Procedures for the Cursor Model of SNOBOL4 Pattern Matching

```
c_len(i) ::= if 0 <= i <= *subject+1-cursor then cursor := cursor+i
c_tab(i) ::= if cursor-1 <= i <= *subject then cursor := i+1
c_pos(i) ::= (cursor = i+1)
c_rtab(i) ::= if cursor-1 <= i <= *subject then cursor := *subject-i
c_rpos(i) ::= if cursor-i <= i <= *subject then cursor = *subject+1-i
c_arb ::= (cursor := (cursor to *subject+1))

c_match(s) ::= (cursor := match(s,subject,cursor))
c_any(s) ::= (cursor := any(s,subject,cursor))
c_notany(s) ::= (cursor := any(~s,subject,cursor))
c_span(s) ::= (cursor := many(s,subject,cursor))
c_breakx(s) ::= (cursor := upto(s,subject,cursor))
c_break(s) ::= (cursor := upto(s,subject,cursor) \ 1)
c_balu ::= (cursor := bal(")",,,subject,cursor))
c_bbal ::= (c_match("(") & c_balu() & c_len(1)) | c_notany("()")
c_bal ::= c_bbal() & c_arbno(c_bbal)

c_cat(p1,p2) ::= p1() & p2()
c_alt(p1,p2) ::= p1() | p2()
c_arbno(p) ::= cursor | (p() & c_arbno(p))

procedure substr(p)
   return subject[.cursor:(p() & cursor)]
end

c_anchor ::= cursor
c_float ::= (cursor := (1 to *subject+1))

procedure c_apply(s,p)
   subject := s
   cursor := 1
   return (c_mode() & substr(p))
end

procedure c_repl(s,p)
   local mid
   subject := s
   cursor := 1
   return {
      (head := substr(c_mode)) &
      (mid := substr(p)) &
      (tail := subject[cursor:0]) &
      mid
      }
end
```

Appendix B — Procedures for the Substring Model of SNOBOL4 Pattern Matching

```
s_len(i) ::= if 0 <= i then subject[.cursor:cursor := cursor+i]
s_tab(i) ::= if cursor−1 <= i then subject[.cursor:cursor := i+1]
s_pos(i) ::= (cursor = i+1) & ""
s_rtab(i) ::= if i <= *subject then subject[.cursor:cursor := *subject+1−i]
s_rpos(i) ::= if cursor−1 <= i <= *subject then (cursor = *subject+1−i) & ""
s_arb ::= subject[.cursor:cursor := (cursor to *subject+1)]

s_match(s) ::= subject[.cursor:cursor := match(s,subject,cursor)]
s_any(s) ::= subject[.cursor:cursor := any(s,subject,cursor)]
s_notany(s) ::= subject[.cursor:cursor := any(~s,subject,cursor)]
s_span(s) ::= subject[.cursor:cursor := many(s,subject,cursor)]
s_breakx(s) ::= subject[.cursor:cursor := upto(s,subject,cursor)]
s_break(s) ::= subject[.cursor:cursor := upto(s,subject,cursor)  1]
s_balu ::= subject[.cursor:cursor := bal(")",,,subject,cursor)]
s_bbal ::= (s_match("(") || s_balu() || s_len(1)) | s_notany("()")
s_bal ::= s_bbal() || s_arbno(s_bbal)

s_cat(p1,p2) ::= p1() || p2()
s_alt(p1,p2) ::= p1() | p2()
s_arbno(p) ::= "" | (p() || s_arbno(p))


s_anchor ::= ""
s_float ::= subject[1:(cursor := (1 to *subject+1))]

procedure s_apply(s,p)
   subject := s
   cursor := 1
   return (s_mode() & p())
end

procedure s_repl(s,p)
   local mid
   subject := s
   cursor := 1
   return {
      (head := s_mode()) &
      (mid := p()) &
      (tail := subject[cursor:0]) &
      mid
      }
end
```