EXTENSIBLE PATTERN MATCHING IN SNOBOL4*

Ralph E. GRISWOLD
Department of Computer Science, The University of Arizona
Tucson, Arizona 85721

Although the string pattern-matching facilities of SNOBOL4 are the most powerful of those of any widely used programming language, they suffer both from complexity and lack of a mechanism for defining new scanning procedures. This paper describes a definitional mechanism that provides extensibility for the existing facilities in which programmer-defined scanning procedures are written as coroutines at the source-language level. The result provides a substantial increase in power as well as the potential for simplifying the existing language. The relationship of this new facility to the present language is discussed, as are implications for more extensive language changes.

1. INTRODUCTION

The SNOBOL4 programming language [1] is best known for its powerful facilities for string analysis. It is the only widely used high-level language that offers extensive general facilities for pattern matching, and these facilities are probably the primary reason for its acceptance. In spite of this, pattern matching causes SNOBOL4 programmers more difficulty than any other aspect of the language. The sheer size and complexity of the pattern-matching facility presents a significant problem. There are 34 language operations related in one way or another to pattern matching. Yet programmers often find it difficult or impracticable to do things they wish to do.

One of the major defects of pattern matching in SNOBOL4 is the lack of a mechanism whereby programmers can *define* what is to be done. Although patterns can be composed of simpler components to make up very complicated structures, the programmer must rely entirely on built-in mechanisms for carrying out the actual analysis.

This paper discusses a proposed solution to this problem in which the programmer can write source-language procedures to perform string analysis. The result is a facility that greatly extends the power of pattern matching and at the same time makes it possible to substantially reduce the number of built-in operations, thus simplifying the vocabulary of the language.

2. THE PRESENT PATTERN-MATCHING MECHANISM

The programmer-defined scanning procedures described in this paper are based on the existing pattern-matching facilities in SNOBOL4. Although there are more radical approaches that could be taken, the success of the present mechanism and the vast amount of experience with its use make it a natural basis on which to build. On the other hand, experience with the extensions described here suggests more far-reaching changes.

In the present system, pattern matching is performed by a scanner that has two arguments, a subject string and a pattern. The analysis of the subject string is carried out according to the structure and contents of the pattern. There is an implicit algorithm for search and backtracking that is "driven by" the pattern. Patterns are data objects that indicate scanning (matching) procedures which are to perform the actual analysis. Scanning procedures move a cursor that identifies the current place in the subject string that is being examined.

For the SNOBOL4 programmer, there are two aspects of pattern matching: pattern construction and the actual scanning process itself. Although the division between these two processes is often obscure in the mind of the programmer, the distinction is an important one.

Patterns are data objects. Of the 34 operations related to pattern matching, 15 construct patterns. An example is the function LEN(n), which constructs a pattern that, during scanning, attempts to match n characters, i.e., advance the cursor position by n. In fact, LEN(n) constructs a pattern that contains a pointer to a matching procedure, SLEN, and the argument, n, to be used during scanning. See fig. 1.

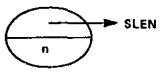

Fig. 1 - The Pattern Constructed by LEN(n)

The details of pattern construction and the internal representation of patterns are not important here, although the general concept of a pattern as a data object containing scanning procedure names and argument values *is* important.

In SNOBOL4, the programmer's knowledge of the structure of patterns, *per se*, is implicit. Their existence and general nature is documented, but they cannot be examined or modified. That situation also applies to the new facilities described in this paper; although there are certainly possibilities for adding facilities for manipulating patterns more directly, this work is restricted to the design and use of scanning procedures.

The scanning procedures in SNOBOL4 are all built in and their operation is somewhat mysterious (and not well documented). The facilities described here are designed to provide a mechanism whereby programmers can define their own scanning procedures and write the code for them at the SNOBOL4 source-language level.

*This work was supported by the National Science Foundation under Grant GJ-36272.

3. THE IMPLEMENTATION OF PATTERN MATCHING

In order for programmers to be able to write scanning procedures, they must know more than they presently do about the process of pattern matching. Until recently, the mechanisms by which scanning could be implemented were poorly understood. Of course, the various implementations of SNOBOL4 contain scanning programs. In some cases, the program itself is the best documentation [2]. Gimpel [3,4] provided the first attempts at analyzing and systematizing the pattern-matching process. Other formulations have been implemented and described [5,7]. While these formulations have had various degrees of success in describing the existing facilities, none has the necessary generality to handle significant extensions. Yet if a formulation of the scanning process cannot handle extensions to the built-in mechanism, it can hardly serve as a basis for programmer-defined scanning procedures. The breakthrough in understanding came with the Doyle-Druseikis coroutine model [8,9]. The facilities described here are based on that model.

In the Doyle-Druseikis model, patterns are constructed as described in the preceding section. The actual form of implementation is not important; a tree structure is used here, since it makes the relationships easy to visualize. As an example, the expression

```
LEN(1) RPOS(0) | LEN(3)
```

produces the pattern shown in fig. 2.

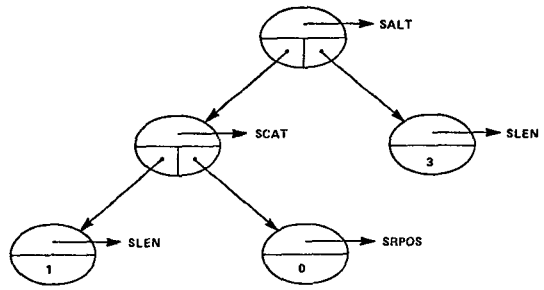

Fig. 2 - A Pattern

The S prefixes identify scanning procedures (to distinguish them from pattern-constructing procedures). SALT and SCAT are the scanning procedures for alternation and concatenation.

It is the scanning procedures themselves that are of interest. In the Doyle-Druseikis model, these procedures are coroutines. The process of pattern matching takes place by successive invocation of coroutines pointed to by nodes in the pattern tree. The environments of suspended coroutines contain "historical" information about the course of scanning and make possible the search for alternatives and the reversal of tentative effects (such as cursor advancement) when the suspended environments are resumed during backtracking.

The details of the coroutine model are too extensive to describe here, but are well covered in the literature [8,9]. The necessary essentials are described below in the form in which they would appear as SNOBOL4 source-language operations.

3.1 Source-Language Coroutine Operations

The function COROUTINE defines a coroutine in much the same way that the function DEFINE defines a function. For example, a programmer-defined coroutine is defined by

```
COROUTINE("F(X,Y)L")
```

In this case, F is the name of the coroutine, X and Y are its formal arguments, and L is a local variable. Using the same convention for coroutines that is used for programmer-defined functions in SNOBOL4, the entry point of this coroutine is at the label F.

Coroutines are not invoked like functions. Instead, an environment for a particular invocation of a coroutine is created. This is done by the function CREATE, as illustrated by the statement

```
E = CREATE("F",1,2)
```

which creates an environment for the coroutine F with arguments 1 and 2. Note that CREATE returns a value that is a SNOBOL4 source-language data object. There may be many environments for a given coroutine in existence at the same time.

A coroutine is activitated by the function RESUME:

```
INVOKE  RESUME(E)
```

The argument of RESUME is a coroutine environment (in this case corresponding to F). When resumed initially, execution of a coroutine begins at its entry point.

Procedures for coroutines resemble those for functions, except that the entry and exit mechanisms are different. Continuing the example above, the coroutine F might begin as follows:

```
F       GT(X,Y)                 :F(F1)
        RESUME()
           .
           .
           .
```

If X is greater than Y, the second statement is executed. By convention, resumption without the specification of a particular environment causes suspension of coroutine activation and return of control to the place that caused its activation (in this case, the statement above labeled INVOKE). If the environment E is resumed again, the coroutine is reactivated at the point where activation was suspended, and so on.

Coroutines may also signal failure, much in the same fashion that defined functions signal failure by FRETURN. This is accomplished by the function FRESUME, which switches environments in the same manner as RESUME, but also signals failure. This process is referred to as "f-resuming".

In summary, there are the following coroutine operations:

```
COROUTINE:  defines a coroutine
CREATE:     creates an environment for a coroutine
RESUME:     resumes activation of an environment
FRESUME:    resumes activation of an environment and
            signals failure
```

If RESUME or FRESUME is not given a specific argument, the environment resumed is the one that caused activation of the current environment.

3.2 Operation of the Scanner

In the Doyle-Druseikis model, the scanner itself is a recursive procedure, not a coroutine. The arguments provided to the scanner are a subject string and a pointer to the root (top) node of a pattern tree. The scanner establishes variables that are global to scanning, such as the subject string and cursor position, creates an environment for the coroutine given in the root node of the pattern,

and then resumes that coroutine. Scanning proceeds as successive scanning procedures are activated. These procedures examine the subject string, tentatively move the cursor, and so forth. When a scanning procedure succeeds (for example, matching desired characters or advancing the cursor to a desired position), it resumes the procedure that resumed it. The scanner completes successfully when it is resumed. If a scanning procedure is unable to carry out a desired process, it f-resumes instead, which causes backtracking in search for alternatives. There is also a mechanism for f-resuming a specific environment, which, by convention, signals the f-resumed procedure to reverse any tentative effects it may have caused (rather than seeking alternatives). Scanning fails if the scanner is f-resumed.

4. PROGRAMMER-DEFINED SCANNING PROCEDURES

Programmer-defined scanning procedures must have access to some of the same variables that built-in scanning procedures do. While the extent of access to such "internal" variables is debatable, the subject and cursor position are certainly essential. These variables are represented by the keywords &SUBJECT and &CURSOR. The use of keywords (rather than, for example, simple identifiers) conforms to the usual SNOBOL4 convention for accessing system variables from the source language. While &SUBJECT is most naturally treated as a protected keyword, it is necessary to be able to change the value of &CURSOR. However, it seems reasonable to restrict the value that can be assigned to &CURSOR to be between 0 and the length of &SUBJECT. This introduces the concept of a range-protected keyword. An attempt to assign an out-of-range value to such a keyword causes a failure signal in much the same way as an out-of-range array index causes failure.

Patterns for programmer-defined scanning procedures are the same as those constructed for built-in scanning procedures. This requires a mechanism whereby the programmer can define pattern-construction procedures. The definition of pattern-constructing procedures and the corresponding scanning procedures are done by the function SPROC. An example is given by a programmer-defined version of LEN:

```
SPROC("LEN:DSLEN(N)C")
```

The colon separates the name of the pattern-construction function (LEN) from the prototype for the scanning coroutine. Note that SPROC defines a function LEN that constructs patterns pointing to DSLEN, and it also defines the coroutine DSLEN. Subsequently, the statement

```
L1 = LEN(1)
```

constructs the pattern shown in fig. 3.

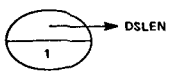

Fig. 3 - A Pattern for a Programmer-Defined Scanning Procedure

The only difference between this pattern and one constructed by a built-in pattern-constructing function is that DSLEN is a programmer-defined coroutine.

The source-language procedure for DSLEN might be coded as follows:

```
DSLEN   C = &CURSOR
        &CURSOR = &CURSOR + N    :F(SLENF)
        RESUME()
        &CURSOR = C
SLENF   FRESUME()
```

The local variable C is used to save the current value of the cursor. If the cursor can be incremented by N, DSLEN resumes the procedure that invoked it, signaling success. Otherwise DSLEN f-resumes, signaling failure. Note that the cursor has not been changed in this case, since it is range-protected. If DSLEN succeeds, but is subsequently resumed, that occurrence indicates a request for an alternative, initiated by backtracking due to the failure of some other scanning procedure. Since DSLEN can only advance the cursor by N characters, it has no alternative course of action and signals that fact by f-resuming, after first restoring the cursor to the value it had when DSLEN was first entered. Some scanning procedures possess the capability of taking alternative action ("matching an alternative"), in which case they resume, signaling success, rather than f-resuming like DSLEN. By convention, a coroutine that has f-resumed to signal failure cannot be reactivated. An attempt to do so is a programming error.

4.1 Additional Examples

A somewhat more complicated scanning procedure is illustrated by ALTN, which accepts the Nth alternative matched by a pattern. An example is ALTN(BAL,3), which matches the third alternative matched by BAL. For example, given the string (A)(BC)(DE)F, ALTN(BAL,3) matches (A)(BC)(DE). The pattern-construction and scanning procedures are defined by:

```
SPROC("ALTN:SALTN(P,N)E")
```

The use of a pattern as an argument introduces a new concept: a scanning procedure that invokes another scanning procedure. This requires creating an environment for the scanning procedure given in the root of the pattern tree. To avoid requiring the programmer to know about the structure of patterns, the function SCREATE is provided to create an environment for the coroutine contained a pattern node. The result returned by SCREATE is the same as that returned by CREATE; the difference is that SCREATE locates the coroutine name and arguments automatically. A procedure for SALTN follows:

```
SALTN   GE(N,1)                  :F(FALTN)
        E = SCREATE(P)
TALTN   RESUME(E)                :F(FALTN)
        N = GT(N,1) N - 1        :S(TALTN)
        RESUME()
        FRESUME(E)
FALTN   FRESUME()
```

SALTN first checks N to be sure that it is positive, failing if it is not. SALTN then creates an environment, E, for the procedure given in the root node of P, and then resumes the corresponding coroutine. A failure signal as a result indicates that P cannot match, in which case SALTN fails. Otherwise, the count is decremented. If the desired alternative has not been reached, E is resumed again, requesting another alternative. If the desired alternative has been reached, SALTN resumes, signaling success. If SALTN is resumed again at this point, that indicates a request for an alternative. This is impossible by the definition of ALTN, so failure is signaled. On the other hand, a failure signal at this point indicates that SALTN is to reverse any tentative effects that it may have caused. SALTN itself causes no reversible effects, but the scanner procedure it invoked may have (for example, if P were BAL, the scanning procedure for BAL would have advanced the cursor). SALTN therefore f-resumes E ("passing the failure signal

along") to give its procedure the opportunity to reverse effects (but not to seek alternatives).

A final example illustrates some of the possibilities afforded by programmer-defined scanning procedures that are not possible in the present SNOBOL4 language. STR is a scanning procedure that provides trace information during the course of scanning. The defining statement is:

```
SPROC("TR:STR(S,P)E")
```

with the procedure

```
STR     E = SCREATE(P)
        OUTPUT = "ENTERING " S
ST      RESUME(E)                          :F(STRF)
        OUTPUT = S " SUCCEEDED"
        RESUME()                           :F(STRFF)
        OUTPUT = "SEEKING ALTERNATIVES FOR " S   :(ST)
STRF    OUTPUT = S " FAILED"
        FRESUME()
STRFF   OUTPUT = S " REVERSING EFFECTS"
        FRESUME(E)
        OUTPUT = "EFFECTS FOR " S " REVERSED"
        FRESUME()
```

The argument S serves as a tag to permit identification of different tracing patterns. Other information, such as cursor position, is easily added. Such a pattern might be used as follows:

```
BALTR = TR("BALTR",BAL)
```

BALTR matches like BAL but also provides trace information.

The examples above are only suggestive of the possibilities for programmer-defined scanning procedures. The variety is endless.

5. CONCLUSIONS

Experience with the use of the facility described above has shown that after an initial learning period in the use of coroutines and the conventions that scanner procedures must obey, it is relatively easy to write and debug programmer-defined scanner procedures. The effects of coding errors potentially may be relatively severe, however. Programmer-defined scanning procedures must work cooperatively with built-in scanning procedures. Failure to observe the scanner protocol can cause program malfunction whose source is difficult to isolate. Such problems have been surprisingly rare in practice, however.

An important consideration in the inclusion of new features is the additional burden of language complexity that they impose. Even the rudimentary facilities described here add substantially to an already complicated language. The fact that programmers do not have to use these facilities is only a partial relief. There is, however, a possibility of significant reduction in the vocabulary of SNOBOL4: the removal of built-in pattern facilities that can now be defined. In fact, many of the built-in pattern-matching facilities are included in SNOBOL4 because they are occasionally necessary (for example, SUCCEED) and because there is no mechanism by which the programmer can define them. With the mechanism described here, virtually all of the present built-in facilities can be defined. Of course many facilities should remain built in for convenience and efficiency because of their frequency of use (for example, LEN). Others, such as ARBNO, probably should remain built in because of their complexity (the likelihood of a programmer-defined ARBNO working properly is small indeed).

5.1 Implications for More Extensive Language Changes

Perhaps more important are the possibilities opened up by a more general concept of programmer-defined scanning. Many scanning procedures serve only as control relationships -- alternation, concatenation, and ALTN are examples. Such procedures are in no way limited to the process of string scanning. They do not examine the subject or alter the cursor; they merely serve to invoke other procedures and to control the search and backtracking processes. Such control procedures can be used, without modification, in processes other than pattern matching. Furthermore, the concept of a cursor is not limited to strings. DSLEN, for example, has a perfectly valid interpretation if the subject is an array. The facilities described here are easily applied to Doyle's string synthesis facility [9], Hallyburton's facility for processing structures [10], and some of the more recently developed features of AI languages [11].

It should be noted that SNOBOL4, as presently structured, is not adequate to handle the control processes required to support coroutines. Although it was possible to implement programmer-defined scanning procedures in SNOBOL4, substantial concessions had to be made, and the actual form is somewhat less pleasant than given here. It has, however, been long recognized that the control structures of SNOBOL4 would benefit from "modernizing". The point is, that although the facilities described here cannot be imbedded in a reasonable way in SNOBOL4 as it presently stands, they may be sufficiently promising to encourage a long-overdue overhaul of the language.

ACKNOWLEDGMENT

Extensible pattern matching in SNOBOL4 was essentially "unthinkable" until the development of the coroutine model of scanning. This work therefore owes its existence to the work of John Doyle and Frederick Druseikis. In addition, John Doyle's implementation of the coroutine model provided an invaluable vehicle for developing and testing programmer-defined scanning procedures. Special thanks go to David Hanson for assistance in preparing this paper and for a number of helpful suggestions about its content.

REFERENCES

[1] Ralph E. Griswold, James F. Poage, and Ivan P. Polonsky, The SNOBOL4 programming language, 2nd ed., Prentice-Hall, Englewood Cliffs, N.J., 1971.

[2] Ralph E. Griswold, SNOBOL4 written in SIL, SNOBOL4 project document S4D28, Bell Labs, Holmdel, N.J., August 25, 1971.

[3] James F. Gimpel, A theory of discrete patterns and their implementation in SNOBOL4, Communications of the ACM, vol. 16, no. 2, February, 1973, 91-100.

[4] James F. Gimpel, Nonlinear pattern theory, SNOBOL4 project document S4D33, Bell Labs, Holmdel, N.J., October 1, 1973.

[5] Paul Joseph Santos Jr., FASBOL, A SNOBOL4 compiler, Electronics Research Laboratory memorandum no. ERL-M134, The University of California, Berkeley, December, 1971.

[6] Timothy T. Tye, CISBOL, A compiler implementation of SNOBOL, research report, Department of Computer Science, The University of Arizona, May 17, 1972.

[7] William R. Sears III, The design of SIXBOL, A fast implementation of SNOBOL4 for the CDC 6000 series computers, SNOBOL4 project document S4D45, The University of Arizona, November 25, 1974.

[8] Frederick C. Druseikis and John N. Doyle, A procedural approach to pattern matching in SNOBOL4, Proceedings of the ACM Annual Conference, November, 1974, 311-317.

[9] John N. Doyle, A generalized facility for the analysis and synthesis of strings and a procedure-based model of an implementation, SNOBOL4 project document S4D48, The University of Arizona, February 11, 1975.

[10] John C. Hallyburton Jr., Advanced data structure manipulation facilities for the SNOBOL4 language, SNOBOL4 project document S4D42, The University of Arizona, May 24, 1974.

[11] Lucio F. Melli, The 2.PAK language primitives for AI applications, technical report no. 73, Department of Computer Science, University of Toronto, December, 1974.