Short Communications
Programming Languages

An Algol-Based Implementation of SNOBOL4 Patterns

J. Nevil Brownlee
University of Auckland

Key Words and Phrases: patterns, SNOBOL4, pattern matching, string processing, pattern implementation, algorithms in Pascal
CR Categories: 4.29

1. Introduction

When a string appears as the subject of a SNOBOL4 statement [5], it may be scanned to see whether or not it contains a specified pattern. The position reached in the subject string during such pattern matching is called the cursor position. Any pattern P may be matched against a subject string S at a given cursor position c. If the match succeeds the cursor will move to a new position. Gimpel [3] formalized P as a function

$$P(S, c) = c'.$$

Since P may have implicit alternatives (i.e. contain several component patterns in an order of preference), c' is an ordered sequence of cursor positions.

In the usual implementations of SNOBOL4 [2–4] a pattern is compiled into a graph, whose nodes are patterns and whose edges indicate the order in which the nodes must be matched. A "scanner" procedure is then called to thread a path through the graph, matching the pattern. This implementation is straightforward, and has the advantage that patterns can be dynamic, i.e. modified after being compiled. However, since the scanner is fundamentally an interpreter (using pattern graphs as its input code), pattern matching appears to be a time-consuming process.

This paper describes an alternative approach in which SNOBOL4 patterns are compiled into Algol functions, which can be combined into larger (more complicated) patterns, and which are directly executed. It was developed as part of the implementation of "Snobal/67" [1], a Burroughs B6700 compiler for a SNOBOL4-like language.

The algorithms presented below were developed in Burroughs B6700 Extended Algol, but are described here in Pascal [6, 7], since this is probably a more widely known language. The version of Pascal used has two extensions: record-valued functions, and the ability to declare varying length arrays. The algorithms are by no means complete; for example, they do not include any attempt to avoid futile searching for a match [3].

2. Pattern Functions

Figure 1 gives the global declarations required for pattern matching. First we declare the type *string* to describe SNOBOL4 strings. A *string* is a record with two components—*length* is the number of characters in the string, and *body* is an array containing them. If a string is non-null, *length* is positive and *body*[1] contains its first character. String *subject* is assumed to be the subject string, in which we will attempt to match patterns.

Next we declare type *state*, a record containing all the variables which will be needed by pattern functions (see below). For temporary storage of *state* variables a push-down stack is provided by array *patternstack*, integer *psp* (which stores the location of the last variable entered), procedures *push, pop, cut,* and function *topofstack*.

A pattern function F requires at least one *state* parameter s, and returns an updated version of this *state*. When F is first invoked, *s.cursor* gives the position in *subject* at which F is to start matching, and *s.succeed* is false. If the function does not match, *F.succeed* is set false; otherwise *F.succeed* is true, and *F.cursor* indicates the starting cursor position for the next pattern component.

Every pattern function which succeeds may be called again to test for its next implicit alternative. In this case it expects as its input *state* the *state* it last returned. Hence the *state* variable can be used to store information between alternatives. Should a pattern function need to store more information between alternatives than can be kept in one *state* variable, it may *push state* variables into *patternstack*. Such a function will expect the stack to be unchanged when it is called for its next alternative, hence any pattern function which fails must remove any items it has stored in *patternstack*.

As an example of a pattern function, Figure 2 shows *len*, which implements LEN (the SNOBOL4 primitive

Author's address: Computer Centre, University of Auckland, Private Bag, Auckland, New Zealand.

Fig. 1. Global declarations for pattern matching.

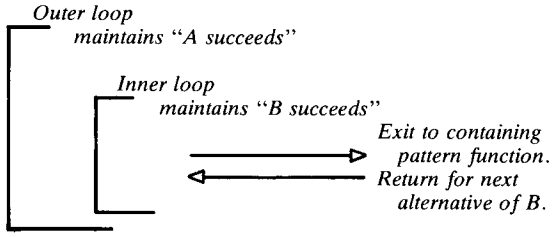
```
type string = record
        length : integer;
        body : array [0 . . length] of char;
        end;
     state = record
        succeed : Boolean;
        cursor, alternative : integer;
        end;

var subject : string;
    patternstack : array [integer] of state;
    psp : integer;

procedure push(p : state);
  begin psp := psp+1; patternstack [psp] := p; end;

procedure cut;
  begin psp := psp−1; end;

function topofstack : state;
  begin topofstack := patternstack[psp]; end;

procedure pop(var p : state);
  begin p := patternstack [psp]; psp := psp−1; end;
```

Fig. 2. Sample pattern function.

```
function len(p : state, l : integer) : state;
  begin
    if ¬ p.succeed then begin
      if subject.length − p.cursor + 1 ≥ l then begin
        len.succeed := true;
        len.cursor := p.cursor + l;
        end
      else len.succeed := false;
      end
    else len.succeed := false;
    end;
```

Fig. 3. Concatenation of pattern functions A and B.

(a) Structure of *cat*.



(b) Program for *cat*.

```
function cat(inp : state) : state;
  var p : state;
  begin
    p := inp;
    if p.succeed then goto 5;
    repeat
      p := A(p);
      if p.succeed then begin
        push(p);
        p.succeed := false;
        5: p := B(p);
          if p.succeed then goto 9;
        pop(p);
        end;
      until ¬ p.succeed;
  9: cat := p;
    end;
```

matching any string of N characters). When *len* is first invoked *p.succeed* is false, so if sufficient characters remain in *subject*, *len* succeeds. If *len* is invoked again, *p* brings back the last value of *len*, i.e. *p.succeed* is true. Since LEN has no implicit alternatives, *len* fails.

3. Operations on Patterns

Two of the most common operators used in building up patterns are concatenation (e.g. A B means "match A, then B") and alternation, (e.g. A | B means "match A, if A fails match B").

The structure of *cat*, an algorithm for concatenating pattern functions A and B is shown in Figure 3(a). Its outer loop repeats as long as A succeeds, i.e. it terminates when there are no further implicit alternatives for A. The inner loop does the same for B. When both A and B have succeeded we have found one match for A B, so we exit from *cat* to continue with the function which called it. Should we need to retry *cat* we must return into the inner loop for B's next alternative.

Figure 3(b) gives a program for *cat*. This uses a working *state* variable *p* for interacting with A and B. The outer loop is a simple repeat loop, which saves its current value of *p* in *patternstack* while executing the inner loop. Exit from the inner loop is implemented by the statement "goto 9"; return into it by "goto 5". The inner loop keeps its value of *p* between retries in the function value (as did *len* in Section 2).

cat could be implemented as a function with procedures A and B as parameters allowing us to handle patterns like A B C as (A B) C. Note, however, that A's repeat loop can be duplicated as many times as required, allowing us to create a *cat*-like function with any given number of pattern arguments.

Alternation is more difficult to handle, since every alternative of each pattern component must be tried before moving on to the next component. *alt*, an algorithm for A | B is shown in Figure 4(a). This pattern has two components. The outer loop tries them in turn, using its inner loop to test their implicit alternatives. When a successful alternative is found we exit from *alt*, making provision to continue the inner loop should we need to retry.

The program for *alt* (Figure 4(b)), is complicated by the fact that three items of information must be saved between retries. These are *altnbr* (the number of the current pattern component), *newp* (the state variable for the current implicit alternative), and *p* (the initial value of *newp*). When *alt* is first invoked *inp.succeed* is false, so we push *inp* (as *p*) and set *altnbr* to zero. The outer repeat loop retrieves this initial *p* for each pattern component. When the inner repeat loop finds a match it *push*es *newp* and exits ("goto 9"), storing *altnbr* in the function value. A retry recovers these values (as *p* and *altnbr*) before returning into the inner loop. When *alt* finally fails the initial *p* is *cut* from the stack.

As with *cat*, an *alt*-like function can be created with

any given number of pattern arguments.

When a match is complete, *patternstack* contains a history of the match. It is a simple matter to put extra items in the stack for other purposes, such as remembering which parts of the pattern were to be assigned to other strings following a successful match. Storing items in *patternstack* is similar in effect to "threading the beads" of a SNOBOL4 bead diagram. As each pattern component is matched ("bead is threaded") items are added to the stack. If a component finally fails, its stack items are removed ("the needle is withdrawn"). Failure to cut the stack back properly will ruin the pattern matching – this corresponds to "cutting the thread."

Pattern functions can be used in a re-entrant manner – for example, to implement

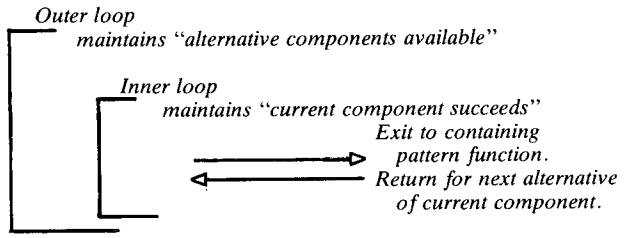
D = '0' | '1' ; B = D D D | D.D | D

we would use *alt* for D and B, and *cat* for the first two components of B. In Figures 3(b) and 4(b) the working variables (*p*, *newp*, *altnbr*) are local, but since they are all saved between retries they might usefully be global.

Recursive use of the pattern functions described above is impossible for two reasons. First, they have no way of breaking recursive loops. To provide this we could include an integer *depth* in *state*, set *depth* equal to *cursor* when a function succeeds, and increment it by one when invoking further functions. Functions would then fail when *depth* exceeded *subject.length*. Second, recursive invocations may store an unknown number of items in patternstack. To cope with this we would need to pass the stack height (value of *psp*) on entry to a function, probably as another integer in *state*, so the function could restore the stack reliably when it failed.

Fig. 4. Alternation of pattern functions *A* and *B*.

(a) Structure of *alt*.



(b) Program for *alt*.

```
function alt(inp : state) : state;
  var p, newp : state; altnbr : integer;
  begin
    if ¬ inp.succeed then begin
      p := inp; push(p); altnbr := 0;
      end
    else begin
      pop(p); altnbr := inp.alternative;
      end;
    repeat
      repeat
        case altnbr of
        0 : newp := A(p);
        1 : newp := B(p);
          end;
      if newp.succeed then begin
          push(newp);
          newp.alternative := altnbr:
          goto 9;
          end;
        until ¬ newp.succeed;
      altnbr := altnbr + 1 ; p := topofstack;
      until altnbr = 2;
      cut;
  9: alt := newp;
    end;
```

4. Conclusion

This implementation of SNOBOL4 patterns is used in the SNOBOL4-like language "Snobal/67" [1], which has an execution speed varying between 3 and 18 times faster than our Burroughs B6700 implementation of SNOBOL4. Unfortunately our SNOBOL4 has no official support, so this performance may be suspect.

As outlined above, the overheads in implementing fully recursive patterns are high: for this reason Snobal/67 does not allow them. To compensate for this Snobal/67 has ARBNX(P), a pattern primitive which matches pattern P as many times as possible. ARBNX is defined as

ARBNX(P) = *P P|P

where * is SNOBOL4's "unevaluated expression" operator.

The algorithms presented above are straightforward to use. Though they are described in a version of Pascal it is possible to implement them in almost any high-level language. This can be particularly useful in producing pattern-matching systems (like Snobal/67) where the emphasis is on execution speed and simplicity of implementation, rather than on providing the full generality of SNOBOL4.

Acknowledgments. I would like to thank my referees for their extremely helpful suggestions for the revision of this paper.

Received January 1976; revised September 1976

References

1. Brownlee, J.N. Snobal/67 – A Burroughs B6700 compiler for a dialect of SNOBOL4. Tech. Rep. 1, Comptr. Centre, U. of Auckland, New Zealand, 1975.
2. Dewar, R.B. SPITBOL – Version 2.0. SNOBOL4 Doc. S4D23, Illinois Inst. of Tech., Chicago, 1971.
3. Gimpel, J.F. A theory of discrete patterns and their implementation in SNOBOL4. *Comm. ACM 16*, 2 (Feb. 1973), 91–100.
4. Griswold, R.E. *The Macro Implementation of SNOBOL4.* W.H. Freeman, San Francisco, 1972.
5. Griswold, R.E., Poage, J.F., and Polonsky, I.P. *The SNOBOL4 Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., Sec. Ed., 1971.
6. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language PASCAL. *Acta Informatica 2* (1973), 335–355.
7. Wirth, N. The programming language PASCAL. *Acta Informatica 1* (1971), 35–63.