F. PATTERN AND STRING OPERATIONS

This section describes the "elemental" set of pattern and string operators provided in patternForth, and how these may be used to construct any of the pattern or string operators commonly provided by pattern matching languages.

This will also describe how single-cell string and pattern identifiers can be used with Forth VARIABLEs, and as "associated data," to provide the functionality of string and pattern variables.

1. Basic string operators

In addition to the string construction and string literal functions previously described, patternForth provides six basic functions for operating on strings.

a) SIZE - returns the length of a string; a null string has a SIZE of 0. Equivalent to the SNOBOL4 SIZE.

b) CONCAT - concatenates two strings. SNOBOL4 does not use an explicit operator for this, as concatenation is indicated by simple juxtaposition.

c) SUBST - extracts a substring of a given length, at a given starting point. (There seems to be no explicit SNOBOL4 function to perform this; instead, it is done as a pattern match with replacement.)

d) FILLED - creates a string of a given length, filled with a given character. The SNOBOL4 equivalent is accomplished with DUPL.

e) $>N - converts a string to a binary number, following the rules of Forth's numeric conversion: signs and decimal points are not legal, and illegal characters merely terminate the conversion. Accomplished in SNOBOL4 with CONVERT.

f) N>$ - converts a binary number to a string. Also accomplished with CONVERT in SNOBOL4. Distinct operators are required in patternForth because Forth is an untyped language.

2. Derived string operators

These SNOBOL4 string functions can be "derived" from the basic set of string operators.

a) RPAD - to pad a string with spaces on the right, can be accomplished by CONCATenating a string FILLED with spaces. The size of this blank string can be calculated from SIZE.

b) LPAD - to pad a string with spaces on the left, is the same as RPAD except for the order of the concatenation.

c) TRIM - to remove trailing blanks, requires the use of pattern matching to identify the non-blank substring, and SUBST to extract it.

d) DUPL - to "duplicate" a string multiple times, can be done by CONCATenating in a loop.

3. String variables

A string variable is a fixed identifier that can reference (or "hold") a variable string text. This would be used, for example, to program a pattern match for an input string which will assume many values.

This may be accomplished in patternForth by using a Forth VARIABLE to hold a string descriptor. In the example given above, each string, when input, is stored in string space and assigned a unique string descriptor. This string descriptor is stored in the Forth variable. The pattern matching routine obtains its subject string by fetching the contents of this variable.

Another way would be to store this "variable" string descriptor in another string; i.e., in the associated data field of another, known string.

4. Basic pattern operators

PatternForth provides five basic functions which can be used in the construction of a pattern.

a) MATCH - matches a fragment of string text exactly. This can be used to match a single character. This corresponds to the inclusion of a string literal or string variable in a SNOBOL4 pattern.

b) ANY - matches a single character to any within a character set, specified by a string. This is the SNOBOL4 ANY.

c) LEN - matches a string having any contents, of a given length. Equivalent to the SNOBOL4 LEN.

d) ARB - matches a string of any length and any contents; attempts to match the largest possible string. Equivalent to the SNOBOL4 ARB.

Note in the source listing (Appendix B) that ARB is itself a pattern, which employs a Forth looping construct. This is an example of using a "private context" across alternatives. In this case, the "private context" is a decrementing loop counter, and the alternative always succeeds, causing ARB to successively match smaller and smaller strings.

e) POS - matches a given cursor position. Corresponds to the SNOBOL4 POS.

In addition, the current cursor position is always available from the variable CURSOR, and the subject string length from the variable EOS.

5. Derived pattern operators

Most of the other pattern functions provided by SNOBOL4 can be derived from the five given above.

a) REM - to match the remainder of the string, can be derived by using CURSOR and EOS to calculate its length, and using LEN. This calculation is used frequently enough that it is as a Forth word; REM is simply LEFT LEN.

b) TAB - to match all characters up to a given position, can be derived from that position, CURSOR, and LEN.

c) RPOS - to match a specific cursor position measured from the right end of the subject, can be translated to a POS by using EOS.

d) RTAB - likewise, can be translated to a TAB by using the EOS value.

e) NOTANY - to match any character NOT in a given set, can be constructed by applying Forth's logical inversion operator to ANY, i.e., ANY 0=.

f) ARBNO - to match zero or more consecutive occurences of a subpattern, can be constructed in the same manner as ARB, using a Forth loop around an alternative construct.

g) SPAN - results from applying ARBNO to ANY.

h) BREAK - results from applying ARBNO to NOTANY.

i) FAIL - failure is forced by putting a 'false' (0) on the stack.

j) SUCCEED - which always succeeds, and always causes a backtrack to proceed forward again, can be implemented by placing an always-true function inside an infinite alternative loop. For example,

```
<< BEGIN 1 | AGAIN >>
```

When executed "forward" for the first time, this will succeed and stack an alternative. If backtracking ever reaches this point, the "alternative" will succeed and processing will move forward once again; the same "alternative" will be restacked for the next backtrack.

k) unanchored searches - can be satisfactorily performed by prefixing a pattern with ARB. This is not exactly equivalent to a SNOBOL4 unanchored search; what is really needed here is a version of ARB which attempts to match the smallest possible string. Such a word is, of course, easily written in the fashion of ARB.

If most of these operators are so simple of implementation, why have these implementations not been included in patternForth? There are four reasons:

* it saves Forth dictionary space by not implementing words which may not be used;

* it reduces the size of the source code and makes it more readable;

* it reduces the size of the lexicon that a new patternForth programmer must learn;

* perhaps most significantly, if these functions become important later, it will be preferable to write specific, optimized routines to perform them, rather than use these simple but less efficient constructs.

6. Capturing match results

At any point in the pattern match, the address, segment, and length of the subject string, and the current scan position, can be obtained from the Forth variables CURADR, CURSEG, EOS, and CURSOR.

By saving the value of CURSOR, in Forth variables, before and after a particular subpattern, the substring which was matched by that pattern can be identified. This substring may be extracted in two ways:

a) if a copy of the subject string descriptor was kept, SUBST may be applied to it with these parameters to create the "captured" substring; or

b) an operator similar to >$, but allowing source text from any 8086 segment (rather than just the Forth segment), could be created to use this address data directly.

7. String replacement

String replacement, at present, suffers from the lack of a simple operator and a straightforward syntax. It may be accomplished "manually" by capturing the match results, as described above, and then applying SUBST and CONCAT to:

a) extract the substrings before and after the matched text, and

b) combining this "prefix" and "suffix" with the substitute text.

8. Pattern variables

A pattern variable is a fixed identifier that can hold a changeable pattern. This changeable pattern can then be made part of a fixed pattern, or otherwise invoked from the precompiled program.

This is done in patternForth by storing the address of the pattern in a Forth VARIABLE. The address of a compiled pattern can be obtained with the Forth ' (tick) operator. This pattern is then later referenced by either

a) `variable-name @ EVALUATE`
to evaulate it as the "main" or entire pattern, or

b) `variable-name @ EXECUTE`
within another pattern, to evaluate it as a subpattern.

It is also possible to store the address of a pattern in the associated data field of a string; in this case, the $@ operator would be used.

Acquiring patterns from input data is more involved, since input data is text, and patterns are compiled Forth words. This conversion is "run-time compilation."

9. Run-time compilation

The ability to run Forth's compiler or "text interpreter" during an application program is a simple extension to Forth [WIN82]. Newer Forths [ANS89] include this feature in the Forth kernel.

Run-time compilation is needed to create patterns from input data, since patterns are compiled Forth words. It emulates two SNOBOL4 functions:

a) CODE - to compile a text string as program source code; and

b) EVAL - to evaluate an expression.

In patternForth, the difference between these two operators is basically whether or not the Forth compiling "state" is active. This is implementation dependent among Forths, but the usual operators are

] to set the "compiling" state

[to set the "interpreting" state

10. Future work

a) ARRAYs

ARRAYs in SNOBOL4 are similar to TABLEs, except that they take an integer index. Unlike a TABLE, it is possible to access all of the elements in an ARRAY, without knowing their contents. ARRAYs may dynamically expand, and ideally contract as well.

The ARRAY may be thought of as a special kind of TABLE, i.e., a set of string descriptors which are accessed in a different way. Two ideas for implementation are:

a) Store the array elements in 64 linked lists, whose headers are stored in a two level tree just like the root of a TABLE. The array index is factored into two parts; one to select one of the 64 lists, the other to determine position along the list.

b) Store pointers to the array elements in such a tree, but allow the tree to expand (grow a new level) as the number of elements exceeds 64. A three-level "root" tree would index 512 elements, and so forth.

The former method has the advantage of making ARRAYs physically resemble TABLEs, simplifying the conversion between the two forms. The latter method has the advantage of faster access to a given element, especially for large ARRAYs.

b) ABORT

The SNOBOL4 ABORT, when encountered anywhere in a pattern, immediately halts the pattern match without trying any more alternatives, and returns failure.

Such an operator in patternForth would require saving the state of the Return stack as well as the Data stack, which itself is not a novel operation [ROD89b].

c) FENCE

The SNOBOL4 FENCE prevents the pattern matcher from "backing up" past a given point in the pattern.

This could be implemented as a special kind of backtrack record. (It may also be possible to derive this from the existing patternForth functions, but how this would be done is still unclear.)

d) REPLACE

The SNOBOL4 REPLACE function, applied to a string, maps one character set to another.

This is a simple function to implement in machine code, using a 256-byte translation table to convert 8-bit character codes. It is not currently included in patternForth because is not expected to be useful with the binary data of control applications.

e) easier string replacement

The problem of replacing a matched substring with another string requires a significant effort on the part of the patternForth programmer. It would be useful to devise a Forth-like syntax for string replacement, and to implement this as a simple operator in patternForth.