AN EXPERIMENTAL APPLICATIVE PROGRAMMING LANGUAGE
FOR LINGUISTICS AND STRING PROCESSING

P.A.C. Bailes and L.H. Reeker
Department of Computer Science, University of Queensland,
St. Lucia, Queensland, Australia 4067

Summary

The Post-X language is designed to provide facilities for pattern-directed processing of strings, sequences and trees in an integrated applicative format.

Post-X is an experimental language designed for string processing, and for the other types of operations that one often undertakes in computational linguistics and language data processing.

In the design of Post-X, the following four goals have been foremost:

(1) To modernize the Markov algorithm based pattern matching paradigm, as embodied in such languages as COMIT[13] and SNOBOL[8];

(2) To provide a language useful in computational linguistics and language data processing, in particular, but hopefully with wider applicability;

(3) To provide a vehicle for the study of applicative programming, as advocated by Backus[1], among others;

(4) To provide a vehicle to study the application of natural language devices in programming languages, as advocated by Hsu[9] and Reeker[12].

The "X" in "Post-X" stands for "experimental", and is a warning that features of the language and its implementation may change from one day to the next. The eventual goal is to produce a language designed for wide use, to be called "Post" (after the logician Emil Post). In this paper, we shall present some of the language's facilities for string and tree processing. A more detailed statement of the rationale behind the language can be found in [2], and more details of the language are to be found in [3].

Pattern Matching

The basic idea of using pattern matching to direct a computation is found in the normal algorithms of Markov, and was embodied in the early string processing language COMIT. The series of SNOBOL languages developed at Bell Laboratories, culminating in SNOBOL4, improved a number of awkward features of COMIT and added some features of their own. Among these latter was the idea of patterns as data objects.

Post-X incorporates patterns into an applicative framework, which will be illustrated below. In doing so, the powerful pattern matching features of SNOBOL4 have been retained, and in fact, improved. In an applicative framework, the pattern match must return a value that can be acted upon by other functions. The pattern itself has been generalized to a much more powerful data object, called the FORM.

A FORM consists of a series of alternative PATTERNS and related ACTIONS. Each pattern is very much like a pattern in SNOBOL4 (with some slight variations). FORMS may be passed parameters (by value), which are then used in the pattern or action portion.

A PATTERN determines the structure of the string to which it is matched. The pattern contains a sequence of concatenated elements, which are themselves PATTERNS, PRIMITIVE PATTERNS (utilizing most of the SNOBOL4 primitives) or STRINGS. The value returned by the pattern is either FALSE (if it fails to match) or a "parse tree" designating the structure of the string that corresponds to portions of the pattern. As an example, suppose that a pattern is $P := p_1 \char`^ p_2 \char`^ ... \char`^ p_n$. It may be matched to a string $S = s_0 s_1 ... s_n$ by the use of the operator "in", and if each of the p_i match a successive letter s_j, one can conceptualize the "tree" returned as
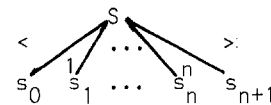
where s_0 represents the unmatched portion to the left of the matched portion and s_{n+1} the portion to the right of the matched portion.

The numbers $1,...,n$ and the characters < and > in the example are SELECTORS, used in the ACTION portion to refer to the appropriate substring. The tree returned is denoted by $\$\$$, and ! is used for selection, but $\$\$!$ can be condensed to $\$$ in this context, so the expression $\$ <$ returns s_0 in the example above, while $\$ 2$ returns s_2. The selectors give the effect of the short persistence variables that were found in COMIT, where they were denoted by numerals. These variables had the advantage of having their scope limited to a single line of the program, thus minimizing the number of variables defined at any one time. In Post-X, the selectors are local to a particular FORM.

Each of the s_i may be a subtree, rather than a string. In the example above, if p_1 were defined as $p_{11} \char`^ p_{12} \char`^ ... \char`^ p_{1n}$, then in place of s_1 would be the subtree
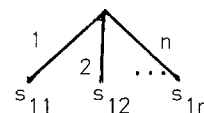
where s_{11} is the portion matched by p_{11} etc.
Then s_{11} would be referenced by $ 1 ! 1.

Composition of functions is often necessary. For the composition of F and G, we write F:G. For example

head:sort

where "head" gives the first element of a sequence and "sort" sorts a sequence of strings into alphabetical order, defines a function which operates on a sequence of strings and gives the first in alphabetical order.

Certain natural variations in the syntax are permitted. For example

$expression_1$ op $expression_2$

is defined to be the same as

(op):[$expression_1$, $expression_2$].

The existence of a conditional function

cond [a,b,c]

producing "b" or "c" depending on "a" is vital; Post-X allows for a multi-way branch of the form

 if
   $condition_1$ then $expression_1$
 elif
   $condition_2$ then $expression_2$
 elif
    ⋮
 else expression$_n$
 fi

the form

 do sequence of names
  expression
 od

is an expression whose value is the function which may be applied to a sequence of epxressions, the value of each of which is given in the expression (in the do ... od) by the corresponding name (i.e. call-by-value). The value of the application is the value of the expression (with "substituted" parameters). User-defined functions are named by declarations, examples of which are given later, and defined in[3].

We have already discussed the basic pattern matching operation and the definition of the FORMS used in that operation. As may be apparent from that discussion, context

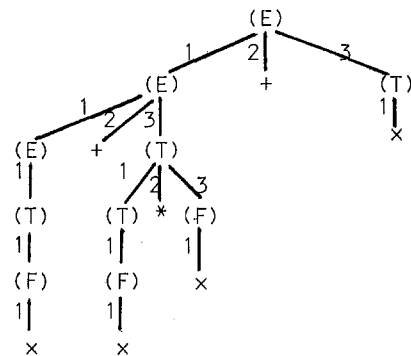free parsing creates no difficulties.

Thus we may define

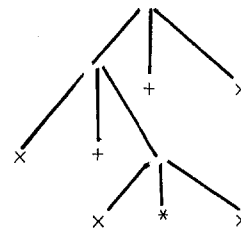E:=E^"+"^T | T

T:=T^"*"^F | F

F:="("^E^")" | "x"

Then the pattern match

E = "x+x*x"

will return the tree

The default action is to produce an unlabelled tree "compressed" by the elimination of single offspring nodes. The example above is given by the tree (pictorially).

or the sequence

[["x","+",["x","x","x"]],"+","x"]

If a labelled phrase marker is desired, then appropriate actions need to be attached. For instance, if a parenthesized representation of the tree above with node labels added is desired, the forms would be:

E:=E^"+"^T {"E["^$1^",+,"^$3^"]"}
 | T {"E["^$1^"]"};
T:=T^"*"^F {"T["^$1^",*,"^$3^"]"}
 | F {"T["^$1^"]"};
F:="("^E^")" {"F[("^$2^")]"}
 | "x" {"F[x]"};

In the example above, the application of E would return

"E[E[E[T[F[x]]],+,T[T[F[x]],*,F[x]]],+,T[x]]"

Translation to a prefix representation of the arithmetic expression could, incidentally, be accomplished by a slight change in the actions:

```
E:=E^"+"^T      {"+"^$1^$3}
    | T;
T:=T^"*"^F      {"*"^$1^$3}
    | F;
F:="("^E^")"    {$2}
    | "x";
```

This time, E="x+x*x" would yield

++x*xxx.

Context sensitive - and even more complex - parsing can be effected by building programs into the actions to give the effect of augmented transition networks.

It should also be noted that the actions need not merely pass back a single value. Several values may be associated with a node, as in attribute grammars[10]. For example

```
E:=E^"+"^T{
        value := $1.value + $3.value;
        code  := $1.code^$3.code^" add"
      }
    | T;

T:=T^"*"^Fα
        value := $1.value + $3.value;
        code  := $1.code^$3.code^" times"
      }
    | F;

F:="("^E ")"
      {value := $2.value;
       code  := $2.code
      }
    | span ("0".."9")

      {value := $1;
       code  := " "^$1};
```

(As in SNOBOL4, a numerical string in a numerical context is treated as a number.)

Reference to the attributes of a node may be made in several ways. For example, in the last grammar given, (E = "1+2*3").value would have the value 7, as would value (E="1+2*3") or (E="1+2*3")!"value"; and (E+"1+2*3").code would be evaluated as "1 2 3 times add".

If it were considered desirable immediately to evaluate the expression (returning 7 as the value of the match in the example above) we can write this in the action portion:

```
E:=E^"+"^T {$1 + $3}
    | T;
T:=T^"*"^F {$1 * $3}
    | F;
F:='('^E^')' {$2}
    | span (0..9) {$1};
```

Certain predefined patterns and pattern-returning functions are available, being closely modelled on those of SNOBOL4 e.g.[3]

```
    any    string
    arb
    break  string
    span   string
    arbno  string
                etc...
```

Two Examples

1. Random Generation of Sentences

Given a context-free grammar as a sequence of rules in BNF notation (i.e. left and right hand sides separated by "::=", nonterminals surrounded by angle brackets), we wish to randomly generate sentences from the grammar. We shall assume for simplicity that a pseudo-random number generator RANDOM is available which generates a number in an appropriate range each time that it is called. We assume also that the grammar is a string of productions, separated by ";" and is called GRAM.

The program will utilize a form LHS_FIND to find a production with a particular left hand side and return its right hand side.

```
LHS_FIND LHS := LHS^"::="^BREAK";"^";"{$3};
```

The alternatives on the right hand side are then converted to a sequence by the pattern ALT_LIST:

```
ALT_LIST := BREAK "|"^"|" {[$1]^(ALT_LIST < $)>}
            | REM {[$1]};
```

The particular alternative on the right hand side is chosen by the procedure

```
SELECT_RHS LIST := LIST ! ((RANDOM MOD
                        SIZE(LIST))+1);
```

The replacement in the evolving sentential form is accomplished by

```
REPLACE GRAM := "<"^BREAK">"^">"
                {$<^((REPLACE GRAM)<
                    SELECT_RHS
                    (ALT_LIST<
                    (LHS_FIND $2 <GRAM)))
                ^$>}
                |NULL{$$};
```

This form finds an occurrence of a nonterminal
(it will find the leftmost as it is applied
below). It uses this nonterminal as a
parameter to LHS_FIND, which is applied to
the grammar GRAM to return the right hand
alternatives. Then SELECT_RHS selects an
alternative, which is placed in the context
of the nonterminal matched by the pattern
portion of the form. Finally, REPLACE is
matched (recursively) to the result. If
the first alternative fails, it means that
there is no nonterminal. In that case, the
second alternative will be matched, and will
return the entire string, which will be a
string in the language generated by GRAM.

The entire program will be invoked as

RAND_STRING GRAM := (REPLACE GRAM) <"<S>".

This assumes that the root symbol is <S> .

It should be noted that the parentheses
can be reduced by using the transformation
available in Post-X into postfix notation,
with the postfix composition operator ".".
Using this, one version of REPLACE_GRAM
would be:

```
REPLACE GRAM := "<"^BREAK">"^">"
                {$<^GRAM.
                   (<(LHS_FIND $2).
                   (<)ALT_LIST.
                   SELECT_RHS.
                   (<)REPLACE_GRAM
                   ^$>}
                |NULL{$$};
```

The freedom that this alternative notation
provides is one of the refreshing aspects
of Post-X. (The particular transformation
applied here to REPLACE_GRAM is, incidentally,
analogous to extraposition in English.)

2. A KWIC Index of Titles

It is assumed that the input consists
of a sequence of titles. It is desired to
provide a primitive alphabetized KWIC index.
An input of ["An analysis of the English
present perfect" "The role of the word in
phonological development"] will produce
["<An> analysis ..." "An <analysis> ..."
"... in phonological <development>" ... etc.]

The top-level program to do this is

```
(1)  KWIC :=
(2)       UNION ALPHA [(<)[PARTITION]].
(3)       ALPHA [(in) TAG_FRONT]].
(4)            SORT.
(5)            ALPHA [(<)[REMOVE_TAG]];
```

Line (2) applies the form PARTITION
to each string in the sequence. PARTITION
will "tag" each word in each string, by
producing a copy of the string with angle
brackets around the word, creating
[" An <analysis>...", "An <analysis> ..."
"An analysis <of> ..." ... etc.] A sequence
is produced for each string in the original
sequence, and these are merged to form a single
sequence by the UNION function.

Line (3) adds an occurrence of the "tagged"
word to the beginning of each string. Line (4)
sorts the sequence obtained (SORT is a built-in
function), and Line (5) removes the word added
to the beginning of the string in line (3).

The forms PARTITION, TAG_FRONT, and
REMOVE are defined as follows:

```
PARTITION :=
         SPAN('A'..'Z')^' '
     {['<'^$1^'>'^$2^$>]^,
      ALPHA:CAT[$1] (PARTITION<$>)}
     |SPAN('A'..'Z') {'<'^$1^'>'}
```

The SPAN matches the first word, and the action
part of the form places that word ($1) between
angle brackets. The PARTITION<$ portion of the
action returns a sequence of PARTITIONS of
the rest of the string, and the first word is
concatenated onto the beginning of each of
these. When only one word remains, the second
alternative is used.

```
TAG_FRONT:=
       '<'^BREAK'>'
     {$2^' '^@$$}
```

The @$$ is the whole string (the "flattening"
of tree $$ to a string), the $2 is the portion
in angle brackets. A copy of this is moved to
the beginning of the string.

```
REMOVE_TAG:=
       SPAN('A'..'Z') ' '
       {$>}
```

This merely removes the string added at the
beginning of the sentence by removing the
first word.

Tree Processing in Post-X

Tree processing facilities are pattern-
directed and similar to string processing
facilities. As pointed out earlier, labelled
trees may be represented as strings containing
brackets. The pattern BAL, carried over from
SNOBOL4, is used to match a full, well-formed
subtree, but is extended to allow a
specification of the value of that tree. Some
tree functions are added for use in forms.
The function FUSE fuses a sequence of subtrees
together at their top node, leaving the label
unchanged. In a tree form, as illustrated

below, the use of a text string refers to a subtree with that label, but the function LABEL will return the label itself. The function RELABEL (tree, name) changes the label on a subtree to that named.

Post-X tree processing facilities are currently undergoing a careful study, and may be changed, but their present capabilities can be illustrated by the specification of forms for two linguistic transformations in English, EQUI_NP_DELETION and THERE_INSERTION:

EQUI_NP_DELETION:=TREE("NP"^BAL^"S"^BAL("NP"^
          BAL)
                    {If $1=$4!1 then
                     $1^$2^$3^$4!2}

THERE_INSERTION:=TREE("S"^BAL("NP"^BAL^"V"^
          BAL^ARB)
                    {$1^FUSE("there"^$2!3^
                     $2!2^$2!4))

These can be compared to a "conventional" formulation :

EQUI_NP DELETION: X NP Y [NP Y] Z
                        S   S

Structural Index   1, 2, 3, 4, 5, 6

Structural Change 1, 2, 3, Ø, 5, 6

         where 2=4

THERE INSERTION: X [NP V 4] Z
                      2   S
Structural Index   1, 2, 3, 4, 5
Structural Change 1, THERE+3, 2, 4, 5

     Illustrating the application of these,

THERE_INSERTION in [S[NP[Det[A]N[boy]]
                     VP[V[is]PP[Prep[in]
                     NP[Det[the]
                     N[garden]]]]]]
[S[there V[is]NP[Det[A]N[boy]]
     PP[Prep[in]NP[Det[the]
          N[garden]]]]]].

     Tree forms are not as natural as they might be, and changes can be expected, with the major goal being to make their use as closely analogous to that of strings as possible.

Sequences

     Post-X has a number of facilities that apply generally to sequences of items. These have been illustrated in the examples by, for instance, the operator ALPHA, which applies a function to each element of a sequence, and UNION, which takes a sequence of sequences and creates a single sequence.
     Less obvious, perhaps, is the fact that strings and unlabelled trees are themselves

sequences; in fact, sequences form the unifying notion within Post-X data structures. There remains work to be done to determine how effectively one can generalize pattern directed processing to sequences without adding too much complexity to the language.
     The LISP programmer will tend to identify sequences with lists, in the sense of that language. There are differences, however. The facilities in LISP are oriented toward lists as right-branching binary trees, and though one can build LISP functions to overcome this, the programmer generally must manage the lists with their links in mind. In Post-X, the programmer is encouraged to deal with the sequence directly, as one becomes accustomed to dealing with strings directly in a string processing language.

Discussion

     The SNOBOL4 language has had few competitors over the years as a general string-processing language. Its development and distribution were undertaken by Bell Laboratories, and thus it has been widely available for more than a decade. Yet SNOBOL4 has never been as widely used for large applications, including those in computational linguistics, as one might expect, and in the light of a decade's experience, it is possible to identify various difficulties that account for this.

     The basic string processing operation of SNOBOL4, pattern matching, is not the source of these difficulties. In fact, SNOBOL4 pattern matching provides a high-level-data-directed facility that should be a standard for other languages. The major problems were in the fact that the pattern-matching was never sufficiently generalized, leading to a "linguistic schism"[7], that the syntactic conventions of SNOBOL4 led to difficulties and poor structuring[5], and that the necessity of constantly assigning values to string variables was clumsy and tended to obscure the semantics of all but the simplest programs

     Recently, various attempts have been made to remedy the problems mentioned above. But to give up pattern matching, as in Icon[7] or to resort to a less rich vocabulary of patterns, as in Poplar[11](despite which, the latter language has a good deal of merit), is to "discard the baby with the bathwater". Post-X is the result of attempts to extend pattern matching and to improve it, at the same time providing a more natural, flexible and comprehensible linguistic vehicle.

References

1. Backus, John [78], Can programming be
    liberated from the Van Neumann style?.
    A functional style and its algebra of
    programs, CACM vol.21, no.8, August,1978,
    613-641.

2. Bailes, P.A.C., and Reeker, L.H.[80], Post-
    X: An Experiment in Language Design for
    String Processing, Australian Computer
    Science Communications. Vol.2, no.2, March,
    1980, 252-267.

3. Bailes, P.A.C., and Reeker, L.H. [80], The
    Revised Post-X programming language,
    Technical Report no.17, Computer Science
    Department, University of Queensland.

4. Galler, B.A., and Perlis, A.J. [70],
    A View of Programming Languages, Addison-
    Wesley, Reading, Massachusetts.

5. Gimpel, J.F. [76], Algorithms in SNOBOL4,
    John Wiley, New York.

6. Griswold, R.E. [79], the ICON Programming
    Language, Proceedings,ACM National
    Conference 1979, 8-13.

7. Griswold, R.E., and Hanson, D.R.[80], An
    Alternative to the Use of Patterns in
    String Processing, ACM Transactions on
    Programming Languages and Systems,Vol 2,
    no.2, April, 1980, 153-172.

8. Griswold, R.E., Poage,J.F., and Polonsky,
    I.P. [71], The SNOBOL4 Programming Language,
    Prentice-Hall, Englewood Cliffs, New
    Jersey.

9. Hsu, R. [78], Grammatical Function and
    Readability: the syntax of loop constructs,
    Proceedings, Hawaii International Conference
    on Systems.

10. Knuth, D.E. [65], On the translation of
    Languages from left to right, Information
    and Control, vol.8, no.6, 607-639.

11. Morris, J.H., Schmidt, E., and Walker, P.
    [80], Experience with an Applicative String
    Processing Language, Proc.Sixth ACM Symp.
    on Principles of Programming Languages.

12. Reeker, L.H.[79], Natural language devices
    for programming readability: embedding and
    identifier load, Proceedings of the 2nd
    Australian Computer Science Conference,
    University of Tasmania, 160-167.

13. Yngve, V. [58], A programming language
    for mechanical translation, Mechanical
    Translation, vol 5, no.1, 25-41.