SETL Newsletter No. 198

Robert B.K. Dewar
January 18, 1978

String Primitives

This newsletter contains a suggestion for introduction of string primitives into SETL.

The suggestions derive from SNOBOL-4, but lack (deliberately) the complexity of this language's pattern matching facility. In particular, they do not contain any imbedded backtracking. In those rare cases where backtracking matches are desired, the suggested functions can be used in conjuction with the existing backtracking primitives to provide the required effect.

The basic notion is a set of function calls of the form:

function(subject, parameter)

this call performs a match at the start of the subject string using the supplied parameter. The value returned is a pair:

[Newsubject, Matched string]

Newsubject is the string resulting after removing the matched substring, and matchedstring is a copy of the matched substring.

If the match fails, then newsubject is the same as subject and matched string is set to om.

The set of functions is as follows, the semantics being those obvious from their SNOBOL-4 ancestry:

```
SPAN (subj, string)
BREAK(subj, string) /* also breaks on end of string */
LEN(subj, integer)
ANY(subj, string)
NOTANY(subj, string)
```

The following additional matching functions are available:

```
STR(subj, string) /* matches for literal string */
NSPAN(subj, string) /* like SPAN but allows null */
RLEN(subj, string) /* takes chars from end of subject */
/* also RSPAN, RBREAK, RANY, RNOTANY, RNSPAN, RSTR */
```

The following additional functions are also provided

```
DUPL(string, integer)
RPAD(string, integer, char)
LPAD(string, integer, char)
```

These are implemented as in SPITBOL.

Some examples:

1) Read fixed length numeric fields

```
MACRO FLD(CARD, N);
      EXPR
        [CARD, TEMP]  :=  LEN(CARD, N);
        YIELD DEC TEMP;   END;
ENDM;
     READC(CARD);
     N1 = FLD(CARD, 3);
     N2 = FLD(CARD, 5);
     .
     .
     .
```

Perhaps FLD should be a predefined macro

2) SCAN ASSEMBLY FIELDS

```
[card, label]  := break(card, "^");
[card]  :=  span(card, "^");
[card, opcode]  :=  break(card, "^");
[card]  :=  span(card, "^");
[card, operands]  :=  break(card, "^");
```

3) A SNOBOL pattern to recognize $a^n b^n c^n \mid d^*$

```
      S(SPAN('A') $ A
.     SPAN('B')   $ B   * EQ(SIZE(A), SIZE(B))
.     SPAN('C')   $ C   * EQ(SIZE(A), SIZE(C))
.     |    SPAN('D'))   RPOS(0)   :F(NOGOOD)
```

In SETL

```
bracktrack  s;
if ok then           /* opens match */
if ok then           /* alternative */
     [s,a] := span (s, "a"); if a = om then fail; end;
     [s,b] := span (s, "b"); if b = om then fail; end;
     if # a ≠ # b then fail; end;
     [s,c] := span (s, "c"); if c = om then fail; end;
     if # a ≠ # c then fail; end; succeed;
else
     [s,d] := span (s, "d"); if d = om then fail; end;
end;
     if s ≠ nulc then fail; end;
     succeed;
else
     goto nogood;
end;
```

```
/* SOME MACROS TO IMPROVE THAT */

      macro     try (func, subj, param);
                expr
                     [subj, temp]  := func(subj, param);
                     if temp = om then fail; end;
                     yield  temp;
                end;
      endm;
      macro     match;
                     if ok then;
      endm;
      macro     endmatch;
                end;
      endm;
      macro     alternatives;
                if ok then;
      endm;
      macro     or;
                succeed;
                ok then;
                elseif
      endm;
      macro     endalts;
                succeed;
                else fail;
      endm;
      macro     failure;
                succeed;
                else
      endm;
```

Now we have

```
backtrack  s;
match
     alternatives

          a := try (span, s, "a");
          b := try (span, s, "b");
          if # a ≠ # b then fail; end;
          c := try (span, s, "c");
          if # a ≠ # c then fail; end;
     or
          try (span, s, "d");
     endalts;
     if s ≠ nulc then fail; end;
failure
     goto  nogood;
endmatch;
```

There is another possible coding style for the string primitives, namely

```
matched string := function(subject, parameter);
```

Here we assume that the function modifies its first argument.

There is considerable debate at the moment as to whether procedures should be allowed to modify their arguments. The two possibilities being considered are:

1. Call by value: In this case the formal parameters of a procedure are treated like read only variables.
2. Call by value with value return: This is similar to call by value except that the parameter are assigned back to the arguments after the call.

   The call

   A := P(B, C) with value return is equivlent to

   [A, B, C] := P(B, C);

without value return.

Value return is somewhat more elegent however it can be very costly for procedures which do not modify their arguments. For example

```
y := sin(f(g(x)));
```

is very expensive if value return is allowed. One might provide a repr "read only parameter" however this repr is hard to check.

For the moment we are implementing call by value. However the string primitives will change if value return is added.