Paul McJones

Recent Changes to CAL SNOBOL

During the past spring and summer (1971), substantial work has been done to our SNOBOL processor. Much of this consists of internal housecleaning intended to facilitate future extensions and is not outwardly visible. However a number of new features have been added, which are described below. Four areas are affected: the compiler, storage management, the interpreter, and standard procedures.

1. The compiler. Part of the compiler has been rewritten to make the installation of detailed syntax error messages easier. As of 7 October 1971 the messages themselves are not installed, but should be shortly. Here are the external changes:

   (i) Source lines longer than 72 characters (which is the number of columns actually processed by the compiler) appear in the listing with 10 spaces inserted between the 72-nd and 73-rd characters of the original lines. (As before, characters past the 90-th, if any, will not be listed.)

   (ii) Listing directives are now available to control the appearance of the source listing. A directive is any line with a minus (-) in column one. Starting in column two should appear one of the words SPACE, EJECT, NOLIST, or LIST (with no leading or interspersed blanks). A directive of this form will not itself be listed, but will have the following effect:

SPACE	lists one blank line;
EJECT	forces the new line of the listing to begin a new page;
NOLIST	stops the listing of subsequent source file lines and disables SPACE and EJECT directives;
LIST	resumes the listing, enabling SPACE and EJECT directives.

   Any other directive line will be treated as a comment; i.e., it will be listed but otherwise ignored.

   (iii) Two outstanding bugs have been fixed. Blanks are no longer lost from the end of a line containing a string literal which is continued to the following line. The listing never ends with a heading line and an otherwise empty page, which used to happen if the source file contained an exact multiple of 56 lines.

(iv) The requirement that a real constant begin with a digit has been dropped, so the syntax is now

⟨real constant⟩ ::= ⟨digit string⟩ •⟨digit string⟩ | ⟨digit string⟩ •| • ⟨digit string⟩

(v) While formerly any one string constant could contain quotes only of the type not used to delimit the constant, now a quote of the delimiting type may be represented by a pair of such quotes. Thus ### is equivalent to ↑≠↑ , and ↑↑↑↑↑↑ is equivalent to ≠↑↑≠, and so on. (Note that neither change (iv) nor (v) affects programs using the old conventions.)

2. Storage management. Users of arrays and especially program-defined data structures will be glad to hear that the space used by these objects is now reclaimed when ~~no longer in use.~~ This change is visible only to the extent that programs which used to run the field length up high should need less space now. There is no COLLECT() procedure à la Bell Labs since garbage collection automatically occurs when necessary.

3. The interpreter. The label END need no longer appear at the end of the program labeling an otherwise empty statement. This label is now considered, like RETURN, FRETURN, and NRETURN, to be system defined. Furthermore, any of these four labels may be used as an ordinary label in the program, thereby losing the system definition. Thus old programs using an END statement will actually terminate execution by "falling off the end of the program", and this is no longer an error but a legitimate way to quit.

Extra arguments to a procedure are now evaluated and ignored. This happens whether the procedure is program-defined or a standard procedure, data constructor, or data selector (field). (In Bell Labs SNOBOL it is an error to call a standard procedure with too many arguments.) As before, missing arguments are assumed to be null strings.

Another change is that program-defined procedures need not have any formal parameters. Thus the first argument to DEFINE() (the procedure prototype) must evaluate to a string of the form:

⟨procedure prototype⟩ ::= ⟨procedure name⟩ (⟨formals list⟩) ⟨locals list⟩

where

⟨procedure name⟩ ::= ⟨identifier⟩

and ⟨formals list⟩ ::= ⟨identifier list⟩ | ⟨empty⟩

and ⟨locals list⟩ ::= ⟨identifier list⟩ | ⟨empty⟩

and ⟨identifier list⟩ ::= ⟨identifier list⟩ ,⟨identifier⟩ | ⟨identifier⟩

Analogous to the last-mentioned change: objects of program-defined data-type (called data structures hereafter) may now have as few as zero fields. (The old minimum was one field.) Thus the single argument to DATA() (the data prototype) must evaluate to a string of the form:

⟨data prototype⟩ ::= ⟨data name⟩ (⟨field list⟩)

where

⟨data name⟩ ::= ⟨identifier⟩

and ⟨field list⟩ ::= ⟨identifier list⟩ | ⟨empty⟩

4. Standard procedures. Section 3 mentioned changes to the previously existing standard procedures DEFINE() and DATA(); also the procedures INPUT(), OUTPUT(), and DETACH() have been made more tolerant. The first two now work on variables which are already attached -- they just remove the old I/O association before making the new one. Similarly, DETACH() no-ops when applied to a variable with no association.

The names of some existing standard procedures have been changed to be compatible with Bell Labs: COMPILE() is now called CODE(); the names of the two procedures TIME() and CLOCK() have been interchanged. Thus TIME() now returns the number of milliseconds used by the job so far, while CLOCK() returns the time-of-day, as 12:59:01.

The following are some new standard procedures and their definitions.

(i) APPLY(f, x_1, x_2, ..., x_n) is like the Bell Labs APPLY(). The first argument f must be a string and correspond to the name of some currently existing procedure, be it program-defined, standard, data constructor, or field. APPLY() calls the procedure named by f with the arguments x_1, x_2, ..., x_n. If f expects fewer arguments the extra x_i are thrown away; if f expects more arguments the deficit is made up with null strings. If f RETURNs, APPLY() RETURNs the value f returns, and similarly if f NRETURNs or FRETURNs.

(ii) ITEM(a, s_1, s_2, ..., s_n) is like the Bell Labs ITEM() only better. The argument a must be any array. If the s_i are all integers and a is n-dimensional, then ITEM() does an NRETURN of the name of the element of a whose subscripts are s_1, s_2, ..., s_n. This is the same as

performing T = a, where T is some otherwise unused variable, and then using T[s_1, s_2, ..., s_n]. Now for the good part: each of the s_i is allowed to be a single integer or a string consisting of integers separated by commas. Starting with i=1, ITEM() scans the s_i until enough integers to satisfy the dimensionality of a have been gathered. If the s_i are used up, ITEM() assumes null strings (each with value zero, of course); extra s_i are ignored. Since ITEM() always NRETURNs, it may be used anywhere an explicit array reference is permitted. Also like an explicit array reference, ITEM() FRETURNs if any of the subscripts are out of bounds.

(iii) TYPE(x) is equivalent to DATATYPE(x) unless x is a data structure. Then TYPE() returns the string DATA, while DATATYPE() returns the ⟨data name⟩ string which was used in the prototype given to DATA() when the type of x was defined.

(iv) PROTOTYPE(q) is like the Bell Labs PROTOTYPE() as applied to arrays and data structures, but has been extended to patterns and names. The application of PROTOTYPE() to an array results in a string of the form:

$$l_1{:}u_1,l_2{:}u_2, \ldots, l_n{:}u_n$$

where l_i(u_i) is the lower (upper) bound of the i-th dimension of the array. The l_i and u_i are in canonical form for integers in SNOBOL; i.e., positive numbers are represented without a sign, no leading "0" characters are present, and the number zero is represented by the single character "0". The application of PROTOTYPE() to a data structure results in the ⟨data prototype⟩ string which was given to DATA() to define the datatype of q.

If q is a pattern, then PROTOTYPE(q) returns one of a fixed set of pattern prototypes, depending on what kind of pattern q is. These strings are in the form of ⟨data prototype⟩s and to further the analogy between patterns and data structures, a set of psuedo field functions, corresponding to the field names in the pattern prototypes, have been implemented. These procedures allow a pattern to be decomposed into the objects originally used to build the pattern.

If q is an ARB, REM, BAL, FAIL, ABORT, or FENCE pattern, PROTOTYPE(q) returns the string ARB(), REM(), BAL(), ..., or FENCE(). Obviously no pseudo field functions exist for these patterns.

If q is a LEN, POS, RPOS, TAB, RTAB, ANY, NOTANY, SPAN, BREAK, or ARBNO pattern, PROTOTYPE(q) returns the string:

x(PARAM)

where x is LEN, POS, ..., or ARBNO. The pseudo field function PARAM() may be used on any of these patterns; the following examples should serve to explain the result:

PARAM(LEN("3"))	returns the integer 3
PARAM(ANY("CBAC"))	returns the string ABC (note the order)
PARAM(ARBNO("CBAC"))	returns the string CBAC

If q is a concatenated or alternated pattern, PROTOTYPE(q) returns CAT(FIRST,REST) or ALT(FIRST,REST), respectively. The pseudo field functions FIRST() and REST() are defined on the class of concatenated and alternated patterns; here are some examples:

FIRST(≠A≠ ∨ ≠B≠ ∨ ≠C≠)	returns the string A
REST(≠A≠ ∨ ≠B≠ ∨ ≠C≠)	returns the pattern which is the value of the expression ≠B≠ ∨ ≠C≠
FIRST(ARB BAL REM)	returns an ARB pattern

If q is a naming pattern, p $ v or p . v , PROTOTYPE(q) returns the string DOL(LEFT,RIGHT) or PRD(LEFT,RIGHT), respectively. Thus we have the procedures LEFT() and RIGHT(). LEFT(p $ v) returns the pattern p. while RIGHT(p $ v) returns the name of the variable v.

Finally, if q is a deferred evaluation pattern, *v, PROTOTYPE(q) returns STAR(RIGHT) and RIGHT(q) returns the name of the variable v.

The fourth type of argument accepted by PROTOTYPE() is name. To aid the exposition, the concept of a family of variables is introduced. Each array is a family of variables, members of which are selected by an ordered set of integer subscripts. A second example of a family of variables is a data structure. A member of a data structure is selected by a field name (actually, the field function with the same name). The last example is the set of all simple variables, each of which is

selected by a non-null string. Note that 12 is the name of a simple variable (i.e., $12 or $"12") and could be the subscript of any number of array elements; no ambiguity exists since each similarly selected variable lies in a different family.

PROTOTYPE(q) returns one of the strings INDIRECT(RIGHT), ITEM(FAMILY, SELECTOR), or APPLY(SELECTOR,FAMILY) as q is a simple variable, an array element, or a data structure field. In the first case, RIGHT(q) returns the string corresponding to the simple variable named by q. Thus RIGHT(.VAR) returns the string VAR. In the latter two cases, FAMILY(q) returns the array or data structure object which contains the variable named by q, while SELECTOR(q) returns a string which can be used to select this variable in FAMILY(q). As examples of these functions, suppose the value of the variable A is a 2-dimensional array and the value of NODE is a data structure with a SON field. Then ITEM(FAMILY(.A[I,J]),SELECTOR(.A[I,J])) is equivalent to A[I,J] and APPLY(SELECTOR(.SON(NODE)),FAMILY(.SON(NODE))) is equivalent to SON(NODE).

(v) NEXTVAR(v) returns the name of the "next" variable in the same family as the variable named by v (which must be of type NAME or STRING). NEXTVAR() treats a family cyclically; the values of

```
NEXTVAR(v)
NEXTVAR(NEXTVAR(v))
...
NEXTVAR(NEXTVAR(... NEXTVAR(v)...))
```

will be different until all the members of family have been returned -- then the cycle repeats.

Note that NEXTVAR() for arrays and data structures could be written in SNOBOL using PROTOTYPE(), ITEM(), and APPLY(). The true power of NEXTVAR() is its ability to access the simple variables sequentially. As an example of its use the function CLEAR() of Bell Labs SNOBOL is written in CAL SNOBOL:

```
      DEFINE("CLEAR()")                  : (CLEAR.END)
CLEAR  CLEAR = .CLEAR
CLEAR.LOOP  CLEAR = NEXTVAR(CLEAR)
     CLEAR = IDENT(CLEAR, .CLEAR)        : S(RETURN)
     $CLEAR =                            : (CLEAR.LOOP)
CLEAR.END
```