November 24, 1970

SETL Newsletter #7 Dave Shields

This newsletter contains a number of suggestions for improvments and modifications of SETL. The numbering follows that of newsletter #3

OPERATORS UNDERLINED

(13) It is suggested that named operators be underlined and not indicated by a suffixed period, as at present. Thus and. is now and, or. is or, etc.

(14) COMMENTS OVERLINED. It is suggested that comments be indicated by overlines, that is, draw a line over a comment (interior spaces included).

(15) MACRO PARAMETERS. The only SETL macro feature currently defined is the macro block of the form

do macroname; ...block macroname; body; end macroname; .

We now allow parameters in the macro which can be lexical names. When the macro is expanded, the argument names are substituted for the names in the macro body. For example,

```
block macro(a,b); a= *a or *b; b=nl; end macro;

...
prev;
macro(c,d); next; ...
```

results in

```
prev; c=*c or *d; d=nl; next;...
```

THIS PROVISO IN TO AID IN THE CLARITY OF PROGRAMS

(16) ADD = TO EXTRACTION STATEMENT. Assignment statements of the form "<extractor> data;" are to be written "<extractor> = data;" if the extractor contains no asterisk (in which case it has a value)

(17) ABBREVIATED FORM FOR MATHEMATICAL UNION, INTERSECTION. The mathematical form of union, $\bigcup_{x \in a} f(x)$, which in SETL is $\{y,\ x \in a, y \in f(x)\}$, may be abbreviated $\{f(x)\ \underline{u},\ x \in a\}$. Similarly, the intersection may be abbreviated, using int for u.

(18) SEQUENCES. Observe that sequences be included in SETL; a convenient "sequence former" operator may be added. We interpret the sequence

1) $a_1, a_2, \ldots, a_i, \ldots$

to be equivalent to the SETL set

2) $\{<1,a_1>, <2,a_2>, \ldots, <i,a_i>, \ldots\}$.

Details on the sequence former operation will be forthcoming; for the present, SETL users are encouraged to use SETL sets of the form 2), where appropriate, to facilitate conversion to sequences.

(19) OPTIONAL CONVENTION TO INDICATE STATEMENT LABELS. Statement labels may be optionally set off in brackets ([,]) to improve readibility. Thus "label:next;" may be written "[label] next;".

(20) EXTENDED DEFINITION OF IMAGE-SET CONSTRUCTION. The image set construct, f(a), defined on page 23 of SETL notes, now requires that f be a variable whose value is a set (or function.) This restriction is to be removed, so that we only require that f is a SETL expression with a set (or function) as its value. For example, we can now write

go to $\{<a,lab1>, <b,lab2>\}$ (data) ;

(21) SUBROUTINE EXPRESSIONS. It is suggested that "process" expressions of type subroutine be added to SETL. That is, components of a SETL expression may be a process which is to applied to the associated argument list. For example, the statement

proc f d;c=d; (while pair c) <*,c>c;;return c;endf; [a];

defines a process of one argument, which is to applied to each of the elements of the set a .

(22) EXTENDED FORM FOR INTEGER RANGE RESTRICTIONS. It is suggested that the range restriction for iteration over a set of integers, which is now of form

1) $min \leq \forall j \leq max$

be extended to allow any of the following additional forms:

$min < \forall j \leq max$ $min \leq \forall j < max$ $max \geq \forall j \geq min$

$max > \forall j \geq min$ $max \geq \forall j > min$.

Note that the loop variable j assumes the values defined by the restriction from left to right; for example, $5 > \forall j \geq 2$ results in j=4,3,2, in that order.

(23) ITERATION OVER EMPTY SETS. Iteration over an empty set is allowed, in which case the corresponding block is not executed. Thus in the sequence

prev; (v∈nl) block; next; ,

control will pass to statement next after executing prev, ignoring block, since the set of values assumed by the loop variable v is empty.

If you have any comments on the proposed modifications of SETL or any new modifications, please see Dave Shields (room 423, CIMS, phone 460-7487).

We are trying to collect oft-used SETL procedures subroutines and functions into a library, so we can avoid constant recoding of the same common set operations and to establish standard names and calling sequences. If you have any routines which you think should be in such a library, please see Dave Shields.

Note that SETL is to be defined in three forms: a documentation version for use in specifying algorithms and writing them out; a keypunchable version, with a minimal character set, for input of SETL programs to a SETL processor; and a console version, for use of SETL from a terminal, which is to have a "dense" character set, so that SETL programs can be expressed in a small number of characters.

NEW MULTI-ASSIGNMENT STATEMENT

(24) The multi-assignment statement is of the form

<extractor>:=data ,

where data is a SETL expression whose value is a tuple, and where extractor contains names of SETL variables and associated expressions which define an association of each name with the corresponding part of data. For each name, there is a structural reference to a part of data; this reference is either defined implicitly by the position of the name within the extractor, or explicitly by the value of an associated SETL expression.

In its simplest form the extractor consists of a certain number, say k, of variables and some number (possibly zero) of minus signs, say n-k:

$<s1, s2, \ldots, sn>$.

Assume that data is an m-tuple and that the mth component of data is not an ordered pair (since in this case data is also an (m+1)-tuple). If si is a name, then assign as value to name the corresponding ith component of name if $i \leq m$, or Ω otherwise; if sn is a name, then view data as an n-tuple if $n \leq m$, and proceed as above, else assign the value Ω. If, on the other hand, sn is a minus sign, then we say that we have cut the extractor; in this case we view data as a (j+1)-tuple, where sj is the rightmost name in the extractor. That is, if the extractor is cut, then the rightmost name is assigned as value either a single component of data, or else Ω.

For example, <x y><a b c d> results in x=a; y = <b c d>; while the cut assignment <x y -><a b c d> results in x=a; y=b .

~~Optionally~~, names in the extractor ~~may~~ must be separated by commas, ~~to improve readibility.~~ Also, the extractor may have components of the form (expn)- , which are equivalent to e consecutive minus signs, where expn has a value the positive integer e. Thus the following are equivalent:

<x,-,-,-,y>, <x,3-,y>, <x,2-,1-,y>.

Note that in this simple form of the extractor, ~~all x x~~ the structural reference of a variable is determined implicitly by its position within the exptractor. To indicate explicitly the component of data that is to be referenced by a variable, we write name z. exp (read "name is exp"). In this case we require that exp have as value a nonnegative integer p. If p=0, then no assignment is made to name, but the extractor is cut. For example, both statements assignment

<x - y -><a b c d>, <x,v z.0,y><a b c d>,

result in x=a, y=c. Note also that explicit assignments do affect the positional value of implicit assignments; for example, in both of the assignments above, y references the third component, even though in the second assignment, no value is assigned to w.

The the names in the extractor are assigned values as follows:

(a) Evaluate the reference expressions for all explicit components of the extractor. Let maxexp be the maximum value so obtained, and minexp the minimum value.

(b) Determine the components of data referenced by each implicit component of the extractor. Let maximp be the maximum (rightmost) component of data so referenced, and minimp the minimum. ~~Let maxminus be the maximum component of data referenced by a minus sign.~~

(c) Let ~~maxvalue~~ m = max(maxexp,maximp); this is the rightmost component of data referenced by a variable. If the rightmost component of the extractor is a minus sign, or if minexp=0, then cut the extractor.

(d) If the extractor is cut, view data as a ($m+1$)-tuple; if the extractor is not cut, view data as a ~~max~~ m-tuple.

(e) Assign as value to each name in the extractor the corresponding component of data, where the number of components in data is determined by d().

For example, let data= <10,20,30,40,50>. Then
"<a - c> data" results in a=10,c=<30,40,50>;
<a - c -> data results in a=10, c=30, since the
extractor is cut by the last minus; <a,d z.4, c,e z.0> data
results in a=10, d=40, c=30, with no assignment to e;
if a=3 then <a z.2, b z.(a/3+2), -> data results
in a=20, b=30, where by a) all explicit references
are calculated before any assignments, so that the
value of a used in calculating the reference for b
is the value of a before the assignment to a
indicated the extractor is performed.

At most one name in the extractor may be
replaced by the symbol *, in which case the left-hand side of the
assignment statement becomes an operator having a value which is the
component of data referenced by *. Such
multi-assignment operators (with a *) have a value, and
may be elements of SETL expressions.

If a name occurring in an extractor is undefined or has a
set as its value, then we allow indexed assignments
to "name" of the form name(iexp) or name{iexp} .
If c is the component of data referenced by name,
then the indexed assignments are respectively equivalent
to name(iexp)=c and name{iexp} = c (c must be a set).
For example if data= <10, {20,30} ,40> and x= {<1,100>},
then <x(1)z.3 ->data results in x = {<1,30>}; and
<x{1}z.2 ->data results in x = {<1,20>, <1,30>} .

More complex 'nested' extractors will be allowed; details will follow.