SETL Newsletter Number 55 September 29, 1971

SETL suggestions and questions S. Finkelstein

1. Initialization can be useful for the compound operator. Possible format:

$$[\underline{op}: x \in s, \text{initexpr}]e(x) \circledast$$

which is initexpr $\underline{op}$ $e(x_1)$ $\underline{op}$ ... $\underline{op}$ $e(x_n)$ where s is $\{x_1,\ldots,x_n\}$.

The code equivalent of y=[$\underline{op}$:x s,initexpr]e(x) is y=initexpr; ($\forall x \in$ set)y=y $\underline{op}$ f(x); end $\forall$x;
The original option should, in any case, be retained.

Besides being useful in situations where obvious helpful initializations exist (e.g., [+:x $\in$ set,0]e(x) for summations, [$*$,x $\in$ set,1]e(x) for products) this form is somewhat advantageous where

i - $\underline{op}$ is not commutative;

ii - e(x) is not precisely the same sort of object as the result of the $\underline{op}$.

For example, if a is a set and s is a set of sets

$$a - [+:x \in s]x$$

is better rendered as

$$[-:x \in s,a]x.$$

A better example is the use of the SHARP function (pages 5, 9 and 11 of my "minimization of boolean functions" paper), which is greatly simplified by the revised compound operator described above.

2. The $\forall$ header should have the same "doing" clause option ~~as the while header. Possible format:~~

$\circledast$ more generally $[op, x_1 \in e_1, x_2 \in e_2(x_1),\ldots,x_n \in e_n(x_1,\ldots,x_{n-1})$
$C(x_1,\ldots,x_n)$, initexpr]

$$(\forall x_1 \in s_1, x_2 \in s_2(x_1), \ldots, x_j \in s_j(x_1, \ldots, x_{j-1}) \text{doing block}, \ldots, x_n \in s_n(x_1, \ldots, x_{n-1}))$$

where block would be performed after each time the value of x_j is changed (but not after it first receives a value from s_j). A doing clause may be included for each of the x_j's in the same header.

3. Wherever one can speak of $x \in s$, x restricted to variable names (e.g., $\forall$ headers, set formers, compound operators, quantified boolean expressions);x should be permitted to be any legitimate expression that can appear on the left side of an assignment statement. The effect would be to take whatever x is and set it equal to the member of s that ordinary would simply be put in the variable. Thus

($\forall$ <left, right> $\in$ set)

is the same as

($\forall x \in$ set) <left, right>=x;

4. The "when" clause in the "doing" clause is objectionable because:

a - The word "when" is misleading: "unless" would be better.

b - The "when" clause itself is exceedingly superfluous since (while C_1 when C_2) can be replaced by (while C_1 <u>and</u> <u>n</u> C_2) or, more simply (while C_1 - C_2)

Caveat: A little extra shuffling is required under circumstances which indicate that C_2 may be undefined if C_1 is not <u>true</u>.

5. Suggestion for a possible $\exists$ header

($\exists x \in$ set) block end $\exists$ x;

meaning the same as

```
copy = set; (while copy ne nl) x from copy;
     block* end while copy;
```

where by block* we mean block with every reference to set replaced by a reference to copy.

Generalizations would exist as with ($\forall x \in$ set); the two could even be in the same header.

6. If x is not a variable name and f is a 1-argument programmer-defined or built-in function then if f changes its argument the statement

```
y = f(x)
```

will, at present, cause an error condition to be raised. If x is a legitimate expression for the left side of an assignment statement there is an obvious (subject to an exceptional case discussed below) possible legal meaning to that statement - namely x is set equal to whatever f puts in its argument as if an assignment statement were involved. Similarly, of course, subroutines can be altered.

This would legitimize such statements as:

```
<A,B> IN P(K):      (where P is a set of tuples)
A{x} FROM S;        (where S is a set)
Y = <A,B> IS F(X);  (where F is a set of tuples)
```

Exceptional case: What should Y=F(A,P(A)); mean if F changes its arguments (i.e., which A should be used when a value is assigned (upon F's return) to P(A), the original A or the value F returns to its first argument)? The answer to this question should probably depend on how the compiler will generate code to send F the values of A and P(A).

Question: Right now how is the comparable problem of the assignment statement

<A,P(A)> = B;

handled?

7. The same notations

$t(i:j)$, $t(n:)$

(but not t_1+t_2, which has another meaning) should be available for sequences as well as for tuples. Also, it seems more reasonable for j in t(i:j) to be the last index desired rather than the number of indices desired starting from i.

8. Is the statement

x = <[A],B>;

now legal? It would mean

x = {<P,B>, P ∈ A};

Similarly x = <[A],[B]>; would mean

x = {<P,Q>,P ∈ A,Q ∈ B};

What about the legality of

<[A],B> = x; meaning
A = HD[X]; B = TL X;

If not, just what are the limitations of the square brackets? If so, should this be explicitly mentioned? This may be related to 6 - how about

x IN [A]; meaning the same as
A = [A] + x;

9. What about allowing the statements

A = <u>HD</u>[x]; B = <u>TL</u>[x]; (x a set of tuples)

to be generalized so that we can obtain (say) the set of 5th components of the tuples in x, or the set of tuples containing the 4th through 9th components of the tuples in x. (<u>HD</u> gives all the 1st components; <u>TL</u> gives the 2nd through the tuple's length.) For instance, x[i:j] could mean

$$\{t(i:j), t \in x\}$$

and similarly for x[i:]. Of course to get 5th components (say) x[5] is unsatisfactory, as is x[5:5].

10. If it were possible to get them without putting SETL into convulsions, pointers of some kind would be good to have.