SETL Newsletter 42 June 4, 1971

Revised conventions concerning tuples J. T. Schwartz

The conventions concerning the use of tuples in SETL are still somewhat confused, and this note will aim to clear them up.

A. Indexed objects, tuples. (This remark belongs more to the logical level than to the level the SETL user will normally encounter.) If n is an integer (remember that in SETL this is a particular kind of atom) and x is any SETL object, then the set {n,{x}} will be called 'x indexed by n'. In this note, we shall find it convenient to write x_n for this set; naturally, this notation is not part of the regular SETL syntax. Note that the object x and the integer n can be reconstructed from x_n in a unique way; n (which must be an integer) will be called the index of x_n.

A tuple is now defined to be a set $t = \{x_{i_1}\ldots,x_{i_k}\}$, in which no index occurs more than once. That element x to which the index i is attached is called the i-th component of t. Note that this allows tuples which are 'sparsely' populated, e.g. tuples with defined first and third components, but with no second component. A tuple of this kind is called irregular; normally the programmer will not use irregular tuples; such use will be inefficient, though the system will not forbid it. Internally, tuples will be maintained by keeping their components in (logically) contiguous memory locations, probably in the sort of 'range' discussed in Newsletter 39.

Note also that tuples are considered to be SETL objects of different type than sets. Some additional explanation of this point: the objects in SETL have types; and in fact programmer-defined object types will be allowed in a systematic way. This means that a 'purer' set theory would consider each SETL object to be really an ordered pair, whose first component is the object's type, and whose second component is what in SETL we consider to be

the object itself. Objects in SETL are equal if and only if both components of this hypothetical pair are equal. Thus the SETL three-tuple t whose components are x,y,z, and which SETL writes as <x,y,z>, would, in a purer set-theory, be regarded as a pair

$$\underline{\text{tupl}},\ \{x_1,y_2,z_3\}$$

while the SETL object tt that could be written in SETL as

$$\{\{1,\{x\}\},\{2,\{y\}\},\{3,\{z\}\}\}$$

would in our hypothetical purer set theory be regarded as

$$\underline{\text{set}},\ \{x_1,y_2,z_3\}\ .$$

Note that SETL provides conversions of type through the binary function as. Thus t might also be written tt as tupl; while tt might also be written t as set. Alternatively if we write

$$\text{ss = t;}\quad \underline{\text{type}}\ \text{ss} = \underline{\text{set}};$$

then ss is the same as tt; and if we write

$$\text{s = tt;}\quad \underline{\text{type}}\ \text{s} = \underline{\text{tupl}};$$

then s is the same as t.

This distinction of types should cause minimum inconvenience to the programmer. At the implementation level, it avoids the necessity for checking sets frequently to see if they are tuples.

B. Notations for tuples. If x,y,...,z are n SETL objects, then the notation

(1) $$t = \langle x,y,\ldots,z\rangle$$

denotes the n-tuple which, were it taken as a set, would be the set

$$\{x_1,y_2,\ldots,z_n\}$$

of indexed objects. The type of t in (1) is of course tupl. Given (1), then

(2) $$t(k)$$

denotes the k-th component of t. The component t(1) may be written as

(3) $$\underline{\text{hd}}\ t.$$

The notation

(4) t(i:j)

denotes the tupl whose components, for $1 \leq k \leq j$, are t(i+k-1). The notation

(5) #t

denotes the number of components of t. The notation

(6) tℓ t

is an abbreviation for

(7) t(2:#t-1)

Note that all of these notational conventions apply also to tuples some of whose compnents are undefined; though the use of such tuples may lead the unwary to surprises.

Similar notations are adopted for bit-strings and character strings. The same notations can be applied to SETL sequences, i.e., to sets s whose members are 2-tuples <k,x>, with no integer k occurring twice in s.

The multiple assignments

```
<a,b,c> = tuple
```

and

```
<a,b,c,-> = tuple
<a,-,b,-> = tuple,  etc.
```

retain their present syntactic form. The first of these is equivalent to the set of assignments

```
a = tuple(1); b = tuple(2); c = tuple(3:#tuple-2);
```

The second is equivalent to

```
a = tuple(1); b = tuple(2); c = tuple(3);
```

The same notations are available for bit strings, character strings, and sequences.

These conventions accord approximately with the suggestions made in newsletter 39, p. 6, and in 34, p. 1, p. 3,4, but with some variances in detail. The present newsletter however obsoletes the others in this regard.

To continue, note that

i. The 1-tuple <x>, which as a set would be $\{x_1\}$ or equivalently {{1,{x}}}, is distinct from the object x. This is the object that in newsletter 34, p. 1 was written as <u>just</u> x; however, this last notation is unnecessary, and is abolished.

ii. The object ∋x, for x a tuple, is simply <u>hd</u> x; the former notation is abolished.

iii. We use the notation

(8) tupl(n:)

rather than tupl(-n) as an abbreviation for tupl(n:#tupl-n+1). The same notation is used in connection with bit strings, character strings, and sequences.

iv. Tuple concatenation may be written using the '+' sign. A sequence may be converted to a tuple by writing

tupl = [+: 1 ≤ n ≤ #seq]<seq(n)>;

a tuple to a sequence by writing

seq = {<n,tupl(n)>, 1 ≤ n ≤ #tupl}; .

v. The iteration header

(9) (∀x ∈ tupl)

iterates over all <u>components</u> of a tupl, in increasing order. It is therefore different from

(10) (1 ≤ ∀n ≤ #tupl) .

The iteration

(∀x ∈ tupl) block(x);

may however be written as

(1 ≤ ∀n ≤ #tupl) block(x(n));

The set-former and quantifier notations that derive from (9) are allowed also.

All these notations are also made available for bit-strings and character-strings, though not of course for sequences (which are sets).

vi. We write x ϵ tupl for ∃y ϵ tupl|y eq x.
Similarly for strings.

vii. We write f[tupl] for the tuple

[+: 1 ≤ n ≤ #tupl] <f(tupl(n))> .

Similarly for character strings and bit-strings.

C. Deviations from earlier notations.

Note that <x,y,z> is an object entirely distinct from <x,<y,z>>. In general, the present notations for tuples are quite close to the earlier (and present) notations for sequences.

D. Functional application.

Suppose that f is a set, and x an object. Then f{x} denotes the set

(11) {if #tℓ y eq 1 then y(2) else tℓ y | type y eq tupl
and tℓ y ne Ω and hd y eq x}.

We then define f(x) as the quantity

(12) if (#f{x}) gt 1 then Ω else ∋f{x} ,

and f[x] as the set

(13) [u: f{y}, y ϵ x] .

We also define

f{x,y} to be identical with (f{x}){y},

f{x,y,z} " ((f{x}){y}){z} , etc.;

and

(14) f(x,y,...,z,w)

to be identical with

(15) (f{x,y,...,z})(w) .

All this accords closely with our present practices. The changes in the treatment of tuples lead however to some slight technical variances. Note e.g. that if f is a set of triples < x,y,z>, and if g is the corresponding set of pairs <x,<y,z>>, (which is quite different) then f{x,y} and g{x,y} happen to be the same.

Generalized notations such as f[x,y], [x] op [y], etc. are defined in accordance with the above, very much in the manner described in the SETL notes, pp. 26-27.

Selection operators (SETL notes pp. 24-25) are abolished; and assignment operators (p. 45) abolished also.