SETL Newsletter # 147

J.T. Schwartz
March 8, 1975

A syntactic Construct Useful for Checking Parameters.

Central utility processes of an operating system must constantly be on guard against bad parameters, whose careless use might cause them to abort (in GYVE terminology, 'terminate'). Information preventing such parameters from being accepted is supplied declaratively in GYVE, where as a matter of fact this whole problem is somewhat less severe than it is in SETLG, since the data objects being manipulated are not so totally dynamic in size and type. In the present newsletter, we shall suggest a syntactic mechanism facilitating checks on a variable's structural form. It should be noted that, although this mechanism is purely syntactic, and has no semantic implications, it embodies a different rule for the handling of illegal cases than SETL does; this permits large clumsy SETL test sequences to be written very much more conveniently and succinctly.

Central to the proposed mechanism is a Boolean valued 'check' operator having the form

(1) $\quad\quad var \mid pattern,$

where *var* is a variable, and *pattern* is the new syntactic construct which we aim to explain. It is helpful, before launching on a formal explanation, to give an example. To check that the value of a variable v is a set with not more than 100 elements, each of which is a pair consisting of an integer first component not exceeding 50 and a character-string second component with not more than 10 characters, we can write

(2) $v|(\#.)\ \underline{\ell t}\ 100\ \underline{\text{and}}\{|<|\ \text{int}.\underline{\text{and}}.\underline{\ell t}\ 50,\ \ \underline{\text{cstring}}.\underline{\text{and}}(\#.)\ \underline{\ell t}\ 10\}$

This is an expression of the form (1), which has the value <u>true</u> if all the structural conditions listed above hold, <u>false</u> otherwise. The general rules visible in this example are as follows: patterns are built recursively out of subpatterns using 'constructor operations'. Specifically, if p is a pattern, then

$$\{|p\}$$

is a pattern such that $s|\{|p\}$ is equivalent to $(\underline{\text{type}}\ s)\ \underline{\text{eq}}\ \underline{\text{set}}$ $\underline{\text{and}}\ (\forall v \in s\ |\ (v|p))$. Similarly, if $p_1,\ldots,p_n$ are patterns, $<|p_1,\ldots p_n>$ is a pattern such that $s|<|p_1,\ldots,p_n>$ is equivalent to $(\underline{\text{type}}\ s)\ \underline{\text{eq}}\ \underline{\text{tupl}}\ \underline{\text{and}}\ s(1)|p_1\ \underline{\text{and}}\ s(2)|p_2\ \underline{\text{and}}\ \ldots\ s(n)|p_n$ $\underline{\text{and}}\ (\#\ s)\ \underline{\text{eq}}\ n$. In addition:

> $p_1\ \underline{\text{and}}\ p_2$ is a pattern, with $s|(p_1\ \underline{\text{and}}\ p_2)$ equivalent to $s|p_1\ \underline{\text{and}}\ s|p_2$
>
> $p_1\ \underline{\text{or}}\ p_2$ is a pattern, with $s|(p_1\ \underline{\text{or}}\ p_2)$ equivalent to $s|p_1\ \underline{\text{or}}\ s|p_2$,

and so forth for the other booleans. Any boolean expression may be considered to be a pattern, since it returns a boolean value.Within such an expression, when it occurs in a pattern or subpattern, any SETL operator may be written with the symbol '.' as argument, the missing argument being the possibly implicit variable to which the pattern or subpattern applies. Note in connection with this rule that the pattern

(3) (#.) lt 50 and {| set.and (#.) lt 100}

describes a set of at most 50 elements each of which is a set of not more than 100 elements; the two occurences of '.' refer to different quanties since the first occurence of '.' is bound to the outermost pattern level which the second occurence of '.' is bound to a subpattern. Note also that in both (2) and (3) we have used the name xxx of a SETL type as a monadic operator, where 'xxx a ' abbreviates '(type a) eq xxx'.

In some cases it will be desirable to introduce an explicit name for the quantity designated by '.' in (2) and (3). We do this by writing

{*name* | p}

instead of {|p}, and <$name_1$ | p_1, $name_2$ | p_2,...,$name_n$ | p_n> instead of <|p_1,...,p_n>. In this second construct, any one of the names $name_j$ may be omitted, in which case the immediately following '|' will be omitted also unless j = 1. As an example of the use of this construct, note that

{<a|t, b|b ne a>}

designates a set of ordered pairs none of which has second component equal to its first component.

If p is a pattern, then

(4) [|p]

is a pattern such that $s|[|p]$ is equivalent to

(5) $\quad (\underline{\text{type}}\ s)\ \underline{\text{eq}}\ \underline{\text{tupl}}\ \underline{\text{and}}\ (\forall\, v(n) \in s|(s|p)).$

Within p in the construct (4), the symbol '.' may be used as a synonym for the 'v' of (5). If it is desired to introduce explicit names for the 'v' and 'n' of (5), one can write

(6) $\quad [name_1\ (name_2)\ |p]$

instead of (4).

Patterns of the type $<|p_1,\ldots,p_n>$ can be concatenated with patterns of the type (4), yielding patterns

$$<|p_1,\ldots,p_n> + [|p]$$

which impose n different conditions on the first n components of a tuple, and a fixed condition on all remaining components.

We stress once more that in checking a value against a pattern we proceed in a completely 'defensive' fashion, converting any operation that would otherwise be an illegal error termination into a check-operation value of <u>false</u>. It is this convention which gives the pattern-check operation particular utility.

Note finally that no operation invoked as part of a pattern-check is allowed to have any side effect.