SETL Newsletter Number 44 a — July 6, 1971

Comprehensive SETL Specifications — Kurt Maly

Basic objects: Sets , tuples and atoms; sets or tuples may have atoms, tuples or sets as members. Atoms may be

Integer. examples: 0, 2, -3

Real. examples: 2.5, -0.5 (but not .5)

Boolean strings. examples: 1b, 0b, 77o, 00b777

Character strings. examples: 'aeiou', 'spaces- '

Label. (of statement) examples: label:, [label:]

Blank. (created by function newat)

Note: Special undefined blank atom is .

Subroutine. Function.

Basic operations for atoms:

Integers: arithmetic: +, -, *, /, // (remainder), exp

comparison: eq, ne, lt, gt, ge, le

other: max, min, abs , random, bitr.

Examples: 5//2 is 1; 3 max -1 is 3; abs -2 is 2, bitr 5 is 101b,

Reals: arithmetic: +, -, *, /, exp

comparison: eq, ne, lt, gt, ge, le.

other: real log real, cos(x), sin(x).

x min y, x max y, abs y, floor x, ceiling x,

random x, bitr x.

Examples: 2.0 log 8.0 is 3.0, floor-2.5 is -3; ceiling 2.5 is 3;

bitr 2.5 is 10b; bitr 10b is 2.0.

Booleans: logical: and, or, not (or n),

// (exclusive or), - (is the

operation and n), / (is the

operation or n).

logical constants t (or true; shorthand for 1b);

f (or false; shorthand for 0b).

Character strings: conversion: dec, oct

Examples: dec '12' is 12; oct '12' is 10.

dec 12 is '12'; oct 10 is '12'.

Bit strings: comparison: eq, ne, lt, gt, ge, le.
(a ge b if a has a 1 wherever b has a 1; two strings are made to be of equal length by filling in zeros at the left.)

Strings (character or boolean):
+ (catenation), *(repetition), string(j), string(i:j), string(i:),#string, ∋ string, ∀x∈string, f[string], nulb, nulc (empty strings). string*integer becomes a hash of the string to give a bit-string of a length determined by the integer; hol string regards a character string as a bit string in some dense internal format; holl is the number of bits needed to represent one character. Thus the length of hol 'abc' is 3*holl.
hol bitstrings pads a bitstring with zeroes to the nearest even multiple of holl, and then performs the reverse conversion.

Examples: 'a' + 'b' is 'ab'; 2 * 10b is 1010b;
2*'ab' is 'abab', 'abc' (2) is 'b', 'abcdef' (2:3) is 'bcd', 'abcd' (2:) is 'bcd', #'abc' is 3, #nulb is 0, ∋'abc' is 'a', ∀x∈ 'abc' ranges over 'a', 'b', 'c', {<'a', 'b'>, <'c', 'd'>} ['ac'] is 'bd', 'b'∈'abc' is t, 'bc'∈'abc' is f.

General: Any two atoms may be compared using eq or ne:
atom a tests if a is an atom. To determine the type of an object the function type may be used. type x eq tupl tests x for being a tupl. (Others are: set, integer, tupl, bstring, cstring, label, blank, real, subroutine, function.) Conversion of type is done through the binary function as. e.g., x as tupl.
The built-in function pair tests an object for being a tuple of length 2.

Basic operations for sets:
n ∈ (nonmembership) ∈ (membership test); nl (empty set); ∋ (arbitrary element), # (number of elements); eq, ne (equality tests); ge, le, gt, lt (various inclusion tests); with, less (addition and deletion of element); lesf (ordered pair deletion).
pow(a) (set of all subsets of a);
npow(k,a) (set of all subsets of a having exactly k elements).
+ (or u) (union); *(intersection); - (difference);

// (symmetric difference)

Examples: a $\in$ {a,b} is t, a $\in$ nl is f, a n$\in${a,b} is f, $\exists$ nl is Λ, $\exists$ {a,b} is either a or b, # {a,b} is 2, #nl is 0, {b} with a is {a,b}, {a,b} less a is {b}, {a,b} less c is {a,b}, {⟨a,b⟩,⟨a,c⟩,d} lesf a is {d}; pow({a,b}) is {nl, {a}, {b}, {a,b}}. npow(2,{a,b,c}) is {{a,b}, {a,c}, {b,c}}. {a,b} + {c} is {a,b,c}; {a,b}*{b,c} is {b}; {a,b,c} - {b,c} is {a}; {a,b} // {b,c} is {a,c}.

Internal assignment operator: is (low right precedence, high left precedence)

Examples: if a+b is c*d is e gt 0 then ....
is equivalent to c=a+b; e=c*d; if e gt 0 then ....

Ordered pairs: ⟨a,b⟩ first and second component extractors are hd and tl.

Tuples: let x,y, ..., z be n SETL object, then t=⟨x,y,...,z⟩ denotes the n-tuple.

Operations on tuples: t(k); t(i:j); #t; hd t (or t(1)); tl t (or t(2:#t-1)); $\exists$t (or hd t); t(n:) (or t(n:#t-n+1)); t1+t2; f[t] (is [+:$\forall$x $\in$ t]⟨f(x)⟩); nult (null-tuple);

Iteration header: ($\forall$ x $\in$ t); (1$\leq\forall$n$\leq$#t);

Examples: let t(1) = 2; t(2) = 5; t(5) = 1; t(6) = 3;
t1(1)=10; t1(2) = 9;
f= {⟨10,11⟩,⟨9,8⟩,⟨5,3⟩};
t(2:4) is 5,.,.,1 ; #t is 6; t1+t is ⟨10,9,2,5,.,.,1,3⟩;
(period indicates that this component is undefined (Λ), period is not part of SETL).
f[t1] is ⟨11,8⟩; $\forall$x$\in$t ranges over 2,5,1,3;
1$\leq\forall$n$\leq$#t, t(n) ranges over 2,5,Λ,Λ,1,3;

Set-definition: by enumeration a,b,...,c

Set former:
$\{e(x_1, \ldots, x_n),\ x_1 \in e_1,\ x_2 \in e_2(x_1), \ldots, x_n \in e_n(x_1, \ldots, x_{n-1}) \mid C(x_1, \ldots, x_n)\}$.

The range restrictions $x \in a(y)$ have the alternate numerical form

$$\min(y) \leq x \leq \max(y)$$

when $a(y)$ is an interval of integers.

Optional forms include $\{x \in a \mid C(x)\}$, equivalent to $\{x, x \in a \mid C(x)\}$; and $\{e(x), x \in a\}$, equivalent to $\{e(x), x \in a \mid \underline{t}\}$.

Functional application: (of a set of ordered pairs, or a programmed, value-returning function)

$f\{a\}$ is $\{$if # tl y eq 1 then y(2) else tl y, $y \in f \mid$ type y eq tupl and tl y ne $\perp$ and hd y eq a$\}$ i.e., is the set of all x such that $\langle a,x \rangle \in f$

$f(a)$ is: if $\#f\{a\}$ eq 1 then $\ni f\{a\}$ else $\perp$, i.e., is the unique element of $f\{a\}$, or is undefined.

$f[a]$ is [u: $f\{y\}$, $y \in a$]

More generally,

$f\{a,b\}$ is $(f\{a\})\{b\}$; $f\{a,b,c\}$ is $((f\{a\})\{b\})\{c\}$; etc.

$f(a,b,\ldots,c,d)$ is $(f\{a,b,\ldots,c\})(d)$.

Constructions like $f\{a,[b],c\}$, etc., are also provided.

Examples: let $f=\{\langle 1,2\rangle, \langle 1,3\rangle, \langle 2,4\rangle\}$; $x=\{1,2\}$; $f\{1\}$ is $\{2,3\}$; $f(1)$ is $\perp$; $f(2)$ is 4; $f[x]$ is $\{2,3,4\}$;

The same notation can be used with operators, so that, for example

$[a]+1$ is $\{x+1, x \in a\}$; or tl[a] is $\{$tl x, $x \in a\}$;

Compound operator:

[op: $x \in s$]e(x) is $e(x_1)$ op $e(x_2)$ op... op $e(x_n)$, where s is $\{x_1,\ldots,x_n\}$.

This construction is also provided in the general form

[op: $x_1 \in e_1 x_2 \in e_2(x_1)$, ..., $x_n \in e_n(x_1,\ldots,x_{n-1}) \mid C(x_1 \ldots x_n)$], where the range restrictions may also have the alternate numerical form.

Examples: [max: x∈{1,3,2}] (x+1) is 4,
[+: x∈{1,3,2}] (x+1) is 9,
[+: 1≤i≤n]a(i) is SETL form of $\sum_{i=1}^{n} a_i$.

[op: while cond doing block] expn; (equivalent to:
v=Λ; times=0; (while cond doing block)
if times eq 0 then v=expn; times=1;
else v=v op expn;; if v eq Λ then quit;;
end while;)

[op: while cond1 when cond2 doing block] expn;

Quantified boolean expressions:

∃ x ∈ a | C(x) ∀ x ∈ a | C(x)

general form is

∃x₁∈a₁, x₂∈a₂(x₁), ∀x₃ ∈a₃(x₁,x₂),... | C(x₁,...,xₙ),

where the range restrictions may also have the alternate numerical form.

Search with assignment:

∃[x] ∈ a | C(x) has same value as ∃x∈a|C(x),
but sets x to first value found such that C(x) eq t.
If no such value, x becomes Λ.

Any number of variables attached to initial ∃ quantifiers may be placed in square brackets.

Alternate forms

min ≤ [x] ≤ max, max ≥ [x] ≥ min, max ≥ [x] > min, etc.

of range restrictions may be used to control order of search.

Conditional expressions:

if $bool_1$ then $expn_1$ else if $bool_2$ then $expn_2$... else $expn_n$.

Assignment and multiple assignment statements:

right-hand-side assignment statements:

a = expn;

left-hand-side assignment statements:

f{exp} = expn; is same as
f = {p∈f| (hd p) ne exp} u {<exp,x>, x ∈ expn};
f(exp) = expn; is same as f{exp} = {expn};
f(a,b) = expn; f{a,b}= expn; etc. also are provided.

In general this holds for all basic SETL retrieval operators (like tl, hd, etc.) and also for programmer-defined infix or prefix functions and for programmer-defined functions of zero arguments. (See also definition of functions.)

multiple assignment statements:

⟨a,b⟩ = expn; is same as a=hd expn; b=tl expn;
⟨a,b,...,c⟩ = expn;⟨a,⟨b,c⟩,...d⟩= expn; etc., are also provided.
⟨f(a),g{b}⟩ = expn; is same as
f(a) = hd expn; g{b} = tl expn;

Generalized forms:

⟨f(a), g{b,c},...,h(d)⟩ = expn;
⟨f(a),⟨g{b,c},h(d)⟩,..., k(e)⟩ = expn;
etc., are also provided.

Control statements:

go to label;
if $cond_1$ then $block_2$ else if $cond_2$ then $block_2$...else $block_n$;
if $cond_1$ then $block_1$ else...else if $cond_n$ then $block_n$;
then block1 if cond1 else if cond2 then block2...;
then block1 if cond1 but block2 if cond2...;

The iff statement:

```
iff  test?                          } header (the deepest-rightmost
   label, action;                   } descendant must be an action-node
                                    } followed by our ';')
test:  block; =cond;                } trailer (In the block of an action node
action: block;                      } the transfer statement: to name; may
end iff;                            } appear.)
                                    } or ';' or 'end iff test;'
```

To the lower-left of any test-node follows its positive-case descendant and to the lower right its negative-case descendant. Any descendant may be a test node again. And the condition itself may be substituted for the test node's name.
(e.g., iff (x gt 0) ?

Likewise the codes for an action node may be substituted if placed in parentheses. Action nodes may be preceded by iteration headers. The trailer may contain definition for nodes in the form "=expn", not occurring in the header. Those may be referenced from within any other definition.

Example:

```
iff  test?
     testl?, actl
     labl, (x gt y)?
          (y=x+1; to on;), act2;
test:  =a eq b;
testl:  a=b+5; =c lt d;
actl:  fn(xy); to on;
on:  x=3; /* no successor type indicated, so go to labl */
xy:  =a * b;
act2:  x=7; end iff;
labl: ....
```

At-blocks: (at label) block;
(instead of ';', there may be 'end;', 'end at;', 'end at label;'). The label to which this at-block refers must be in the routine which contains the at-block and it is enabled as target for the block by enclosing it in two sets of square brackets.

Iteration headers:
(while cond) block;
(while cond doing blocka) block;
(while condl when cond2) block; (equivalent to: (while condl)
if n cond2 then continue; block;)

(while cond1 when cond2 doing block1) block2;

$(\forall x_1 \epsilon a_1, x_2 \epsilon a_2(x_1), \ldots, x_n \epsilon a_n(x_1, \ldots, x_{n-1}) |$
$C(x_1, \ldots, x_n))$ block;

in this last, the range restrictions may have such alternate numerical forms as

$\min \leqslant x \leqslant \max$, $\max \geqslant x \geqslant \min$, $\min \leqslant x < \max$, etc.,

which control the iteration order, and $\forall x_1 \in a_1$ may have the form $\min < \forall x < \max$, etc.

Scopes:

The scope of an iteration or of an else or then block may be indicated either with a semicolon, with parentheses, or in one of the following forms:

end $\forall$; end while; end else; end if; etc.;
or: end $\forall x$; end while x; end if x; etc.
or: $(\forall x \epsilon a)$ til done; block done:...
(while cond) til done; block done:... etc.

Loop control:

quit; quit $\forall x$; quit while; quit while x;

and

continue; continue $\forall x$; continue while; continue while x;

Subroutines and functions (are always recursive)

To call subroutine:

sub(param$_1$, ..., param$_n$);
sub[a]; is equivalent to $(\forall x \epsilon a)$ sub(x);;

generalized forms

sub(param$_1$, [param$_2$,param$_3$],...,param$_k$)

are also provided.

To define subroutines and functions:

subroutine:

define sub(a,b,c); text; end sub;

return; - used for subroutine return

function:

definef fun(a,b,c); text; end fun;

return val; - used for function return

A function which performs a retrieval operation may contain storage blocks and load blocks (see also left-hand-side assignment statements).

(load) block
(load) block end; } normal return statement (i.e., return val;)

(store name) block;
(store name) block end;
(store name) block end name; } return statement as in subroutine (i.e., return;)

infix and prefix forms:

define a infsub b; text; end infsub;

definef a infin b; text; end infin;

define prefsub a; text; end prefsub;

definef prefun a; text; end prefun;

inverted form:

[; subroutine body; define subname (a,b,...);]

[; subroutine body; define subname(a,b...);-]

produces an immediate call subname(a,b...);

This also applies to function definition and makes them usable in expressions. For function, the degenerate form:[; function body;] (equivalent to [;function body; definef fname;-])

is available.

Name scopes:

Normally internal to main routine or subroutine, unless declared external.

Initial statements:

initial block;

External declarations:

external a,b,c,...; - refers to main routine

suba external a,b,c,...; - refers to subroutine suba

external (a,aa), (b,bb),...; - changes name

suba external (a,aa), (b,bb),...; - changes name

Local subroutines:

subname local; (occurring in subroutine or function S, makes all variables having same name in S and subname common).

subname local $name_1,...,name_k$; (used in S makes $name_1,...$ $name_k$ local to S and all other names are identified with those appearing in subname).

local;
local $name_1,...,name_k$; } (if S is directly embedded within subroutine subname then these two declarations are abbreviations of those two above.)

Macro blocks:

To define a block:

block mac(a,b,...); text; end mac;

inverted form:

[; body; block mac(a,b,...);]

[; body; block mac(a,b,...);-] (equivalent to:

[; body; block mac(a,b,...);]mac(a,b,...);

To use:

mac(c,d,...);

Dynamic compilation:

compile x;

Example: x='definef fm(a); return(5*a); end fm;';

y=compile x; z=y(5); will give z the value 25.

Input-output:

Unformatted character string:

er is end record character; input, output are standard i/o media; record (n,s); - reads till er character, from character n.

Standard format i/o:

read a; reads a set from input, in standard format

print expn; prints a set on output, in standard format