January 12, 1971
Jack Schwartz

IIC. Recapitulation of the basic parts of the SETL language.

In the present section, we recapitulate, in capsule form, the principal basic features of the SETL language. While this merely repeats information given in considerably more detail in the preceding section, it may be hoped that such a *precis* may serve as a useful brief reference for the reader.

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

Integer. examples: 0, 2, -3

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 Ω.

Subroutine. Function.

Basic operations for atoms:

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

comparison: eq, ne, ℓt, gt, ge, ℓe

other: max, min, abs

Examples: 5//2 is 1; 3 max -1 is 3; abs -2 is 2.

Booleans: logical: and (or a), or, exor, implies (or imp), not (or n)

logical constants t (or true, or 1b); f (or false, or 0b).

Character strings: conversion: dec, oct

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

Strings (character or boolean):

+ (catenation), * (repetition), first, last, elt (extraction)

len (size), nul, nulc (empty strings).

Examples: 'a' + 'b' is 'ab'; 2 * 10B is 1010B;

2 * 'ab' is 'abab', 2 first 'abc' is 'ab',

2 last 'abc' is 'bc', 2 elt 'abc' is 'b',

len 'abc' is 3, len nul is 0.

General: Any two atoms may be compared using eq or ne;

atom a tests if a is an atom.

Basic operations for sets.

ε (membership test); nℓ (empty set); ∋ (arbitrary element),

(number of elements); eq, ne (equality tests);

incs (inclusion test); 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).

Examples: a ε {a,b} is t, a ε nℓ is f, ∋ nℓ is Ω,

∋ {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} incs {a} is t.

pow({a,b}) is {nℓ, {a}, {b}, {a,b}}.

npow(2,{a,b,c}) is {{a,b}, {a,c}, {b,c}}.

Ordered pairs: <a,b> first and second component extractors are

hd tℓ; n-tuples <a<b,c,...,d>> = <a,b,c,...,d>

Examples: hd<a,b> is a, tℓ<a,b> is b,

hd<a,b,c> is a, tℓ<a,b,c> is b,c .

Note that <a,b> is identical with {{a},{a,b}}, so that for example {a} ε <a,b> is t while a ε <a,b> is generally f.

See also: extraction operators, generalized extraction operators, replacement operators, and multi-assignment statements.

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 ε a(y) have the alternate *numerical form*

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

when a(y) is an interval of integers.

Optional forms include {x ε a | C(x)}, equivalent to {x, x ε a | C(x)}; and {e(x), x ε a}, equivalent to {e(x), x ε a | t} .

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

f{a} is {tℓ p, p ε f | (hd p) eq a}; i.e. is the set of all x such that <a,x> ε f

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

f[a] is {tℓ p, p ε f | (hd p) ε a}, ie, the image of a under f.

More generally,

f(a,b) is g(b) and f{a,b} is g{b}, where g is f{a};

f[a,b] is {$\underline{t\ell}$ $\underline{t\ell}$ g, gεf|($\underline{hd}$q)εa $\underline{and}$ (($\underline{hd}$ $\underline{t\ell}$ q)εb)} .

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

<u>Compound operator:</u>

[$\underline{op}$: xεs]e(x) is $e(x_1)$ $\underline{op}$ $e(x_2)$ $\underline{op}$... $\underline{op}$ $e(x_n)$,

where s is$\{x_1,\ldots,x_n\}$.

This construction is also provided in the general form

[$\underline{op}$: $x_1 \varepsilon e_1, x_2 \varepsilon e_2(x_1),\ldots,x_n \varepsilon e_n(x_1,\ldots,x_{n-1}) | C(x_1\ldots x_n)$],

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

Examples: [$\underline{max}$: xε{1,3,2}](x+1) is 4,

[+: xε{1,3,2}](x+1) is 9,

[+: $1 \le i \le n$]a(i) is SETL form of $\sum_{i=1}^{n} a_i$.

<u>Quantified boolean expressions:</u>

∃xεa|C(x) ∀xεa|C(x)

general form is

$\exists x_1 \varepsilon a_1, x_2 \varepsilon a_2(x_1), \forall x_3 \varepsilon a_3(x_1,x_2),\ldots \quad | C(x_1,\ldots x_n)$,

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

Search with assignment:

$\exists [x] \varepsilon a | C(x)$ has same value as $\exists x \varepsilon a | C(x)$, but sets x to first value found such that C(x) eq t. If no such value, x becomes Ω.

Any number of variables attached to initial $\exists$ quantifiers may be placed in square brackets.

Alternate forms

$min \le [x] \le max$, $max \ge [x] \ge min$, $max \ge [x] > min$, etc.

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

Conditional expressions:

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

Generalized extraction and replacement operators; generalized multiassignments.

The *extraction operator* has the form

(1) $\langle part_1, \ldots, part_n \rangle$

where each *part* has one of the forms

name, name z expn, z expn, *, * z expn, -, n-, or exop or exop z expn, where *exop* is itself an extraction operator. *Name* may be a simple name or may be an *indexed name* of one of the forms name (exp), name {exp}, name (exp_1,exp_2), etc.

Each *expn* has an m-tuple of non-negative integers as a value. Such an operator associates a sequence of integers, called a *structural address*, with each name which occurs within it. Example: in the operator

<<a z 3, b> z <1,2>, *>

the sequence 1,2,3 is associated with a; 1,2,2 with b; and 2 with *. The asterisk * may be used as a name at most once in an extraction operator. The structural address $n_1,\ldots,n_k$ associated with a name (or with the "special name" *) by an extraction operator (1) determines the quantity that will be assigned to the name when (1) is used either in the form

<$part_1$,...,$part_n$> expr   (if * is used once as a name)

or in the form

<$part_1$,...,$part_n$> =expr   (if * is not used as a name).

Examples:

x := <*,-,iz<2,1>,w><a,<b,c,d>,e,f,g>

results in the assignments

x-a,i=b,w=<f,g>;

x=<*,-,iz<2,1>,w,-><a,<b,c,d>,e,f,g>

results in the assignments x=a,v=b,w=f.

The *replacement operator* has the form (1), where each part has one of the forms

exp r expn, exp r, exp, -, n-,

or is itself a replacement operator. At least one occurrence of r is required. Each *expn* has an m-tuple of non-negative integers as a value. Such an operator associates a structural address with each *exp* which occurs within it; the rules for calculating this address are the same as those applying to extraction operators. When a replacement operator is applied to a structure built up in nested fashion out of n-tuples, any element of the structure addressed by a structural address A is replaced by the *exp* to which A belongs.

Examples:

<x,y r 3,-><a,b,<c,d>,e> has the value <x,b,y,e>;

<x,y r 3><a,b,<c,d>,e> has the value <x,b,y>;

<x,y r<3,1>><a,b,<c,d>,e> has the value <x,b,<y,d>,e>.

Statements: are punctuated with semicolons.

Assignment and multiple assignment statements:

a = expn;

f{exp}= expn; is same as

f ={pεf|(hd p)ne exp}u{<exp,x>,xεexpn};

f(exp) = expn; is same as f{exp}={expn};

f(a,b) = expn; f{a,b}= expn; etc. also are provided.

<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$_n$;

Iteration headers:

(while cond) block;

(while cond doing blocka) block;

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

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

$$\min \leq x \leq \max, \quad \max \geq x \geq \min, \quad \min \leq x < \max, \quad \text{etc.},$$

which control the iteration order.

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 ∀; end while; end else; end if; etc.;
or: end ∀x; end while x; end if x; etc.
or: (∀x∈a) til done; block done:...
(while cond) til done; block done:... etc.

Loop control:

quit; quit ∀x; quit while; quit while x;

and

continue; continue ∀x; continue while; continue while x;

Subroutines and functions (are always recursive)

To call subroutine:

sub(param$_2$,...,param$_n$);

sub[a]; is equivalent to (∀x∈a) sub(x);;

generalized forms

$sub(param_1, [param_2, param_3], \ldots, param_k)$

are also provided.

<u>To define subroutines and functions:</u>

subroutine:

```
define sub(a,b,c); text; end sub;
return;      - used for subroutine return
```

function:

```
definef fin(a,b,c); text; end fun;
return val; -used for function return
```

infix and prefix forms:

define a <u>infsub</u> b; text; end <u>infsub</u>;

definef a <u>infin</u> b; text; end <u>infin</u>;

define <u>prefsub</u> a; text; end <u>prefsub</u>;

definef <u>prefun</u> a; text; end <u>prefun</u>;

<u>Name scopes:</u>

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

<u>External declarations:</u>

```
     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
```

Macro blocks:

To define a block:

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

To use:

```
do mac(c,d);
```

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

The following algorithm produces an action table for a general precedence parse. The input to the algorithm is assumed to be a set of ordered k-tuples, where a grammatical production A → BCD is represented as <A,B,C,D>. The procedure *unorder* converts a k-tuple to an unordered set, and is used to form the set of all characters of a grammar. The map *starts*{x} gives all syntactic types which can be the first term of a sequence into which x can be expanded; *ends*{x} those which can be the last term of such a sequence. The table produced contains the following values:

$t(i,j)$ = 1 if $i \doteq j$, 2 if $i \gtrdot j$, 3 if $i \lessdot j$;
= 0 if the relation between i and j is ambiguous;
= 4 if the sequence ij is ungrammatical.

The following program generates all permutations of n in lexical order. The next sequence after a given s_n is defined by the following rule: increase the last possible element by the smallest possible amount. That is, we find the last element s_j which is not part of a monotone decreasing "tail," interchange it with the smallest s_k with $k>j$ and $s_k>s_j$, and then place all the elements $s_{j+1},\ldots,s_n$ into ascending order. A signal is transmitted through "more" when the process restarts.

```
define f perm (n,more);
/*initialize if new*/
if n more then more=t;seq={<j,j>,1≤j≤n};return seq;;
/*if sequence is monotone decreasing, there are no more
    permutations. otherwise find last point of increase */
if n (n>∃[j]≥1|seq(j)ℓt seq(j+1)0 then more=f;
    returnΩ;end if;
/*then find the last seq(k) which exceeds seq(j) and swap */
find= n≥∃[k]>j|seq(j)ℓt seq(k);
    <seq(j),seq(k)> = <seq(k),seq(j)> ;
/*then rearrange all the elements after seq(j+1) into
    increasing order */
(j<∀k≤(n+j+1)/2) kk=n-k+j+1;
    <seq(k),seq(kk)> = <seq(kk),seq(k)>;end ∀k;
return seq; end perm;
```

```
definef prectab(gram);
characters = [u: xεgram] unorder(x);
starts = complete {<hd x, hd tℓ x>,xεgram};
ends = complete {<x,last x> , xεgram};
same = nℓ;(∀xεtℓ [gram]) (while pair x)
    <p,x>=x; <p,hd x> in same;;end ∀x;
small = {<hd x,y> ,xεsame, yεstarts{tℓ x}};
large = {<y,z>, xεsame, yεends{hd x},
    zεstarts{tℓ x}u{tℓ x}};
tabl = nℓ; (∀xεcharacters, yεcharacters)
<c(1),c (2),c(3)> = <yεsame{x}, yεlarge{x}, yεsmall{x}>;
tabl(x,y) = if # {1≤j≤3|c(j)}gt 1 then 0 else
    if 1≤∃[j]≤3|c(j) then j else 4; end ∀x;
return tabl;

definef unorder(tuple); t=tuple;set=nℓ;
while pair t) <*,t>t in set;;return set with t; end unorder;

definef complete reln; prectab external characters; r=nℓ;
(∀xεcharacters) set=reln{x}; todo=set;
(while todo ne nℓ) y from todo;
todo = todo u{zεreln{y}|n zεset}; set=set u reln{y};
end while; r{x}=set;end ∀x; return r; end complete;

definef last tuple; t=tuple;(while pair t) t=tℓ t;;
return t; end last; end prectab;
```