SETL Newsletter Number 13 — January 14, 1971

Additional miscellaneous algorithms — J. T. Schwartz

10. Various operations concerned with permutations.

10A. Cycle form of permutation (omitting unit cycles).

set = {j, 1 ≤ ∀j < n}; cycs = nl; (while set ne nl) beg from set;
cycl = nl;
cycl(1) = beg; (while perm(beg) ∈ set) beg = perm(beg);
cycl(#cycl+1) = beg;
set = set less beg; end while; if #cycl gt 1 then cycl in cycs;
end while;

10B. Same, including unit cycles. Replace final if by: cycl in cycs;

10C. Standardized cycle form of a permutation.

cycs = {minfirst c, c ∈ cycs}; (∀ c ∈ cycs) place(c) =
#{d ∈ cycs | d(1) gt c(1)};;
cyco = {<place(c), c>, c ∈ cycs}; definef minfirst c; k=1;
(2 ≤ ∀j ≤ #c) k = if c(j) lt c(k) then j else k;;
return {<if j ge k then j+1-k else n+1-k+j, c(j)>, 1 ≤ ∀j ≤ #c};
end minfirst;

10D. Integer sequence representing a permutation. Form cycle form cycs, including unit permutation, then standardized cycle form from this. Then order these in decreasing order of first elements. Parentheses can be dropped, and reconstructed by finding

is = nl; (1 ≤ ∀n ≤ #cyco) cy = cyco(n); (1 ≤ ∀m ≤ #cy) is (#is+1) =
cy(m);; end ∀n;

And the inverse of this:

cyco = nl; cyc = nl;least = is(1); (1 ≤ ∀n ≤ #is) if is(n) lt least then
cyco(#cyco+1) = cyc; cyc = nl; min = is(n); cyc(1) = is(n);
else cyc(#cyc+1) = is(n); end if; end ∀n;

10E. "In place" inversion of a permutation. Permutation given by perm(n)

```
todo = {n, 1 ≤ ∀n ≤ nn ; (while todo ne nl) start from todo;
now = start; next = perm(start);
(while perm(now) ∈ todo) ⟨next, now, perm(next)⟩
⟨perm(next), next, now⟩; todo = todo less now;;
perm(start) = now; end while todo;
```

10F. Multiplication of permutations in cycle form. Second Knuth algorithm. This exploits the fact that (abcd...ef) restricted to the set of letters appearing is product

(f⟶*) (e⟶f) (d⟶e) ... (a⟶b) (*⟶a), and that

(x⟶y) map = $\overline{\text{map}}$ means $\overline{\text{map}}$(x) = map(y), with all other values unchanged.

```
map = nl; (#cycs ≥ ∀n ≥ 1 | #cycs(n) gt 1)c = cycs(n);
mapstar = if map(c(1)) ne Ω then map(c(1)) else c(1);
(1 ≤ ∀i < #c) map(c(i)) = if map(c(i+1)) ne Ω then
      map(c(i+1)) else c(i+1);;
map(c(#c)) = mapstar; end ∀n;
```

10G. Multiplication of permutations in cycle form. First Knuth algorithm. This keeps set which might be leftmost occurrence of some variable.

```
cyco = nl; list = nl; tag = nl; (1 ≤ ∀n ≤ #cycs)
(1 ≤ ∀m ≤ #cycs(n)) ⟨list(#list+1), tag(#list+1)⟩ =
                                      ⟨cycs(m), f⟩;;
⟨list(#list+1), tag(#list+1)⟩ = ⟨cycs(1), t⟩; end ∀n;
```

```
(while 1 ≤ ∃[j] ≤ #list | n tag(j))
⟨start, current, tag(j)⟩ = ⟨list(j), list(j+1), t⟩;
cyc = {⟨1, list(j)⟩} ; do buildcyc(start, current, j);
if(#cyc gt 1) then cyco(#cyco+1) = cyc;; end while;

block buildcyc(start, current, j); [newelt:];
(while j+1 < ∃[k] < #list | list(k) eq current and tag(k) eq f)
⟨j, current, tag(k)⟩ = ⟨k, list(k+1), t⟩;;
if current ne start then cyc(#cyc+1) = current; j = 1; go to newelt;
      end if; end buildcyc;
```

10H. <u>Inversion of a permutation, Boothroyd algorithm</u>. This brief but surprising algorithm works as follows. Let c be a circular permutation, which we may think of as shifting elements arranged in a circle. Let s be the set of these elements. Initially, set f=c, and mark each element of s as a "head". Then, process the elements p of s, in any random order, as follows. Find the first element $g=f^k(p)$ which is marked as a head; remove the head mark of $r=f(g)$; and redefine $f(g)$ as $f(r)$ and $f(r)$ as p. To follow the action of this algorithm, divide the set s into a set of runs, each run consisting of a sequence $q, g(p), g^1(q), \ldots g^k(q)$ beginning with an element marked as a head, and continuing up to but not including the next element marked as a head. Then note inductively that

i. The first element of a run is marked, all other elements are unmarked (by definition);

ii. All elements of a run but its last have already been processed;

iii. For all but the first element of a run, f(p) is the previous element; for the first element of a run, f(p) is the first element of the next succeeding run (or is p itself, if only one run remains.)

All these remarks hold initially, with runs of unit length. Since by ii every unprocessed element is the tail q of a run, our algorithm always finds p; causes the head r of the next run to point to q, removes the mark on r, and causes p to point to the head of the next run but 1, thereby joining two runs and preserving the conditions i, ii, and iii. Since this process works for every cyclic permutation, it works for any union of cyclic permutations, i.e., any permutation. In SETL, we assume a set s and a mapping f. Then

heads = s; ($\forall p \in s$) q = p; (while <u>n</u> $q \in$ heads) q = f(q);; r = f(q); $\langle f(r), f(q)\rangle = \langle p, f(r)\rangle$; heads = heads <u>less</u> r; end $\forall$p;

11. <u>Algorithm for matching problem</u>. Given a multivalued map f on a set s, one may ask if there exists for a 1-1 g such that $g(x) \in f\{x\}$. The necessary and sufficient condition is that $\# f[t] \geq \# t$ for each subset t of s. To prove this sufficient, call t <u>thin</u> if

$\#f[t] = \#t = c(t)$; otherwise thick. If t_1 and t_2 are thin, then $\#f[t_1 \underline{n}\ t_2] = c(t_1) + c(t_2) - \#(f[t_1]\ \underline{int}\ f[t_2])$
$\leq c(t_1) + c(t_2) - c(t_1\ \underline{int}\ t_2)$,
so that (by additivity of c) $t_1\ \underline{int}\ t_2$ is thin also, and $\#(f[t_1]\ \underline{int}\ f[t_2])$ is $c(t_1\ \underline{int}\ t_2)$. Hence there exists a minimal thin set t, and letting $x \in t$ correspond to some chosen $y \in f\{x\}$, we have condition satisfied for s less x, f[s] less y, since for t_1 not thin there can be no touble, and for t_1 thin, $y \in f[t_1]$ implies $x \in t_1$. More generally, c can be any positive additive function, and we can require that a map g with all q(x)'s disjoint and $\#g\{x\} \geq c(\{x\})$ be found. Argument here reduces $c(\{x\})$ by 1 whenever y is chosen. Thus algorithmic procedure hinges on finding a minimal thin set. We take $c(x) = [t{:}y \in x]k(y)$

It is convenient to use the following function

definef pwr(x,n); return $\{y \in pow(x) \mid \#y\ \underline{eq}\ n\}$, end pwr;

which will probably be available as a primitive.

gfin = match (s, f[s], f,k); definef match (s, im, f,k);
t = minthin (s, im, f,k); x = $\exists$ t; y = $\exists$ f$\{x\}$ int im);
if k(x) gt 1 then k(x) = k(x)-1; g_1 = match (t, f[t] less y int im,f,k)
else g_1 = match (t less x, f[t] less y int im, f,k);;
g_2 = match ($\{z \in s \mid \underline{n}\ z \in t\}$, $\{z \in im \mid \underline{n}\ z \in f[t]\}$, f,k);
return g_1 u g_2 with $\langle x,y \rangle$;; end match;
definef minthin (s, im, f,k); $(1 \leq \forall n \leq \#s, y \in pwr(s,n))$
kk = $[t{:}z \in y]k(z)$; xx = $\{x \in f[y] \mid x \in im\}$; if kk gt #xx then
print "necessary condition violated"; exit;;

if kk eq#xx, then return y;;end ∀n;print "necessary condition violated"; exit; end minthin;

11A. A slightly more efficient minthin algorithm if k = 1.

definef minthin(s, im, f); n = 1; (while n le#s)

minim =#im+1; (∀y ∈ pwr(s,n)) xx = {x ∈ f[y] / x ∈ im};

if #xx lt n then print "necessary condition violated"; exit;;

if #xx eq n then return y;;

if #xx lt minim then minset = {xx}; minim = #xx;

else if #xx eq minim then xx in minset; end if; end ∀y;

(∀xx ∈ minset) fill = {x ∈ s / (∀z ∈ f{x} / n z ∈ im or z ∈ xx)};

if#fill eq minim then return fill;; end ∀xx; n = minim+1; end while;

return s; end minthin;

12. Cantor's Diagonalizer: Given s and multi-valued map f:s → s, produces set which is not element

cantset = {x ∈ s / n x ∈ f{x}}.

13. Power set generator. On successive calls, get successive elements of pwr(set,n). When set = nl, process resets.

define nexpow(set,subs,n); initially flag = 0; if flag eq 1

then go to advance;;

if n gt#set then subs = Ω; return;; flag = 1; ordset = nl;

s = set;

(while s ne nl) elt from s; ordset(#ordset+1) = elt;; last =#set;

img = {⟨i,i⟩, 1 ≤ i ≤ n}; go to ret;

```
advance:  if set = nl then go to drop;; if img(n) lt last then m=n;
go to found;; if n > ∃ [m] ≥ 1 | img(m) lt (img(m+1)-1)then go to found;
drop:  flag = 0; subs = Ω ; return;
found:  img(m) = img(m)+1; im = img(m); (m < ∀k ≤ n) im = im+1;
img(k) = im;;
ret:  subs = {ordset(img(i)), 1 ≤ i ≤ n}; return; end nexpow;
```

14. Maps generator. On successive calls, get successive maps of froms into into.

```
define nexmap(map, froms, into); initially flag = 0; if flag eq 1
      then go to advance;;
          if froms eq nl then map = nl; flag = 1; return;; flag = 1;
ordfrom = nl; ordto = nl; s = froms;
(while s ne nl) elt from s; ordfrom(#ordfrom+1) = elt;; s = into;
next = nl; elt from s; first = elt; (while s ne nl) eltx from s;
next (elt) = eltx; elt = eltx; end while; last = elt;
img = {<ordfrom(i), first>, 1 ≤ i ≤ #froms}; go to ret;
advance:  if set = nl  then go to drop;
(#froms ≥ ∀j ≥ 1) if img(j) ne last then m = j; go to found;;
drop:  flag = 0; map = Ω ; return;
found:  img(ordfrom(m)) = next(img(ordfrom(m)));
(m < ∀k ≤ #froms) img(ordfrom(k)) = first;;
ret:  map = img; return; end nextmap;
```