SETL Newsletter Number 8

November 25, 1970
J. T. Schwartz

Additional Miscellaneous SETL Algorithms

1. Permutation generator. As with other combinatorial generators, this depends on a notion of order, and steps from one item to the next in order. We use lexicographic order of permutations of integers $1 \leq \forall n \leq nn$; basic step is as follows. Find latest n which fails to exceed some m following; interchange this with the last following element which exceeds it, and put all the elements following the old position of n into ascending order (by reversal). We keep for each element the number of following elements which it exceeds, and keep a stack of all those positions which contain an element not exceeding all following elements.

```
ni = {<n,n>, 1 ≤ n ≤ nn}; exfol = {<n,0>, 1 ≤ n ≤ nn};
stack = ni lesf. n;
(while stack ne. nl.)<i, stack(#stack)><stack(#stack),∩>;
e = n-exfol(i); <ni(i), ni(e)><ni(e), ni(i)> ;
<exfol(e), exfol(i)><exfol(i), exfol(e)+1>; if exfol(i) lt. n-i then
stack(#stack+1) = i;;
(1 ≤ ∀j ≤ (n-i)/2) <ni(i+j), ni(n-j+1)> <ni(n-i+1), ni(i+j)>;;
(i+1 ≤ ∀j ≤ n) exfol(j) = 0; stack(#stack+1) = j;;
```

Note that the array ni may be replaced by any other in this procedure.

2. Generator of unordered trees. Standard order is established as follows. First take highest number of nodes, then, recursively arranging subtrees in standard order, take lexicographic order of these arrangements. To advance from one tree to the next, we find last tree which can be advanced, which is first of a terminal run of trees, all equal, and advance it if possible, reducing all its followers to single-node trees. If it can't be advanced, we add one node from a follower, and arrange this k+1-node tree in single node fashion. If it has no followers, we take the first element of the preceding run, and follow a similar procedure.

```
define enum n; desc =Λ; top = newat.; (1 ≤ ∀j ≤ n-1) a = newat;
desc(top,j) = ⟨a,1,1⟩; (while top ne Λ) print treeout top;
top = nexut top;;

the descendants of a node are triples
⟨node, treenumber, number of nodes in tree⟩

definef nextut top; enum external desc;
if desc (top,1) eq Λ then return Λ;; ld = ldesc top;
fe = ld firsteq top;  tryadv: if (nextut *desc (top,fe)) ne Λ
then do advance (fe); restart (top, fe+1); return top; end if;
if fe ne ld then do pulla(fe, ld); restart (top, fe+1);
return top; end if;
if fe eq 1 then return Λ;
fe = fe-1 firsteq top; go to tryadv;
definef ldesc top; nextut external desc; j = 1;
(while desc (top, j+1) ne Λ) j = j+1;; return j; end ldesc;
definef j firsteq top; nextut external desc; k = j;
(while - desc (top, k-1) eq - desc (top,k)) k = k-1;; return k;
end firsteq;
block advance (fe); desc (top,fe) =
⟨nextut *desc (top,fe),* - desc (top,fe) + 1,- - desc (top,fe)⟩;
end advance;
block pulla (fe, ld); nds = nl; addnodes (*desc(top, ld), nds);
node from nds;
ndz = nl; addnodes (*desc (top, fe), ndz); attach (ndz, node, 1);
desc (top, fe) = ⟨node, 1,- - desc (top, fe) + 1⟩;
attach (nds, top, ld); end pulla;
end nextut;
define restart (top, loc); nextut external desc; nds = nl;
(loc ≤ ∀j ≤ ldesc top)  addnodes (*desc (top, j), nds);;
attach (nds, top, loc);
return; end restart;
define attach (nds, top, loc); nextut external desc; j = loc;
```

(while desc (top, j) ne Λ) desc (top, j) = Λ; j=j+1; j = loc;
($\forall$ n $\in$ nds) desc (top, j) = ⟨n, 1, 1⟩; j=j+1; return; end attach;
define addnodes (top, nds); nextut external desc; top in nds; j = 1;
(while desc (top, j) ne Λ) addnodes (*desc (top,j), nds);
j = j+1;; return; end addnodes; treeout is omitted output routine.

3. Huffman table. Given set char of characters with frequency function freq(c). First sort into decreasing frequency to get nchar(n)=ordered sequence. Then, using binary search and insert function elt insert nchar proceed as follows: (while #nchar gt. 1)
node = newat.; nc = #nchar; r = nl.; l = nl.;
⟨l(node), r(node), freq(node)⟩=
⟨nchar(nc), nchar(nc-1), freq(nchar(nc))+ freq(nchar(nc-1))⟩;
⟨nchar(nc-1), nchar(nc)⟩=⟨Λ, Λ⟩; node insert. nchar;;
top = node; code = nl.; walk(nl., top);
define walk(seq, top); external code; if l(top) ne. Λ then s = seq;
s(#s+1) = 0;
walk (s, l(top)); s(#s) = 1;
walk (s, r(top));
else ⟨top, seq⟩ in. code;; return; end walk;

4. Huffman encode and decode of sequence cseq of characters, using code table and tree prepared above.
hcseq = nl.; (1 ≤ $\forall$j ≤ #cseq, 1 ≤ k ≤ #code(cseq(j)) hcseq(#hcseq+1)=
code (cseq(j)) (k);;
and the dehuffman process
cseq = nl; node = top; (1 ≤ $\forall$j ≤ #hcseq)
node = if hcseq(j) eq. 0 then l(node) else r(node);
if l(node) eq. Λ then cseq (#cseq+1) = node; node = top;; end $\forall$j;

5. Fisher-Galler equivalence declaration analysis algorithm.
Each node is related to a certain "master node", or to none at all; this relation involves a relative offset, and is summarized by a map mast(node) = ⟨node', offset⟩. A set of triples {⟨node, node', offset⟩} is to be processed into this form, and a

flag set if a contradiction results.

```
err = f.; (∀t ∈ trips) ⟨node, nod2, offs⟩t; (while mast(node) ne.Λ)
⟨node, offs⟩ ⟨*mast(node), offs +- mast(node)⟩;;
(while mast(nod2) ne.Λ)
⟨nod2, offs⟩⟨*mast(nod2), offs -- mast(nod2)⟩;;
if node eq. nod2 and. offs ne. 0 then err = t.; go to done;
else mast(node) = ⟨nod2, offs⟩;;
end ∀t; done:;
```

6. Polynomial addition and multiplication in form described by Knuth, Volume I, p. 273. A polynomial is a sequence poln(n) of terms, each of which has the form ⟨coef, exp of x, exp of y, exp of z⟩ with a coefficient and three exponents. The exponents are in standard order.

```
definef poll plus. pol2; return  plusadj (poll, pol2, ⟨1,0,0,0⟩);
end plus;
definef poll times. pol2; p = nl.; (1 ≤ ∀j ≤ pol2)
p = plusadj (poll, p, pol2 (j));; return p; end times;
definef plusadj (poll, pol2, adj); n = 1; m = 1; sum = nl.;
(while n le.#poll or m le.#pol2)
if m gt.#pol2 then ⟨sum(#sum+1), n⟩ = ⟨poll(n), n+1⟩;
else if n gt.#poll then ⟨sum(#sum+1), m⟩ = ⟨pol2(m) mon.adj, m+1⟩;
else p2a = pol2(m) mon.adj; pl = poll(n); if - pl eq. - p2a then
c = *pl +*p2a; ⟨m,n⟩⟨m+1, n+1⟩; if c ne. 0 then sum(#sum + 1) =
⟨c, - pl⟩;;
else ⟨sum(#sum+1), n, m⟩ if - pl bigexp. p2a then ⟨pl, n+1, m⟩
     else ⟨p2a, n, m+1⟩; end if; end while; end times;
definef el bigexp. e2; (while el ne.Λ) val = if *el gt. *e2 then t.
else if *e2 gt. *el then f. else Λ; if val ne. Λ then return val;;
el = - el; e2 = - e2; end while; return t.;
definef p mon. q; /*/ monad multiplication /*/
return ⟨*p**q, 2 z.p + (2 z.q), 3 z.p + (3 z.q), 4 z.p + (4 z.q)⟩;
end mon; end plusadj;
```