This newsletter gives a rewrite of the generalized nodal span parse routine described in Newsletter 46. A SETLB program was written in accordance with the SETL algorithm therein described, debugged, and retranslated into true SETL.

The description of the algorithm and its arguments is unchanged, but the form of the "undigested" or incomplete span has been changed to <P, π, q, j>, whereas it had previously been <P, π, j, q>. The meaning of the symbols is described in Newsletter 46.

The program was debugged using the following simple expression grammar with attributes (erasable productions were not tested):

\[ e + e \ op \ e \ n \ c | n | c | (e) \]

n and c are assigned attribute 4; op is a lexical type meaning operator where +, - are assigned attribute 1; *, / attribute 2; and ↑ attribute 3. The attribute testing functions always return true for the last three productions, and the resultant e is assigned the attribute 4. The attributes test for \( e + e_1 \ op \ e_2 \) is that attribute \( e_1 \) ≥ attribute(op) and attribute \( e_2 \) > attribute(op). The resultant e is assigned attribute(op). This grammar obeys the usual precedence rules for expressions.

A strong effort was made to stay true to the original code. However, changes needed to remove bugs, and the incompleteness of SETLB and its irritating deviations from true SETL did force a fair amount of recoding. The code presented here is true to the actual running program aside from external statements (due to different name scoping conventions) and small changes within statements which reflect left-right rather than right-left precedence, etc.

The program required 220K core memory to execute, and parsed \( A + B \ast C \uparrow 2 \) in the phenomenal execution time of 186.7 seconds. \( ((A+B)\ast C)\uparrow 2 \), which does not generate false starts was parsed in 135.6 seconds. I plan to rewrite the algorithm to eliminate false starts, maintain more extensive division
list information, and minimize the number of undigested span entries. These improvements should increase efficiency by at least a factor of five. I do not know, however, when my schedule will allow for this effort.

Thanks are due to Alan Goldberg who coded the getspans routine and assisted in debugging.
define genodparse(input, igram, root, terms, testfn, atfn, spans, divlis, amb, wdkind);

/* general nodal span parse allowing arbitrary grammar, null productions, attributes and attribute testing. Grammar is set of productions (igram) in form 
<x_1, ..., x_n, a> corresponding to productions a \rightarrow x_1...x_n. Input is a string of tuples <y, {attr}> where y is a lexical type and {attr} is a set of possible attributes for this input token. */

/* first calculate the fixed relationships needed */
newerase = {hd x, x e igram|(#x) eq 1}; gram={xcigram|(#x)gt 1};
erase = n\$; (while newerase ne n\$) erase = erase u newerase;
newerase = {g(#g), g e gram| (1 < j < #glg(j) e erase)}
an g(#g) e erase};
end while newerase; eraselast = n\$;

/* eraselast(g) is the last erasable symbol - possibly 0 - of a production string, barring the last symbol itself */
(\forall g e gram) j = 0; (1 < Vi < #g-1)
if g(i) e erase then j = i; else quit;; end Vi; eraselast(g) = j;
end 't/g;
begins = {<g(#g), g(j+1)>, g e gram, 0 < j < eraselast(g)};
syms = {g(j), g e gram, 1 ~ j ~ #g};
ultbegins = closef(begins,syms);

/* prepare for main loop of parse */
divlis = n\$; compspans = n\$; incspans = n\$; startat = n\$;

/* process first input token */
n = l; startat(n) = ultbegins{root}; makenewith(input(n));
/* digested complete span is <p, π, q, atr>
incomplete spans are <p, π, q, j> */

/* */
block spst; sp(l); end spst;
block prod; sp(2); end prod;
block spnd; sp(3); end spnd;
block inx; sp(4); end inx;
/* main loop of parse */
(1 < \forall n < #input)
SETL-95 -4

/* startat is inductively the valid symbols in the nth position*/
startat(n) = ultbegins[\{prod(inx+1), sp \in incspans |
  spnd \equiv n a inx+1 \lt \#prod\}];
makenewwith(input(n)); end Vn;

/* check if grammatical return n\$ if not */
totspans = \{sp \in compspans | (spst \equiv l) \& (spnd \equiv \#input+1)
  \& (prod(#prod) \equiv root)\};
if totspans \equiv n\$ then <spans,divlis,amb> = <n\$,n\$,f>;
  return;

/* else determine ambiguity */
amb = if #totspans \geq 1 then t else f;

/* clean up set of spans and division list */
/* wdkind gives validated reading of tokens. compdiv and newdiv
  the division lists */
spans = n\$; compdiv = n\$; newdiv = n\$; wdkind = n\$;
getspans(totspans); divlis = newdiv u compdiv;
return; end genodparse;

define makenewwith(c); genodparse external gram, startat,
eraselast, divlis, incspans,n;

/* */
block indivlis(x); if divlis(spn) \&\& then divlis(spn)=n\$;
  divlis(spn) = divlis(spn) with x; end indivlis;

symb = c(1); ats = c(2); todo = n\$;
(\forall g \in gram \&\& g(#g) \equiv startat(n)) /* for all valid productions */
(1 \leq \forall i \leq eraselast(g)+l | g(i) \equiv symb) /* match to input symbol*/
(\forall x \in ats) reading = <symb,x>; /* all input readings */
spn = <n,g,n+1,i>; spn in todo; indivlis(reading);
end \forall x; end \forall i; end \forall g;

/* now try to extend incomplete spans by appending input symbol*/
(\forall sp \in incspans | (spnd \equiv n) \& (prod(inx+1) \equiv symb))
(\forall x \in ats) reading = <symb,x>;
spn = <spst,prod,n+1,inx+1>; spn in todo; indivlis(reading);
end \forall x; end \forall sp;

/* now process spans in todo. if complete, buildok will
  test and generate attributes and then generate all
  possible new spans using the new complete spans.
  All incomplete spans maintained in incspans.
  Try to extend incomplete spans - e.g. if next symbol
  is erasable */
SETL 95-5

/* */
block iscomp; inx eq #prod-1; end iscomp;
block erasenext; prod(inx+1) c erase; end erasenext;
block extnd; spn = <spst,prod,spnd,inx+l>;
    spn in todo; indivlis(nl); end extnd;
(while todo ne nl) sp from todo; sp in incspans;
if iscomp then buildok(sp); else if erasenext then extnd;;
end while todo; return; end makenewith;

/* */
define buildok(sp);
/* builds complete digested spans from sp and checks validity. */
/* If valid, complete span is inserted into compspans, and new */
/* spans generated from it. Spseq is a chain of incomplete spans */
/* from which sp was generated, and setseq(i) is the (sequenced) */
/* division list of spseq(i). Spseq(i+1) is generated by chopping */
/* off an element of setseq(i) from spseq(i) -- the particular */
/* element is indicated by ixseq(i). This process is accomplished */
/* in the routine extend, and will be iterated for all possible */
/* extensions. The iteration is accomplished in the routine advance. */
/* Atseq(i) maintains the attributes of setseq(i)(ixseq(i)), and */
/* it is these attributes which determine the validity and */
/* attributes of the completed span. Valid attributes are */
/* maintained in atset */
/* */
/* Lastparts returns the division list of an incomplete span. */
/* Seqof sequences a set. */
/* Atrib returns the attributes of its argument. */
/* Testfn and atfn are the mappings of productions into their */
/* testing and attribute evaluation routines */
*/
genodparse external gram,startat, eraselast, divlis, incspans,
compspans, testfn, atfn;
makenewith external todo;
spseq = nl; setseq = nl; ixseq = nl; atseq = nl;
spseq(1) = sp; setseq(1) = seqof(lastparts(spseq(1)));
ixseq(1) = 1; atseq(1) = atrib(setseq(1)(1));
extend; /* accomplish first extension, setting values of */
/* spseq, setseq, ixseq, atseq */
atset = nl; test = testfn(prod); atfnc = atfn(prod); nparts = #prod-1;
attrue = {<j, atseq(nparts+1-j)>, 1<j<nparts};
/* sequence of attributes of symbols of prod */
if test(attrue) then atfnc(attrue) in atset;
/* advance will return t if it can generate a new extension, 
   f otherwise */
(while advance( )) attrue = {<j,atseq(nppts+1-j)>,1≤ j≤ nparts};
if test(attrue) then atfnc(attrue) in atset;
end while advance( );
/* place all new complete spans in newcomp, then try to build on them*/
newcomp = {<spst, prod, spnd, atr>, atr e atset}- compspans;
(∀spnew e newcomp) spstl = spnew(1); prod1 = spnew(2); spnd=spnew(3);
/* try for a span starting with new nonterminal preceded perhaps 
   by eraseables */
(∀g e gram, 0 ≤ j ≤ eraselast(g) | (g(#g) e startat(spstl))
   a (g(j+1) eq prod1(#prod1)))
spn = <spstl,g,spndl,j+1>; spn in todo; indivlis(spnew);
/* note use of previously defined macro */
end ∀g;
/* now try to extend incomplete spans */
(∀spinc e incspns | (spinc(3) eq spstl) a (spinc(2)(spinc(4)+1)
   eq prod1(#prod1)))
spn = <spinc(1), spinc(2), spnd , spinc(4)+1>; spn in todo;
   indivlis(spnew);
end ∀spinc; end ∀spnew;
comspans = compspans u newcomp; return;
end buildok;
/* */
define extend;
/* see comment to buildok. Spseq is extended to its full length by 
   first determining its current length (# ixseq), and then chopping 
   off the division list element setseq(i)(ixseq(i)). 
   Subsequent extensions always chop setseq(i)(1). Subsequent calls 
   to extend from advance will first reset ixseq, then backup the 
   spseq extension if necessary generating any other possible 
   extensions */
genodparse external terms;
buildok external ixseq, spseq, setseq, atseq;
/* */
block spstn; spn(1); end spstn;
block prodn; spn(2); end prodn;
block inxn; spn(4); end inxn;
block spndn; spn(3); end spndn;
block divleltnt; (divlelt eq n) or (divlelt eq Ω); end divleltnt;
block spnelterm; divlelt(l) c terms; end spnelterm;
block nuloff; spn = <spstn,prodn,spndn,inxn-1>; end nuloff;
block oneoff; spn = <spstn,prodn,spndn-1,inxn -1>; end oneoff;
block eltoff; divst = divlelt(l); spn =<spstn,prodn,divst,inxn-1>;
   end eltoff;
end def extend;
def advance;
buildok external ixseq,spseq,setseq,atseq;
/* this routine advances the last element of spseq (e.g. by 
   incrementing ixseq) and extends. If advancement is impossible, 
it removes one or more elements from the sequence and attempts 
to advance and extend. The process ends if spseq would become 
null, in which case f is returned, otherwise t. */
/* */
block nomore; seqn(ixn) eq Ω; end nomore;
block backup; if n eq 1 then return f;;
   spseq(n) = Ω; setseq(n) = Ω; ixseq(n) = Ω; atseq(n) = Ω;
   n = n-1; ixn = ixseq(n)+1; seqn = setseq(n); end backup;
end def lastparts(span);
genodparse external divlis;
/* this returns the division list of span, which is the set of 
   spans or readings (<terminal symbol, attribute>) of which the 
   last production symbol of span is an instance. If this last 
symbol can only be realized as a null production, there may be 
no division list entry, in which case {n} is returned */
return if divlis(span) eq \( \Omega \) then \( \{n\} \) else divlis(span);
end lastparts;

define seqof(set); /* sequences a set */
seq = n; i = 0;
(\( \forall x \in \) set) i = i+1; \(<i,x> \in \) seq end \( \forall x; \) return seq;
end seqof;

define attrib(divlelt); /* returns attributes of division list element */
genodparse external terms;
block nulat; return n end nulat;
block oneat; return divlelt(2) end oneat
block eltat; return divlelt(4) end eltat
/* other macros defined in extend */
if divlelt=nul then nulat; else if spnelt then oneat; else eltat;
end attrib;

define closef(begins,syms);
/* note that this yields a bit more than a true closure function. For our use we desire \(<s,s> \) to be a member of the closure set for all \( s \in \) syms */
closer = n;
(\( \forall s \in \) syms) closes = n; new = \{s\};
(while new ne n closes = closes u new; new = begins[new]-closes;
end while new; closer = closer u \(<s,x>, x \in \) closes); end \( \forall s; \)
return closer; end closef;

define getspans(topspan);
genodparse external testfn, atfn, compdv, newdiv, amb, spans, wdkind;
buildok external ixseq, spseq, setseq, atseq;
/* This routine collects all spans which enter into its argument. The spans are stored in spans, and their division lists are stored in compdv and newdiv */
/* compdv contains the initial splitoff of digested complete spans; newdiv the validated divisions of its incomplete initial parts */
todo = topspan;
(while todo ne n) next from todo; next in spans;
<st, prd, endd, att> = next;
sp = <st, prd, endd, #prd-1>; /* undigested form to recover division list */
/* now we use a process like buildok which calls out the divisions which pass all tests and gives the attributes. These are also counted for ambiguity */

ndivs = 0; spseq = nλ; setseq = nλ; ixseq = nλ; atseq = nλ;

spseq(l) = sp;
setseq(l) = seqof(lastparts(spseq(l))); atseq(l)=attrib(setseq(l)(1));
ixseq(l) =l;

extend;
test= testfn(prod); atfnc = atfn(prod); nparts = #prod-1;
atrst;

/* Atrst tests whether the spseq division of sp is valid and whether the attributes of this division matches the attributes of sp. If yes, the initial splitoff is placed in compdiv, and the others in newdiv. Both types are placed in todo */

(while advance( )) atrst; /* do same for all spseq sequences */
end while advance;
if ndivs gt 1 then amb = t;
end while todo; return; end getspans;

/* */

define atrst; genodparse external compdiv,newdiv,spans;

getsans external test, atfnc, ndivs, next, nparts, att, todo;
buildok external atset, setseq; ixseq;
attrue = {<j, atseq(nparts+l-j)>, 1 ≤ j ≤ nparts};
if(test(attrue) a atfnc(attrue) eq att) then ndivs = ndivs+1;
if compdiv(next) eq Ø then compdiv(next) = nλ;;
compdiv(next) = compdiv(next) with setseq(l)(ixseq(l));

/* If a span, place the division list element in todo */
if((#(setseq(l)(ixseq(l))) eq 4) a n (setseq(l)(ixseq(l)) e spans)
   then (setseq(l)(ixseq(l))) in todo;
putifterm(l); /* collects token type information */
(l < ∀j ≤ nparts) /* now repeat process for the rest of the sequence*/
if newdiv(spseq(j)) eq Ø then newdiv(spseq(j)) = nλ;;
newdiv(spseq(j)) = newdiv(spseq(j) with setseq(j)(ixseq(j));
if((#(setseq(j)(ixseq(j))) eq 4) a n (setseq(j)(ixseq(j)) e spans))
   then (setseq(j)(ixseq(j))) in todo;;
putifterm(j); end ∀j;
end if test; return; end atrst;
define putifterm(j); /* record, in wdkind, the reading of input which yielded a parse */
genodparse external terms, wdkind;
buildok external spseq, setseq, ixseq;
getspans external nparts, sp;
if prod(nparts+1-j) e terms then
  termat = <spseq(j)(3)-1, hd(setseq(j)(ixseq(j))),
           tt(setseq(j)(ixseq(j)))>;
/* termat is <token number, terminal type, attributes> */
termat in wdkind; return; end putifterm;