SETL Newsletter No. 95 January 17, 1973

Generalized nodal span parse routine, corrected version.

Yeshiah Feinroth

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 $<P,\pi,q,j>$, whereas it had previously been $<P,\pi,j,q>$. 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):

$e \rightarrow e \text{ op } e|n|c|(e)$

n and c are assigned attribue 4; op is a lexical type meaning operator where +, - are assigned attribute 1; *, / attribute 2; and $\uparrow$ 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 $e \rightarrow e_1 \text{ op } e_2$ is that attribute $(e_1) \geq$ 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 * C $\uparrow$ 2 in the phenomenal execution time of 186.7 seconds. ((A+B)*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
     <x1,...,xn,α> corresponding to productions α → x1...xn.
     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 ε igram|(#x) eq 1}; gram={xεigram|(#x)gt 1};
erase = nℓ; (while newerase ne nℓ) erase = erase u newerase;
newerase = {g(#g), g ε gram|(1 ≤ j < #g|g(j) ε erase)
                                 a n g(#g) ε erase};
end while newerase; eraselast = nℓ;
/* eraselast(g) is the last erasable  symbol - possibly 0 -
     of a production string, barring the last symbol itself */
(∀g ε gram) j = 0; (1 ≤ ∀i < #g-1)
if g(i) ε erase then j = i; else quit;; end ∀i; eraselast(g) = j;
end ∀g;
begins = {<g(#g),g(j+1)>, g ε gram, 0 ≤ j ≤ eraselast(g)};
symbs = {g(j), g ε gram, 1 ≤ j ≤ #g};
ultbegins = closef(begins,symbs);
/* prepare for main loop of parse */
divlis = nℓ; compspans = nℓ; incspans = nℓ; startat = nℓ;
/* process first input token */
n = 1; 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(1); 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 < ∀n ≤ #input)
```

```
/* startat is inductively the valid symbols in the nth position*/
startat(n) = ultbegins[{prod(inx+1), sp ϵ incspans|
                        spnd eq n a inx+1 ℓt #prod}];
makenewith(input(n)); end ∀n;
/* check if grammatical return nℓ if not */
totspans = {sp ϵ compspans|(spst eq 1) a (spnd eq #input+1)
                              a(prod(#prod) eq root)};
if totspans eq nℓ then <spans,divlis,amb> = <nℓ,nℓ,f>;
                              return;;
/* else determine ambiguity */
amb = if #totspans gt 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 = nℓ; compdiv = nℓ; newdiv = nℓ; wdkind = nℓ;
getspans(totspans); divlis = newdiv u compdiv;
return; end genodparse;
/*  */
define makenewith(c); genodparse external gram, startat,
     eraselast, divlis, incspans,n;
/*  */
block indivlis(x); if divlis(spn) eq Ω then divlis(spn)=nℓ;;
     divlis(spn) = divlis(spn) with x; end indivlis;
symb = c(1); ats = c(2); todo = nℓ;
(∀g ϵ gram|g(#g) ϵ startat(n))  /* for all valid productions */
(1 ≤ ∀i ≤ eraselast(g)+1|g(i) eq symb)   /* match to input symbol*/
(∀x ϵ ats) reading = <symb,x>; /* all input readings */
spn = <n,g,n+1,i>; spn in todo; indivlis(reading);
end ∀x; end ∀i; end ∀g;
/* now try to extend incomplete spans by appending input symbol*/
(∀sp ϵ incspans|(spnd eq n) a (prod(inx+1) eq symb))
(∀x ϵ ats) reading = <symb,x>;
spn = <spst,prod,n+1,inx+1>; spn in todo; indivlis(reading);
end ∀x; end ∀sp;
/* now process spans in todo. if complete, buildck  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  */
```

```
/*  */
      block iscomp; inx eq #prod-1; end iscomp;
      block erasenext; prod(inx+1) ε erase; end erasenext;
      block extnd;  spn = <spst,prod,spnd,inx+1>;
                    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(nparts+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 ϵ atset}- compspans;
(∀spnew ϵ newcomp) spstl = spnew(1); prodl = spnew(2); spnd=spnew(3);
/* try for a span starting with new nonterminal preceded perhaps
   by eraseables */
(∀g ϵ gram, 0 ≤ j ≤ eraselast(g)  | (g(#g) ϵ startat(spstl))
                                     a (g(j+1) eq prodl(#prodl)))
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 ϵ incspns | (spinc(3) eq spstl) a (spinc(2)(spinc(4)+1)
   eq prodl(#prodl)))
spn = <spinc(1), spinc(2), spnd , spinc(4)+1>; spn in todo;
   indivlis(spnew);
end ∀spinc; end ∀spnew;
compspans = 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 divleltnl; (divlelt eq nl) or (divlelt eq Ω); end divleltnl;
     block spnelterm; divlelt(1) ϵ 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(1); spn =<spstn,prodn,divst,inxn-1>;
        end eltoff;
     n = #ixseq; ixn = ixseq(n); spn = spseq(n); seqn = setseq(n);
     divlelt = seqn(ixn); nmax = #prodn-1;
     (n <∀ m ≤ nmax) ixseq(m) = 1;
     if divleltnl then nuloff; else if spnelterm then oneoff;
        else eltoff;
     spseq(m) = spn; setseq(m) = seqof(lastparts(spn));
     divlelt = setseq(m)(1); atseq(m) = atrib(divlelt);
     end ∀m; return; end extend;
/*  */
     definef 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;
     n = #ixseq; ixn = ixseq(n); seqn = setseq(n); ixn = ixn+1;
     (while nomore) backup;;
     ixseq(n) = ixn; atseq(n) = atrib(seqn(ixn)); extend;
     return t; end advance;
/*  */
     definef 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 {nl} is returned */
```

```
    return if divlis(span) eq Ω then {nℓ} else divlis(span);
    end lastparts;
/*  */
    definef seqof(set); /* sequences a set */
    seq = nℓ; i = 0;
    (∀x ∈ set) i = i+1; <i,x> in seq; end ∀x; return seq;
    end seqof;
/*  */
    definef atrib(divlelt); /* returns attributes of division list
       element */
    genodparse external terms;
    block nulat; return nℓ; end nulat;
    block oneat; return divlelt(2); end oneat
    block eltat; return divlelt(4); end eltat
    /* other macros defined in extend */
    if divleltnl then nulat; else if spnelterm then oneat;
       else eltat;;
    end atrib;
/*  */
    definef closef(begins,symbs);
    /* 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 ∈ symbs */
    closer = nℓ;
    (∀s→symbs) closes = nℓ; new = {s};
    (while new ne nℓ) closes = closes u new; new = begins[new]-closes;
    end while new; closer = closer u {<s,x>, x ∈ closes}; end ∀s;
    return closer; end closef;
/*  */
    definef 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 compdiv and newdiv */
    /* compdiv contains the initial splitoff of digested complete spans;
       newdiv the validated divisions of its incomplete initial parts */
    todo = topspan;
    (while todo ne nℓ) 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 = nℓ; setseq = nℓ; ixseq = nℓ; atseq = nℓ;
spseq(1) = sp;
setseq(1) = seqof(lastparts(spseq(1))); atseq(1)=atrib(setseq(1)(1));
   ixseq(1) =1;
extend;
test = testfn(prod); atfnc = atfn(prod); nparts = #prod-1;
atrtst;
/* Atrtst 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( )) atrtst; /* do same for all spseq sequences */
end while advance;
if ndivs  gt 1 then amb = t;
end while todo; return; end getspans;
/*  */
define atrtst;  genodparse external compdiv,newdiv,spans;
getspans external test, atfnc, ndivs, next, nparts, att, todo;
buildok external atset, setseq; ixseq;
attrue = {<j, atseq(nparts+1-j)>, 1 ≤ j ≤ nparts};
if(test(attrue) a atfnc(attrue) eq att) then ndivs = ndivs+1;
if compdiv(next) eq Ω then compdiv(next) = nℓ;;
compdiv(next) = compdiv(next) with setseq(1)(ixseq(1));
/* If a span, place the division list element in todo */
if((#(setseq(1)(ixseq(1))) eq 4) a n (setseq(1)(ixseq(1)) ε spans)
   then (setseq(1)(ixseq(1))) in todo;
putifterm(1); /* collects token type information */
(1 < ∀j ≤ nparts) /* now repeat process for the rest of the sequence*/
if newdiv(spseq(j)) eq Ω then newdiv(spseq(j)) = nℓ;;
newdiv(spseq(j)) = newdiv(spseq(j) with setseq(j)(ixseq(j));
if((#(setseq(j)(ixseq(j))) eq 4) a n (setseq(j)(ixseq(j) ε spans))
   then (setseq(j)(ixseq(j))) in todo;;
putifterm(j); end ∀j;
end if test; return; end atrtst;
```

```
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) ε terms then
   termat = <spseq(j)(3)-1, hd(setseq(j)(ixseq(j))),
     tℓ(setseq(j)(ixseq(j)))>;
/* termat is <token number, terminal type, attributes> */
termat in wdkind; return; end putifterm;
```