SETL Newsletter Number 46 July 21, 1971

J. Schwartz

Generalized nodal span parse routine; preliminary draft.

The following newsletter gives a nodal span parse routine incorporating Earley's startat improvement, but substantially generalized from that given in the SETL notes. The main generalizations are as follows:

i. An arbitrary context-free grammar, not required to be in Chomsky normal form, is treated. In particular, "erasing" or "null" productions $\alpha \rightarrow$ are allowed.

ii. A system of calculated node attributes which can be tested is provided. This gives a parser with some of the power of the Sager "restriction language". The attribute value assigned to a node ν is calculated from the attributes of the descendants of ν, the function employed in this calculation depending on the syntactic type of ν.

The arguments of the routine are as follows:

input - a string of pairs, representing the input. The first component of each pair is a terminal syntactic type name; the second component is an associated attribute. Each input token is assumed to have some particular set of possible attributes, obtained from the token by some process of "dictionary lookup". An attribute may be composite, perhaps highly composite; each particular syntactic-type attribute pair in the set of attributes corresponds to some particular "reading" of the token.

igram - the grammar, i.e., a set of "productions"π; see the algorithm below for the form assumed for these productions;

root - the root syntactic type;

terms - the set of terminal syntactic types;

testfn - a mapping sending each production to the attribute-testing routine used to validate well-formedness of its collection of descendants;

atfn - a mapping sending each production to the routine which calculates the attribute of a node from the attributes of all its descendants;

spans-a collection of spans present in the input(output argument);

divlis - subdivision information for spans (output argument);

amb - ambiguity flag (output argument);

The algorithm given below begins by calculating the set of all erasable syntactic types, and, for each production $x_1,\ldots,x_n$ the maximum length (less than n) of an erasable string $x_1,\ldots,x_j$. The "ultbegins" relationship, which tells us when can generate a string whose first element is , is also calculated.

Spans are maintained in two forms. The basic "undigested" form is

(1) $\langle p, \pi, j, q \rangle$;

here $p<q$, π is a production $\alpha \longrightarrow x_1,\ldots,x_n$, and $j<n$. We say that this span is present in an input string if the p-th through (q-1)-st symbols of the input string can be derived from $x_1,\ldots,x_j$ by the application of successive productions of <u>gram</u>. The span (1) is called <u>complete</u> if $j=n$, <u>incomplete</u> otherwise. A complete span (1) is only valid if it can be divided into subspans (corresponding to the descendants of the tree-node corresponding to the span (1)) having attributes which pass the test <u>testfn</u>(π). If this is the case, an attribute can be assigned to the span using the associated function atfn(π); once this test-and-calculation process has been applied, the span will be maintained in the "digested" form

(2) $\langle p,\pi,q,atr\rangle$,

where atr is one of the possible attributes of the complete span specified by $\langle p,\pi,q\rangle$.

Digested complete spans are held in a set compspans; incomplete spans are held in a set incspans.

The algorithm begins by initializing startat(1) to the set of all syntactic symbols which can begin a sentential string of the language under consideration; following which a main scan loop begins.

The "division list" function divlis maps each span (1) into a set S each of whose elements corresponds to one of the ways in which the element x_j of the production $\tau:\alpha\longrightarrow x_1,\ldots,x_n$ can be realized.

(a) If x_j can be realized by the erasure of an erasable intermediate element, the atom nl will be present in S;

(b) If x_j is a terminal symbol representing some particular terminal type, and this is matched by an input symbol which can be read as an item of this terminal type, then a pair consisting of the terminal symbol and its attribute will occur in S;

(c) If x_j is an intermediate symbol which can be realized by some digested complete span y, then y will be present in S.

Note that the elements of divlis(sp) are therefore always either markers nl; pairs $\langle c, atr\rangle$ corresponding to terminal syntactic types, or digested complete spans.

The rules for span formation are as follows: If the first j symbols $x_1,\ldots,x_j$ of $\tau:\alpha\longrightarrow x_1,\ldots,x_n$ are all erasable, and x_{j+1} is a terminal symbol which is a possible reading of the n-th input token, then the span $\langle n,\pi,j+1,n+1\rangle$ is present, provided however that $\alpha \in$ startat(n); if $\langle p,\pi,j,n \rangle$ is present and incomplete, and x_{j+1} is a terminal symbol which is a possible reading of the n-th input token, then the span $\langle p,\pi,j+1,n+1 \rangle$ is present. If $\langle p,\pi,j,n \rangle$ is present and incomplete, and x_{j+1} is an erasable intermediate symbol, then $\langle p,\pi,j+1,n \rangle$ is present. Note however that spans of this form, tentatively admitted, must be rejected unless they are built up in some way from subspans whose attributes satisfy the validity test associated with the production π.

If $\langle p,\pi,j,n \rangle$ is present and incomplete, while the complete digested span $\langle r,\pi',q,atr \rangle$ is present, and if π' has the form $\beta \rightarrow z_1 \ldots, z_m$ while $x_{j+1} = \beta$, then $\langle p,\pi,j+1,q \rangle$ is present. Finally, if the complete digested span $\langle p,\pi',q,atr \rangle$ is present, if π' has the form $\beta \rightarrow z_1 \ldots z_m$, and if the first j symbols of π are erasable while $x_{j+1} = \beta$ and $\pi \in$ startat(n), then $\langle p,\pi,q,j \rangle$ is present.

The attributes of the possible decompositions of an undigested complete span <u>sp</u> are tested for validity when <u>sp</u> is taken up for processing (cf. routine <u>buildok</u> below); one digested complete span will be formed from <u>sp</u> for each attribute which <u>sp</u> might have. This is achieved as follows:

Any element of divlis(sp) represents some possible "last subpart" of <u>sp</u>; using this last subpart, we may calculate a corresponding first part. Iterating this division process, we obtain a complete division of <u>sp</u> into subparts. Each such sequence of subparts corresponds to a sequence of attributes. If these attributes satisfy the test condition associated with the production π around which the span <u>sp</u> is built, the division is acceptable, and a digested span is built by attaching a calculated attribute to <u>sp</u>. This attribute is calculated by applying the attribute calculation function associated with π to the sequence of subpart attributes. The routine <u>buildok</u> (see below) which accomplishes all this makes use of a generation-and-extension mechanism which generates all the possible divisions of <u>sp</u> in turn.

The SETL algorithm is as follows:

```
define genodparse(input, igram, root, terms, testfn, atfn, spans, divlis,
    amb);
/* generalized nodal span parse allowing arbitrary grammar,
   null productions, attributes and attribute testing */
/* first calculate the fixed relationships needed.  grammar assumed
   given as set of (n+1)-tuples <x_1,...,x_n,α> corresponding
   to productions α → x_1...x_n  */
newerase = {hd x, x ∈ igram | (#x) eq 1}; gram = {x ∈ igram | (#x) gt 1};
erase = nl; (while newerase ne nl) erase = erase u newerase;
newerase = {g(#g), g ∈ gram | (1≤∀j<#g | g(j) ∈ erase)
                  a  g(#g) n ∈ erase};
end while newerase; eraselast=nl;
(∀g ∈ gram)  j=0; (1≤∀i<#g-1)
if g(i) ∈ erase then j=i  else quit;; end ∀i; eraselast(g)=j;
end ∀g;
begins= {<g(#g),g(j+1)>, g ∈ gram, 1≤j≤eraselast(g)};
symbs= {g(j), g ∈ gram, 1≤j≤#g};
ultbegins=closef(begins, symbs);
/* the above need not be repeated unless the grammar is changed */
/* now initialize preparatory to main loop of parse */
divlis=nl;compspans=nl;incspans=nl;
startat=nl;startat(1)=ultbegin {root} ;
makenewith(input(1));

/* digested complete span is <p,π,q,atr>  */
/* other spans are quadruples <p,π,j,q>  */
block spst; sp(1); end spst; block prod; sp(2); end prod;
block inx; sp(3); end inx; block spnd; sp(4); end spnd;
/* main loop of parse */
(1<∀n≤#input)
/* first calculate startat */
```

```
        startat(n)=ultbegin [{prod(inx+1),sp∈incspans}];
        makenewith(input(n));end ∀n;
        /* check on grammaticalness */
        if   {span∈compspans | sp(1) eq 1 a
             sp(3) eq(#input+1) a (sp(2)) (#(sp(2)))
                         eq root}  is totspans eq nl
             then ⟨spans,divlis,amb⟩=⟨nl,nl,f⟩;return;;
        /* else clean up set of spans and determine ambiguity */
        spans=nl;
        /* 'getspans' subroutine will also prepare cleaned division list */
amb =   if (#totspans) gt 1 then t else f; compdiv=nl; newdiv=nl; wdkind=nl;
        /* compdiv gives initial split-off of digested complete spans;
           newdiv  the validated divisions of its incomplete initial parts;
           wdkind the validated reading of particular tokens.  */
        getspans [totspans]; divlis=newdiv u compdiv; return;
        /* main block containing new-span formation */
        block makenewith(c);
        todo=nl;
        (∀g∈gram, reading∈c | g(#g)∈startat(n) a g(lasterase(g)+1)
             eq hd reading)
             spn=⟨n,g,lasterase(g)+1,n+1⟩;spn in todo;
             divlis(spn)=if divlis(spn) is divl ne Ω then divl else nl
                         with reading; end ∀g;
        (∀sp∈incspans, reading∈ c|  spnd eq n a
                   prod(inx+1) eq hd reading)
             spn=⟨spst,prod,inx+1,n+1⟩;spn in todo;
             divlis(spn)=if divlis(spn) is divl ne Ω then divl else nl
                         with       reading;end∀sp;
        (while todo ne nl)   sp from todo;
        iff                    iscomp?
                  (buildok(sp)),        haveit?
                                  (continue;) ininc+
                                              erasenext?
                                            extend, (continue;);
```

```
iscomp:=inx eq (#prod-1);
haveit:=sp∈ incspans;
ininc: sp in incspans;
erasenext:=prod(inx+1)∈ erase;
extend:sp=⟨spst,prod,inx+1,spnd⟩;
       divlis(sp)=if divlis(sp) is divl ne Ω then divl else nl
                 with  nl;
end iff; end while todo;
end makenewith;

define  buildok(sp);
genodparse external testfn,atfn,compspans,todo,gram,lasterase,
     startat,divlis, n;
/* note use of previously defined macros prod, etc.  */
/* this routine forms digested complete spans in all
   possible ways from an undigested complete span, and
   builds on the new spans in all possible ways  */
spseq=nl; setseq=nl; ixseq=nl; atseq=nl;
spseq(1)=sp;setseq(1)=seqof (lastparts(spseq(1)));;
ixseq(1)=1; atseq(1)=atrib(spseq(1);setseq(1)(1));
extend (spseq,setseq,ixseq,atseq);
atset=nl; test=testfn(prod); atfnc=atfn(prod);nparts=#prod-1;
attrue={⟨j,atseq(nparts+1-j)⟩ ,1≤j≤nparts};
if test(attrue) then (atfnc(attrue)) in atset;;
(while advance(spseq,setseq,ixseq,atseq))
attrue= {⟨j,atseq(nparts+1-j)⟩ ,1≤j≤nparts};
if test(attrue) then (atfnc(attrue)) in atset;;
end while advance;;
/* now one forms complete spans and tests to see if they
   are new */
newcomp= {⟨spst,prod,spnd,atr⟩ ,atr∈atset}
             - compspans;
```

```
/* and builds all elements on these new spans */
(∀spnew ∈ newcomp) <spstl,prodl,spndl,- > =spnew;
(∀g ∈gram,0≤j≤lasterase(g)|  g ∈startat(spstl) a g(j+1)
     eq prodl(#prodl))
spn= <spstl,g,j+1,n+1>;spn in todo;
     divlis(spn)=if divlis(spn) is divl ne Ω then divl else nl
           with spnew; end ∀g;
( ∀ spinc ∈ incspans | spinc(4) eq spstla
     (spinc(2)(spinc(3)+1)) eq prodl(# prodl))
spn=< spinc(1),spinc(2),spinc(3)+1,n+1> ;spn in todo;
     divlis(spn)=if divlis(spn) is divl ne Ω then divl else nl
           with spnew;
end ∀spinc; compspans=compspans u newcomp;
return;
end buildok;
define extend(spseq,setseq,ixseq,atseq);
genodparse external divlis, term;

/* this takes the last element of spseq; chops off
   the division list element referenced by the last element
   of  ixseq, making this the next element of spseq,
   using its division list to calculate setseq, setting
   the next element of ixseq to 1; and calculating
   one more element of atseq */
n=#ixseq; ixn=ixseq(n); spn=spseq(n);
seqn=setseq(n); divlelt=seqn(ixn); nmax=#(spn(2))-1;
(n<∀m≤nmax)  ixseq(m)=1;
iff          diveltnl?
      nuloff,    spnelterm?
              oneoff,    eltoff;
diveltnl:=divlelt eq nl;
nuloff:spn= <spn(1),spn(2),spn(3)-1,spn(4)> ;
     atseq(m)=nl; /* nl attribute belongs to nl subpart */
spnelterm:=(spn(2)(m)) ∈ term;
```

```
oneoff:spn=<spn(1),spn(2),spn(3)-1,spn(4)-1>;
     atseq(m)=divlelt;
eltoff: <divst,-,-,atr> =divlelt;
     spn=<spn(1),spn(2),spn(3)-1,divst>;
     atseq(m)=atr;

end iff;
spnseq(m)=spn;
               setseq(m)=seqof(lastparts(spn));
end ∀m;return;
end extend;

definef advance(spseq,setseq,ixseq,atseq);

/* this advances the last element of spseq; if extension is impos-
   sible, it cuts one or more elements off the sequence, then
   advances and extends. The process ends if the sequence
   becomes null, in which case f is returned as the function
   value; otherwise t */
n=#ixseq;ixn=ixseq(n); seqn=setseq(n);
ixn=ixn+1;
iff          ismore?
         newelt+                    canback?
       diveltnl?                 backup+    fail,
   nulat,    spnelterm?          moreis?
        oneat,  eltat,    ext,         (to canback;);
ismore:=seqn(ixn) is divelt ne Ω;
newelt: ixseq(n)=ixn;
diveltnl:=divelt eq nl;
nulat: atseq(n)=nl;
spnelterm:spn=spseq(n);prd=spn(2);=prd(n) ∈ terms;
oneat: atseq(n)=divelt;
eltat: atseq(n)=divelt(4);
canback:=(#ixseq)gt 1;
```

```
backup: spseq(n)=Ω;setseq(n)=Ω;ixseq(n)=Ω;
     atseq(n)=Ω;n=n-1;ixn=ixseq(n)+1;
                              seqn=setseq(n);
moreis:=seqn(ixn) is divelt ne Ω ;
ext: ixseq(n)=ixn; atseq(n)=if divelt eq nl then nl
     else if (spseq(n)(2) ∉ term then divelt else divelt(4);
fail: return f;
end iff; extend(spseq,setseq,ixseq,atseq); return t;
end advance;
definef lastparts(span);
genodparse external divlis;
/* this returns the set of last parts associated with a given
    span; which is divlis(span) unless the span covers a null
    string of symbols, in which case it is a set consisting
    of the single element nl   */
<st,-,-,nd> =span;
return if st eq nd then {nl} else divlis(span);
end lastparts;
definef getspans(topspan);
genodparse external testfn,atfn,compdiv,newdiv,amb;
/* this is routine which flags all the spans which enter
    into a given span, and also cleans the division list,
    but not completely, since this might require an elaborate
    expansion  */   /* note use of macros prod, etc., defined
    above  */
todo={topspan};
/* compdiv gives initial split-off of digested complete span;
    newdiv the validated divisions of its incomplete initial parts */
(while todo ne nl) next from todo; next in spans;
<st,prd,endd,att> =next;
     sp=<st,prd,#prd-1,endd>;
/* now we use a process like buildok which calls out
    the divisions which pass all tests and give the specified
    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)));
ixseq(1)=1;atseq(1)=atrib(spseq(1),setseq(1)(1));putifterm(1);
extend (spseq,setseq,exseq,atseq);
test=testfn(prod);atfnc=atfn(prod);nparts=#prod-1;
attrue= {<j,atseq(nparts+1-j),1≤j≤nparts};
if test(attrue)then if atfnc(attrue) eq att then
     ndivs=ndivs+1;compdiv(next)=compdiv(next) with setseq(1);
(1<∀j≤nparts) newdiv(spseq(j))=if newdiv(spseq(j)) is newd ne Ω
     then newd else nl with setseq(ixseq(j));putifterm(j);;;end if test
(while advance (spseq,setseq,ixseq,atseq))
attrue= {<j,atseq(nparts+1-j)>,1≤j≤nparts};
if test(attrue) then if atfnc(attrue) eq att then
     ndivs=ndivs+1;
     compdiv(next)=compdiv(next) with setseq(ixseq(1));
     (1<∀j≤nparts) newdiv(spseq(j))= if newdiv(spseq(j)) is newd
     ne Ω then newd else nl with seqseq(ixseq(j));putifterm(j);;;
          end if test; end while advance;
/* the above can easily be put into a more efficient form */
if ndivs gt 1 then amb=t;; end while todo;
return;
block putifterm(j); /* auxiliary block to collect token-type
                        information */
if    prod(nparts+1-j) ∈ term then
      termat= <(spseq(j)(4))-1,nl,nl >;
     <termat(2),termat(3)>=setseq(ixseq(j));
/* which makes up
     <token number,terminal type,attributes >   */
     termat in wdkind; end if prod;
end putifterm;
end getspans;
definef atrib(span,divelt); genodparse external term;
/*  auxiliary routine to calculate attribute  */
iff                  diveltnl?
               nulat,     spnelterm?
                    termat,       compat;
```

```
diveltnl:=divelt eq nl;
nulat: return nl;
spnelterm:=span(2) ∈ term;
termat: return divelt;
comput: return divelt(4);
end atrib;
/* seqof merely sequences an unordered set of elements */
```