SETL NEWSLETTER # 27 May 3, 1970

Code for the Postparse Setup Procedure J. T. Schwartz

(Postparse metalanguage analysis)

A. Lexical scan driver.

```
/*lexical scan driver for modified lexical scanner nt */
definef nextoken; /*external declarations should be supplied */
if giveno ne Ω then giveno = giveno-1; return outgive(giveno);;
[find:]    if n cnow is cstring(nn) ϵ {'=','[','/'} then return nt;
     go to {<'=',eqℓ>,<'[',bkt>,<'/',slsh>}(cnow);
[eqℓ:] if n (tk is cstring(j is nonblank)) ϵ {']',':'} then
     return nt;; /*else*/
[tree:] do treed; if tk ne ':' then do saveup(tk);;
     block treed; gstate=treedes; nameyet= f; isopt=0;
     /*option counter*/ nn = j+1; end treed;
     block saveup(tok); <giveno, outgive(giveno+1)>=
     <giveno+1,<'special',tok>>; end saveup;
     return <'special', '=:'>;
[bkt:]  if n (tk is cstring(j is nonblank)) eq '=' then return nt;;
     /*else*/ go to tree;
[slsh:] if (twotok is cstring(nn+1)+ $cstring(nn+2)) eq '//'
     then j=nn+2; do treed; return <'special', '///'>;;
/*else*/ threetok = twotok + cstring(nn+3); st =
if threetok ϵ {'ob/', 'rs/', 'th/'} then treedes
else if threetok ϵ {'a0/', 'a1/', 'a2/', 't1/'} then act
else if threetok eq 't2/' then ptest else Ω;
if st ne Ω then nn = nn+4; gstate=st; isopt=isopt+1;
do saveup(threetok); if st eq 't2/' then commodd= t;
/* odd-even comma switch */
return if isopt eq 1  then <'special','//'> else <'special','/'>;
end if;  if(threetok+cstring(nn+4)) eq 'end/' then
do saveup('er/'); return <'special','end/'>;;
/* slash within isopt = 0 and comment remain to be treated*/
/* else try comment*/  if cstring(nn+1) eq '*' then
nn=endcomment; if(cstring(nn+1)+cstring(nn+2)) eq '*/'
then nn = nn+2; go to find;; /*else*/
```

```
print 'section skipped owing to ill-structured comment
   delimiter'; go to find;
/* try slash within isopt = 0 */
if isopt ne 0 then return nt;;
/* else */ nn=nn+1; if n nameyet then nameyet = t;
gstate = name; else gstate = treedes;; return <'special', '/'>;
/* the initial lexical states therefore are treedes, act,
   ptest, name */  /* routines nonblank, endcomment follow */
end nextoken;

definef nonblank;  nextoken external cstring, nn;
j = nn; (while cstring(j) eq ' ') j = j+1;;
return j;  end nonblank;

definef endcomment /* skips imbedded comments */
nn = nn + 2;
advance :(while n(c is cstring(nn)) ε {'*','/'})nn=nn+1;;
go to {<'*',star>, <'/',slash>}(c);
star: if cstring(nn+1) eq '/' then return;;
/*else*/ nn=nn+1; go to advance;
slash: if cstring(nn+1) eq '*' then return;;
/*else*/ nn=nn+1; go to advance;
end endcomment;
```

Precedence declaration for preparse setup:

```
tokinf = {<tℓ[tilb(x)], tilb(x)(1),'0'>,
x ε {'///', '=:', '//', '/',
'th/ [ ] end/ ob/ a0/ a1/ a2/ t1/ t2/ rs/',
',', '::', ':'} u {<{'(', '{', '['}, '*'>,
<{')', '}', ']'}, ')'>, <'er/',er>}
typinf = Ω; triple=Ω; endfilesigns=nℓ;
lprinf = {<tℓ x, 2*hd x>,
x ε tilb('( /// =: // / th/ , :: : )')};
rprinf = {<'(', 21>,<{')', er},1>};
lprinf(var') = lprinf(',')+1;
```

	A	O	(	)	.	,	:	'	/	bℓ	;	er	
treedes	go nmotnm	go nxt	do specend	do specend	do perend	do specend	do specend	do qbeg go quoted	do specend	skip	do specend	do dertest skip	
nmotnm	cont	cont	end	end	end	end	end	end	end	end	end	end	
nxt (int)	end	cont	end	end	end	end	end	end	end	end	end	end	
name	go nl	go nl	go nl	go nl	do elend	go nl	go nl	do qbeg go quoted	skip	skip	go nl	do dertest skip	
nl	cont	cont	cont	cont	do nlend	cont	cont	end	end	cont	cont	do dertest skip	
act	go al	go al	go al	go al	go al	do specend	do itest2 go al	do qbeg go quoted	go al	skip	go al	do dertest skip	
al	cont	cont	cont	cont	cont	cont	do itestl cont	do quotend cont	do sltest cont	cont	do etest cont	do dertest skip	
ptest	go al	go al	go al	go al	go al	do comont	do itest2 go al	do qbeg go quoted	go al	skip	go al	do dertest skip	
quoted	cont	cont	cont	cont	cont	cont	cont	do quer-test cont	cont	cont	cont	do dertest skip	

B. Lexical table to be used by the inner scan routine nt

C. Action code invoked in table B.

```
  {<perend,'curpointer=curpointer+1; token=":"; state="special";
          action = "end";'>,
 <specend, 'token=cstring(curpointer); curpointer=curpointer+1;
          state = "special"; action = "end";'>,
 <elend, 'token=".elided"; curpointer=curpointer+1;
          state = "nl"; action="end";'>,
 <nlend, 'curpointer=curpointer+1; action="end";'>,
 <etest, 'if nonblank  eq "," then action = "end";
     definef nonblank; rpak external curpointer, cstring;
 j = curpointer; (while (c is cstring(j)) eq ' ')j=j+1;;
 return c; end nonblank;'>,
<sltest, 'threetok=cstring(curpointer+1) + cstring(curpointer+2)
   + cstring(curpointer+3); if threetok ∈ {"th/", "ob/",
 "a0/", "a1/", "a2/", "t1/", "t2/", "rs/"}
or  (threetok + cstring(curpointer+4)) eq "end/"
  or (2 first threetok) eq '//' then action = "end";;'>,
<itest1, 'if type(cstring(curpointer+1)) eq "0" then
          action = "end";'>,
<itest2, 'if type(cstring(curpointer+1)) eq "0" then
   getint = t; token = ":"; curpointer = curpointer + 1;
    state = "special"; action = "end"; end if;'>
<quotend,  advance : token = token + cstring(curpointer);
   curpointer = curpointer+1; (while(c is cstring(curpointer))ne"""
 doing curpointer = curpointer+1;) token = token + c;;
 if cstring(curpointer+1) eq c /*double quote case */
 then token = token+c; curpointer=curpointer+1;
          go to advance; end if;'>,
<comout, 'curpointer=curpointer+1; state="special";
 action = "end"; token=if commodd then "::" else ",";
  commodd = n commodd;'>,
<qbeg, 'curpointer=curpointer+1;'>,
<quentest, 'curpointer=curpointer+1; if cstring(curpointer) ne
   '''''''' then action = "end"; end if;'>,
```

```
<dertest, 'if cstring(curpointer+1) ne er then return;;
/*else*/ action = "end"; outgive(2) = <'special','er/'>;
 outgive(1) = <'special','er/'>;
 curpointer = curpointer + 2; if token ne nulc then
  giveno = -; outgive(3) = "/"; else giveno =3; token="/";
  end if;'>}
```

Note: The nt routine should be patched as follows:

```
initially  getint = f;  (replace state = next by:)
if getint then getint = f; state=nxt; else state=gstate;;
```

D. Preparse precedence declaration

Precedence declaration for postparse setup:

```
tokinf = {<tℓ[tilb(x)],tilb(x)(1),'o'>,
x ε {'///', '=:', '//', '/ ',
 'th/ [ ] end/ ob/ a0/ a1/ a2/ t1/ t4   rs/',
 ',', '::', ':'} u {<{'(', '{', '['}, '(>,
 <{')', '}', ']'}, )'>, <'er/',er>}
typinf = Ω; triple =Ω ; endfilesigns = nℓ;
lprinf = {<tℓ x, 2*hd x>,
x ε tilb ('( /// =: // / th/ , :: : )')};
rprinf = {<'(',21>, <{')',er},1>};
lprinf('var') = lrpinf(',')+1;
```

E. Postparse self-description

```
/th/10   ///
grammar =: '///' gramparts.2 / grammar.delimiter. /15
   /a1/ labno = 0;  /* label generation counter */
statesused=nℓ; syntypes=nℓ; allex=nℓ; aset = nℓ; secaltset=nℓ;
       errnode=newat; nodetype(errnode) = '.errornode';
pakbod = nulc; paksw = nulc;                          ///
grampart =: '/' ('th/'[integer])/grammar section.
   key token. threshold specification/2,10
/a0/
   a0yet = f;
/t2/ok = threshold eq Ω; threshold=dec tk:3;,
'contains illegal multiple definition of threshold'
   =: '/' 'end/' /end statement/ 2,12
/t2/ do endtest;, 'contains specified overall errors'
   /a2/ do endclean;
   =: '=:' syndefpart.2/syntactic type description/10
/a1/partno = 0; altset=nℓ; isseq=f; stname='.omitted';
/t2/ok = n isseq;, 'contains unterminated altnernative sequence'
/a2/ do choicets; /* to establish syntactic-type related maps*/
                                                      ///
syndefpart =: ']' syndefp./alternative definition.
   keysymbol. final alternative/10
   /t2/ok = isseq;, 'contains illegal close of alternative
   sequence before opening' /a2/ seq(#seq+1)=alt;
   seq in altset; isseq = f;
   =: '[' syndefp. /initial alternative/10
   /t2/ok = n isseq;, 'contains illegal nested opening
   of sequence of alternatives' /a2/ seq={<1,alt>}; isseq=t;
   =: syndefp. /a2/ if isseq then seq(#seq+1) = alt;
   else alt in altset;;                               ///
```

```
syndefp =: firstgroup. '//' ('/' option.1)
   /alternative.keysymbol.compound alternative/10,2
/a0/partno = partno+1; /*firstgroup builds 'alt' */
/a1/tlyet = f; t2yet=f; alyet=f; a2yet=f;
/t2/ok = partno ne 1; 'involves missing syntactic type name'
   =: firstgroup./t2/ok = partno ne 1;'involves missing
syntactic type name'
   =: [name] /t2/ok = partno eq 1; 'involves misplaced
syntactic type name' /a2/stname =tk :1; stname in syntypes; ///

option =: 'ob/' [integer]/option describer.illformed subpart.
   obligatory subpart specification/10
/t2/ok = n simpl;, 'contains illegal option  following
   simple alternative'
nd = (:1); do obok1;, 'contains an integer out of range';
ok=ok2;, 'contains an integer not specifying a syntactic type'
   = 'ob/' (',' [integer].2)/obligatory subpart list/ 10,2
/t2/ok = n simpl;, 'contains illegal option following simple
   alternative'
nd = (:2); do obok1;, 'contains an integer out of range',
ok=ok2; block obok1; nob=1; ok=t; ok2=t;
(while(intn is desc(nd,nob)) ne Ω doing nob = nob+1;)
if (ndx is nodenume(dec tk intn)) eq Ω then ok=f;
   else if ndx ϵ literals or ndx ϵ lexics or
   desc (ndx,1) ne Ω then ok2 = f; else nd in oblig; end if;
   end while; end obok1; 'contains an integer not specifying a
    =: 'tl/' codeseq. /pretest specification/10   syntactic type'
/t2/ ok=n simpl; 'contains illegal option following simple
  alternative', ok=n tlyet; tlyet=t;, ' contains illegally
  repeated pretest group' /a2/pretests(alt)=tupl;
/*codeseq builds tupl */
    =: 'al/' codeseq. /preaction specification/10
/t2/ok = n simpl; 'contains illegal option following simple
alternative', ok=n alyet; alyet=t;, 'contains illegally
repeated preaction group'/a2/ preactions(alt)=tupl;
```

```
    =: 'a2/' codeseq./ postaction specification/10
/t2/ ok = n a2yet; a2yet=t;, 'contains illegally repeated
  postaction group' /a2/ if lexaltnow
/* lexaltnow, stypenow  set by firstgroup */
    then lexacts(stname) = tupl;
    else if stypenow then sucacts(stname)=tupl; else
    postacts(alt) = tupl;;

    =: 'a0/' codeseq. /preliminary action specification/10
/t2/ok = n a0yet; a0yet=t;, 'contains an illegally repeated
preliminary action group' /a2/ prelacts(stname) = tupl;

    =: 't2/' (',' cmpair.2) /postest specification/
10,2/a1/ tupseq = nℓ; /t2/ ok = n t2yet; t2yet=t;,
'contains an illegally repeated postest group'
/a2/ if lexaltnow then lextests(stname)=tupl; else if stypenow
then suctests(stname) = tupl; else postest(alt)=tupl;
   = 't2/'  cmpair. / postest specification/10
/a1/tupseq=nℓ; /t2/ ok = n t2yet; t2yet=t;,
'contains an illegally repeated postest group'
/a2/ if lexaltnow then lextests(stname) = tupl;
else if stypenow then sucacts(stname) = tupl;
else postest(alt) = tupl;

    =: 'rs/' [integer] / resolution specifier/10
/t2/ndtp = tka  alt; ok=lockey(stname,ndtp) eq Ω;,
'contains illegal repetitive resolution specification for literal
already treated', ok=n simpl;, 'contains a resolution specifica-
tion,illegal for simple alternative', resint=dec   tk:2;
ok=(des is desc(alt,resint)) ϵ literals or desc(des,1) ne Ω;,
'contains resolution specification referring to nonliteral node'
/a2/ lockey(stname,ndtp) = resint;              ///

cmpair  =: cstring. '::' [string] / postest pair. delimiter./
   10/a2/tupseq(#tupseq) = <tupseq(#tupseq),tk :3>;      ///
```

```
codeseq =: ',' cstring.2/ code string group. delimiter./10
/a1/tupseq = nℓ;
/a2/tupl= [cons: #tupseq ≥ j ≥ 1] tupseq(j);
/* cstring builds tupseq */
define seqadd(string); postup external tupseq, pakbod,
          labno, paksw;
pakbod = pakbod + 'lab' + dec labno + ':;' + string+'return;';
paksw = paksw + if paksw eq nulc then nulc else ','
          + '<' + dec labno + ',lab' + dec labno + '>';
tupseq(#tupseq+1) = labno; labno = labno+1; return;
end seqadd;
definef a cons b; return <b,a>; end cons;
    =: cstring. /a2/ tupl = tupseq(1);                    ///

cstring =: ':' cpart.2 / code string. delimiter. /10
          /a1/ str = nulc; /*cpart builds str*/
          /a2/ seqadd(str);
        =: [string] /a2/ seqadd(tk:1);                    ///

cpart =:':'[integer] / code subpart. delimiter. node
          designator/10
  /t2/ n = dec tk:2; ok=nodenume(n) ne Ω;, 'contains
an integer out of range'
   /a2/ str = str +      '(partof(node,'+loctup(anode)+')';
definef locseq(anode); postup external desc, parent;
/* auxiliary function to generate locator */ ℓseq = nℓ;
this = anode; (while (parthis is parent(this)) ne Ω
          doing this = parthis;)
find = 1 ≤ ∃[j] ≤ #desc{parthis} | desc(parthis,j) eq this;
lseq(#lseq+1) = j; end while; return lseq; end locseq;

definef loctup(anode); if (seq is locseq(anode)) eq nℓ then
return 'Ω'; else if #seq eq 1 then return dec seq(1);;
/* else generate tupl */ tupl = '<' + dec seq(#seq);
(#seq > ∀j ≥ 1) tupl = tupl+',' + dec seq(j);;
tupl = tupl + '>'; return tupl; end loctup;
```

```
          =: [string] /a2/ str = str + tk :1;

firstgroup  =: '/'  comptree. namespec. evwt.
   /alternative describer section. delimiter.
          compound alternative describer
   /10 /a0/ simpl = t; lexaltnow = f; stypenow = f;
/a1/ simpl = f; alt=errnode; mult=nℓ; nodenume = nℓ;  parent=nℓ;
          val = nℓ; /*used in building alternative tree */
/* in this case comptree builds 'alt' */ /a2/alt=val(:2);
 do buildfixed; /*to build function 'fixed(node)' for various nodes*/
          =: '[ ]' [name] / lexical alternative/10
/a1/ lexaltnow = t; /t2/ ok = lexalts(stname) eq Ω;,
'illegal repetition of lexical alternative'
/a2/ (ℓt is tk:2) in allex; lexalts(stname) ={ℓt};

          =: '[ ]' (',' [name].2)/lexical alternative list/10,2
/a1/ lexaltnow = t; /t2/ ok = lexalts (stname) eq Ω;,
'illegal repetition of lexical alternative'
/a2/ j=1; lexa = nℓ; (while (ℓa is desc(:2,j)) ne Ω
          doing j = j+1;) (ℓt is tk ℓa) in lexa;
ℓt in allex; end while;
lexalts (stname) = lexa;

          =: '{ }' (',' [name,string].2)/unqualified literal
list/10,2  /a2/ genlits(:2); define genlits(nd);
postup external desc, simpl, nodetype, alt, literals, tk;
simpl = f; alt = newat; aset = nℓ;
j = 1; (while (ℓa is desc(nd,j)) ne Ω doing j=j+1;)
(tk ℓa) in aset; end while; alt in literals; nodetype(alt)=aset;
return; end genlits;
          =: '{ }' [name,string] / unqualified literal/10
/a2/ genlits(:1);

          =: litspec. ':' lexspec. /lexically qualified literal/10
```

```
          =: [tname] /t2/ ok = sesor(stname) eq Ω;
stypenow = t;,'contains illegal repetitive successor type
specification'
/a2/ sesor(stname) = ssor is tk:1; ssor in statsused;
                                                              ///
litspec =: '{ }' (',' [name,string].2)/literal specifier.
delimiter. literal list/10,2/a2/genlits(:2);

          =: '{ }' [name,string]/literal/10/a2/ genlits(:1);
                                                              ///
lexspec =: '[ ]' [name] / lexical specifier. character. lexical
          specification/10,2
/a2/ lextype(alt) = {ℓt is tk :2}; ℓt in allex;

          =: '[ ]' (',' [name].2)/lexical specification/10,2
/a2/ lextype(alt) = lexset(:2); definef lexset(nd);
postup external desc, allex, tk;
j=1; aset = nℓ; (while (ℓa is desc(nd,j)) ne Ω doing j=j+1;)
aset = aset with (ℓt is tk ℓa); ℓt in allex; end while;
return aset; end lexset;
                                                              ///
comptree  =:  'catenation' item.2 /composite alternative describer.
          delimiter. alternative tree section/10
/t2/ ok=t; mt = Ω /*auxiliary for node multiplicities*/; jj=1;
(while (subn is desc(:1,jj)) ne Ω doing jj=jj+1;)
subnd = val(subn);
if mult(subnd) eq Ω then continue;;
/* else */ mt = mult(subnd); mult(subnd) = Ω;
if desc(:1,jj+1) ne Ω then ok=f;;end while;,
'contains illegal multiple element in position other than last',
ok=t; jj=1; (while (litn is desc(:1,jj) ne Ω
doing jj=jj+1;) litnd=val(litn);  if litnd ϵ literals then return;;
/*else*/ end while; ok=f; val(:1) = errnode;,
'lacks required literal element'
```

```
/a2/ j=1; literals=literals less litnd;
(while (ℓn is (desc(:1,j)) ne Ω doing j=j+1;)
if j eq jj then continue;; parent(lnd) = litnd;
desc(litnd, if j ℓt jj then j else jj-1) = lnd; end while;
val(:1) = litnd; minlast(nd) = mt;
                                                              ///
item  =:  litz. ':' lexz. /tree describer item. delimiter.
          lexically qualified literal/11
/a0/val(item) = errnode; multsw=f; /*switch used for
          multiplicity */
      =: '{ }' (',' [name,string].2)/literal list/10,2
/a2/ nd = newnode(:1); putlitz(nd,:2);

definef newnode node; postup external val, nodenume;
nd = newat; nodenume(#nodenume+1) = nd;
val(node)=nd; return nd; end newnode;

define  putlitz(nd,lnd); postup external literals, litname,
          nodetype, tk;
nd in literals; litnume(#litnume+1)=nd; aset=nℓ; j=1;
(while (dn is desc(lnd,j)) ne Ω doing j=j+1;)
aset = aset with tk dn; end while; nodetype(nd) = aset; return;
end putlitz;

      =: '{ }' [name,string]/literal/10
/a2/ nd=newnode:1;  putlitz(nd,:1);

      =: '[ ]' [name]/lexical item/10
/a2/ lexmake(:1);

      =: '[ ]' (',' [name].2)/lexical item list/10,2
/a2/ lexmake(:2), define lexmake(nod); postup external lexics,
   newnode, nodetype,desc,tk,allex; nd; nd=newnode:1;
nd in lexics; aset = nℓ; j=1; (while(sn is desc(nod,j)) ne Ω
doing j=j+1;)  ℓt = tk sn; ℓt in allex; ℓt in aset; end while;
nodetype(nd) = aset; return; end lexmake;
```

```
      =: '( )' comptree. /parenthesized subgroup/10
/a2/ val(:1) = val(:2);

      =: ('[ ]' [name]) ':'[integer] / multiple lexical type /
5,7 /a2/ lexmake(:1); mult(nd) = dec tk:4;

      =: ( '[ ]' (',' [name].2)) ':' [integer] /multiple
lexical list/5,2,7 /a2/ lexmake(:2); mult(nd) = dec tk :5;

      =: ':' litz. lext. [integer] /multiple qualified
literal/10 /a2/ mult(nd)     = dec tk :4;

      =: [tname] /a2/ nd=newnode :1;
nodetype(nd) = tn is tk :1; tn in statsused;

      =: [tname] ':' [integer] / type designator with
multiplicity/10 /ob/1/a2/nd = newnode :2;
nodetype(nd) = tn is tk :1; tn in statsused;
mult(nd) = dec tk :3;
                                                              ///
litz  =:  '{ }' (',' [name,string].2)/literal specifier.
   delimiter. literal list/10,2
/a2/ nd = newnode :1; putlitz(nd,:2);

      =   '{ }' [name,string] /literal/2
/a2/ nd = newnode :1; putlitz(nd,:1);
                                                              ///
lext  =: '[ ]' (, [name]2) /lexical qualification list
          delimiter. lexical list/10,2
/a2/ lexmake(:2);
      =: '[ ]' [name] / lexical qualifier/10
/a2/ lexmake(:1);
```

```
namespec  =:  'catenate' [name,string] [name,string]
           [name,string] /name specifier.period.name group/10
/t2/ ok = partno eq 2;, 'contains compound name, illegal
in all but first alternative', ok=(n2 is tk :3) ne '.elided';,
'contains illegally elided complaint name'
   /a2/ if (n1 is tk :2) eq '.elided' then n1=stname;;
if (n3 is tk :4) eq '.elided' then n3= n1;;
namesynt(stname) = n1; keysymbol(stname) = n2;
altname(alt) = n3;

      =:  [name,string]
   /t2/ ok= partno ne 2; 'contains simple name, illegal for
first alternative', ok=(nm is tk :1) ne '.elided';,
'contains illegally elided name' /a2/ altname(alt) = nm;
                                                          ///
evwt =: ',' [integer]   evidential weight specifier. comma.
           evidential weight group/10
   /t2/ j = 1; ok = t; gset=nl;
(while (intn is desc(:1,j)) ne Ω doing j=j+1;)
int = dec tk intn; if (lit is litnume( j )) eq Ω
then ok = f; else do gath(lit); end if; end while;
gathalt(alt) = if ok then get else nl;, 'contains integer not
corresponding to literal node'
block endtest; /*final test operations - threshold, states
defined and used, print allex */
print 'the following lexical types are used:', allex;
ok = t; if threshold eq Ω then print 'threshold
definition omitted'; ok = f;;
if set is  {x ϵ statsused | n x ϵ syntypes} then print
'syntactic types are used but not defined; these are:', set;ok=f;;
if set is {x ϵ syntypes|n x ϵ statsused} then print 'syntactic types
are defined but not used; these are:',set; ok=f; end endtest;
block endclean; /*final cleanup operation-assemble final paktext*/
paktext= 'define rpak(ncall);
go to {' + paksw + '(ncall)' + pakbod + 'end rpak;';
print paktext; end endclean;
```

```
      =: [integer] /t2/ int = dec tk: 1;
ok = if (lit is litnume(int)) eq Ω then f else t;,
'contains integer not designating literal node'          ///

block gathalt(alt); stup = nodetype(lit); seq = locseq(lit);
(#seq ≥ ∀j ≥ 1) stup = <seq(j),stup>;;
stup =<stup,int>; stup in gset; end gath;

block choicets;
/* 'aset' is the 'altset' of the postparse algorithm*/
(∀a ∈ altset) if atom a then aa = Ω; do procalt(a);
  else aa=a;  (1 ≤ ∀n ≤ #a) do procalt(a(n));;
  end if; end ∀ a;
block procalt(ax);  hedset = nodetype(ax);
 (∀ ht ∈ hedset)
 if (ℓk is lockey(stname,ht)) eq Ω then
/* no special resolution */
  aset(stname,ht) = if(as is aset(stname,ht)) eq Ω
    then nℓ else as with if aa eq Ω then ax else aa;
else /* exists special resolution */
  resset = nodetype(desc(ax,ℓk));
(∀ rt ∈ resset) secaltset(stname,rt) =
 if (se is secaltset(stname,rt)) eq Ω then nℓ else se
   with if aa eq Ω then ax else aa; end ∀rt; end if;
end ∀ht;
end procalt;  end choicets;

block buildfixed;
/* builds function 'fixed(node)' for various composite
 elements of an alternative. this has value 0 if a given
 syntactic type occurs only once; Ω if the type is preceded
 by a multiple occurrence; otherwise it notes the number
 of like preceding items which the alternative contains.
 ** might usefully be generalized to apply to lexical
 elements also **/
```

```
typect = nℓ;  firsnodz = nℓ;
(1 ≤ ∀n ≤ #nodenume | n (cnode is nodnume(n)) ϵ lexics
and n nodenume(n) ϵ literals and desc(nodenume(n),1) eq Ω)
/* so that only composite nodes are processed */
typ = nodetype(cnode);
if (tc is typct(typ)) eq Ω then firsnodz(typ) = cnode;
 tc = 1; go to settc;
 else if tc = -1 then continue;;
settc : fixed (cnode) = tc;
typct(typ) = if minlast(parent(cnode)) ne Ω then
     tc+1 else -1;
end ∀n;  /* now go thru and first to zero if still 1 */
(∀ payr ϵ firsnodz | typct(hd payr) eq 1)
 fixed (tℓ payr) = 0;;
end buildfixed;
```