SETL Newsletter Number 67 February 22, 1972

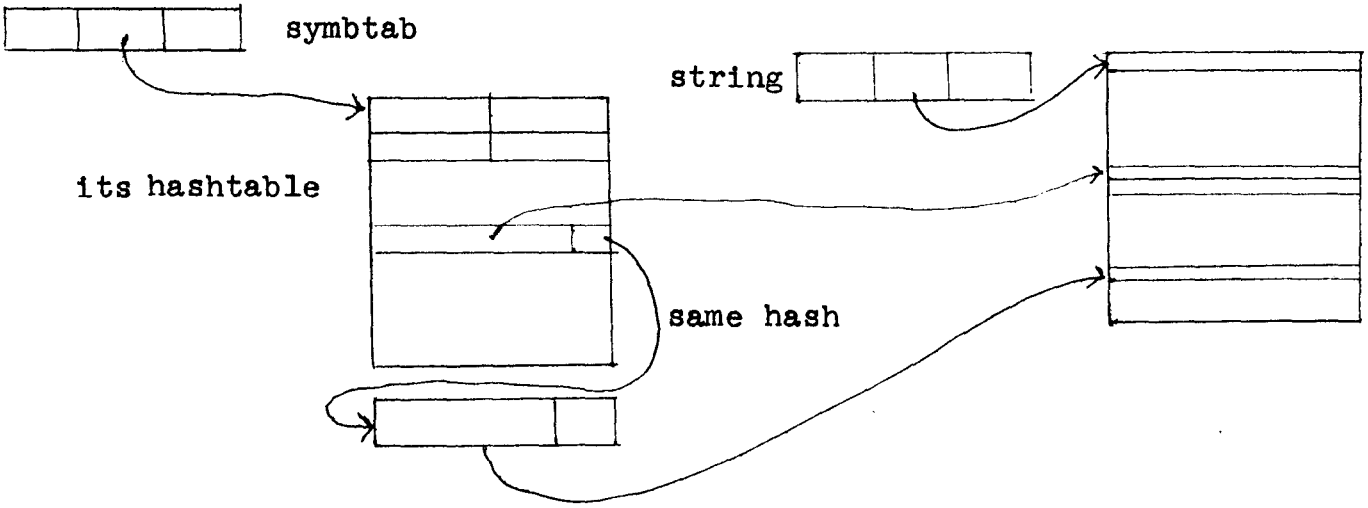
Data structures of the SETL compiler for the LITTLE version. Kurt Maly

To supplement the data elaboration of the SETL programs we will give here for all main data structures the implementations of them on the LITTLE level.

Scan - the lexical scanner.

We will try to strike a balance between pure LITTLE data structures and use of the SETL library routines which will procure a good efficiency with respect to time as well as space. Whenever we say 'set' or 'tuple' a data object of the SETL working stack is to be understood. This stack is used by all library routines and provides dynamic storage allocation and garbage collection therefore.

symbtab: is a set whose members are standard short integers. These are pointers (offsets) to what looks like a long character string where the actual tokens are stored.



Instead of using the library routine augment we use our own:

x=hashsymb(token).,

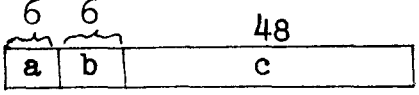
Function hashsymb(token).,

Essentially the same as augment, but when token is hashed it is compared with the words in 'string' to which the element of symtab points. If no match is found the token is appended to string and its offset entered as a new member of the set and same is returned as result;otherwise just the corresponding offset is returned.

To make it even more efficient we will actually have two symtabs.
symbtabl which will be used for tokens of lexical type operator, delimiter, nullop, objtype, er, ef and which is preset.
symbtab: which will be used for all other tokens.
The number of both will point to the same 'string' and both sets will have the same structure.

string: As already mentioned, it is a long character string and again a different concatenation routine is required. Initially 500 words will be requested for it. And every time a token is added, we first check for overflow and if that is the case we double the size.

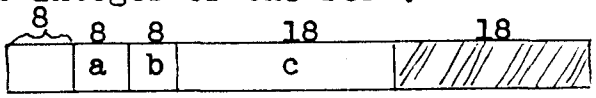
token: is a LITTLE array of dim 132)., size 160.,
the first word of it is of the following form:



a: 6-bit integer indicating how many computer words this token takes up.
b: 6-bit integer. If token belongs to symbtabl it will define its 'kind' (used in the preparser) otherwise it will be zero.
c: the token itself as a character string. The last word of the token is padded with blanks to the right. If the token is not a name but a quoted character string the first 12 bits of c give the

number of characters, if it is a bitstring those 12 bits give the number of bits.

result: is a short integer of the form:



a: is an 8-bit integer giving the lexical type of the token.
b: is an 8-bit integer giving the 'kind' of the token if defined at this time otherwise 0.
c: is an element of symbtab(i.e., pointer to 'string').

Note: left out is the data field. It is only used for user-defined operators and constants; and this has been eliminated in the following way: user-defined operators are now of lexical type 'uop' and its name appearing as token. The condensation procedure and small parts of 'nextoken' have been changed in order to take care of it. Constants are stored now such that their internal form (i.e., bitstring) are now their names.

table: is a LITTLE array of dim (#states); size(120);
it is indexed by:i=.F.typ, 8, table(state).,

```
i=1 ;                go to endc
i=2                  go to endcl
i=3                  go to loop
i=4                  go to contc
4<i≤ #states+4 is c    state=i-4, go to contc.,
i>c                  go by i-c(l1,...,lk).,
```

Each card is read into size card(6)., dim card(81).,

type: is a LITTLE array of dim (64)., size (60).,
it is indexed by char=card(n)., typ=type(char).,
and the value of typ is of the form 8 * k+1 in order to

make the indexing of 'table' easy.

Nt - the macro expander.

macro: since its domain is a sparse subset of symbtab it is implemented as a set. The elements of the range are of the form of 'mbody'. In the SETL algorithm they have the form <afn,mbody>. Instead of storing afn we mark each element in mbody which is a formal argument with a new lexical type and instead of the symbtab pointer we place the ordinal of that argument there.

argstack: is a tuple each element being a tuple of integers (actually the one-word result from scan).

all other stacks: are tuples where the values are integers (see argstack).

bras: is a set. The value of the mapping is then used in a go by i (ℓ1, ...)., statement.

Nextoken - the editor.

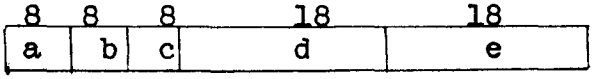
swiset, iterset, bras: like bras in nt.
rel, opset, lparset, rparset: true subsets of symbtab.
blostack: like argstack in nt.
all other stacks as in nt.

Preparser - the parser proper.

The kind of a lexical item is an integer and for those items which do not have a preassigned kind (see scan) the mappings symbkind, typkind, sets are used.

rprec, rprec1, lprec, lprec1, gross, mask: tuples of dim (# kinds)

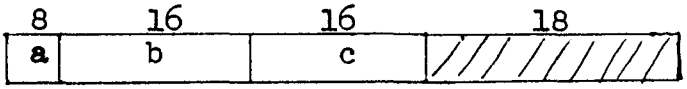
statstak: is a tuple whose elements are of the following form:



a: integer type
b: lexical type (will be zero if intermediate node)
c: kind (if item represents intermediate node it will be the numerical equivalent of 'var')
d: pointer to string (nodname (integer) if intermediate node)
e: state

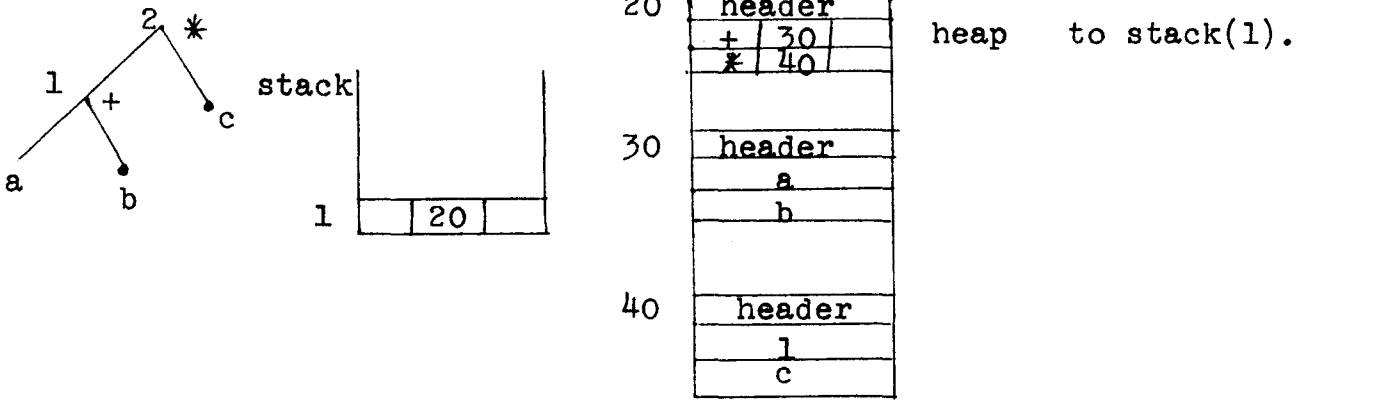
bakstack: a LITTLE array of dim(2) whose entries are 'result' from scan.

desc: is a tuple whose elements are tuples whose elements in turn are of the form mentioned in 'statstak'. In the rootwords of those tuples (which represent an intermediate node with its descendants) we record the nodetype of this node as follows:



a: type
b: nodetype (equals pointer to string)
c: pointer to object in heap.

example:



desc corresponds to stack(1).

As has already been said all those tuples will be SETL objects but, in general, we will not call the SETL-library routine 'of' which evaluates (or assigns) 't(i)';rather, we will take advantage of our knowing the types of the arguments and call a lower level routine which evaluates t(i) for tuples. In some cases

we will even bypass the calls and write instead in-line code or call a tailor-made routine for evaluation of said expression.

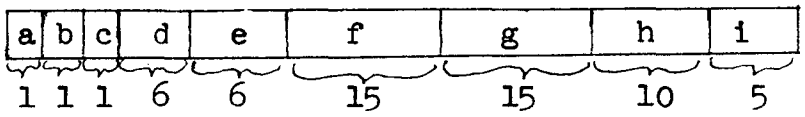
Postparser - checking and diagnosing routine.

The syntactic types will be integers in the range of 1 to # synts , alternative nodes integers from 1 to # alts, and all other nodes integers from # alts+1 to # nodes.

tests: tests will be a LITTLE array of dim (# of all tests), (Index to rpak.) The actual mappings (like pretests, etc.) will be of dim (#alts) or dim (#synts) and size(15)., 10 bits give an index to 'tests' and 5 bits give the number of tests for that alternative or syntactive type.

acts: acts will be a LITTLE array of dim (# of all acts), size(30); 15 bits as index to rpak, 15 bits as pointer to 'string'. The rest is the same as for tests.

minlast, literals, lexics, oblig, altname, fixed, lextype: are collected in a LITTLE array 'nodinfo' of dim (# nodes) of size(60). Each entry is of the form



a: flag for literal
b: flag for lexical
c: flag for obligatory
d: minlast
e: fixed
f: lextype (structured like tests)
g: altname
h: pointer to idesc (if intermediate node).
i: # descendants

namesynt,keysymb: LITTLE arrays of dim (# synts), size(18) value is pointer to 'string'.

sesor: LITTLE array of dim (# synts), size(18);

Value is integer (syntactic type).

lexalt: converted to a tuple (of dim(# synts))
whose members are sets of short integers (alternatives).

gathalt: converted to a tuple (of dim(# alts))
whose members are sets of pairs the first of which is a pointer to 'string', the second an evidential weight.

altset: converted to a tuple (of dim(# synts))
whose members are mappings of integers (i.e., nodetype(node)) → sets (i.e., sets of: alternative nodes or tuples of alternative nodes).

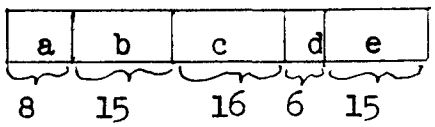
secaltset,lockey: converted to a mapping (i.e., set of triples <syntactic type,key(integer),value like that of altset> or <syntactic type,nodetype(node),integer> respectively.

idesc: is a LITTLE array of dim(# of nodes) and size(18). The values are integers (node names) and brothers occupy consecutive locations.

inodetype: converted to a tuple (of dim(# nodes))
whose members are either sets of tokens (member of symbtab) or integers (syntactic type).

typecount: tuple of integers (of dim(# synts))

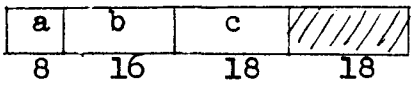
partseq: tuple of integers of the form



a: type
b: integer (syntactic type)
c: integer (node)
d: integer (non-multiple node if 0)
e: integer (alternative node)

tastack: tuple whose members are pairs the first of which is like the value in the test mappings (e.g., suctests) the second like in the act mappings (e.g., sucacts).

mstack: tuple whose members are integers of the form



a: type

b: integer

c: integer (value of namesynt)

Appendix The Postparser

```
definef postparse(syntype,node,nodenum);
parser external lockey,altset,secaltset,sesor,threshold,
keysymb,namesynt,pretests,preacts,posttests,postacts,
rpark,gathalt,minlast,fixed,oblig,literals,lexics
prelacts, sucacts,suctests,lexalt,lextype,idesc,inodetype;
parser external mstak,desc,nodtype,symbtab;
initial setup;mstak=nl;
allsynt=hd[keysymbol];
allalts=[u: x ∈(altset u secaltset)]x(∃) u [u: x∈ tl[lexalts]]x;
new=allalts;(while new ne nl) node from new;
if node n∈ allnodes then node in allnodes;new=new u
idesc {node} ;;
twigs={node ∈ allnodes | idesc(node,1) eq Ω};
pile allsynt;string;pin allsynt;pile allnodes;string;
pin allnodes;pile allalts;elt allnodes; pin allalts;
pile twigs;elt allnodes;pin twigs;
range tests(alt);alt aselt allalts;stores pretests,lextests,
postests set tupl(int,string),preacts,postacts,lexacts set int,
minlast int 6, altname string, gathalt set tupl
(elt symbtab,int 5);
range name(synt);synt aselt allsynt;stores prelacts,
sucacts set int,suctests set tupl(int,string),
lexalt set elt allalts,sesor elt allsynt,keysymbol
namesynt string;
oblig=oblig assub allnodes;
literals=literals assub allnodes;
lexics=lexics assub allnodes;
range lexfix(x);x aselt twigs;stores lextype set string,
fixed int 5;
range inodetyp(alt);alt aselt allnodes;stores
```

```
inodetype obj;
range idesc(x,j);x aselt allnodes; j int;stores idesc
elt allnodes;
end initial;                                    partseq=nl;
synt=syntype;tastack=nl;mstak=nl;typecount=nl;partreg=nl;
range tastack(i);i int;stores tastak tupl(set tupl
      (int,string),set int);
range mstak(j);j int;stores mstak tupl (int 12,
string);
range typecount(j);j int;x aselt allsynt;stores
typecount int;
range partseq(j);j int;stores partseq tuple (elt
allsynt,int,int6,elt allalts);
[syntry:] (1≤∀n≤ if prelacts(synt) is pa ne Ω then 0 else # pa)
   rpak(pa(n));
   iff          (nodetype(node)is noty eq Ω)?
        (lexalts(synt)is setalts ne Ω)?   islockey?
    (lextry=t;go to findalt;),maynext,  keypres?   arealts?
                                  aresecalts? maynext,findalt,maynext,
                               findalt,maynext;
islockey:  lextry=f;=lockey(synt,noty)is keyloc ne Ω;
keypres: = desc(node,keyloc)is keydesc ne Ω;
aresecalts:  key=if nodetype(keydesc) is x ne Ω then x
          else keydesc(2);= secaltset(synt,key)is setalts ne Ω;
arealts:  = altset(synt,noty)is setalts ne Ω;
end iff;
  [maynext:]  iff       (sesor(synt)is synt ne Ω)?
              testoracts?        (nodenum ge 0)?
       stackacts,syntry,    goguess,  (return f;);
  testoracts:=suctests(syntype)is tsts ne Ω or sucacts(syntype)
      is acts ne Ω;
```

```
stackacts:  tastack(#tastack+1)= <if tsts eq Λ then nl else
tsts,if acts eq Λ then nl else acts>;go to syntry;
end iff;
[goguess:] guess(node,syntype,score);if score
lt threshold then print prefix(syntype,nodenum)+
'involves an ill-formed or missing'+keysymbol(syntype);;
return f;
[findalt:] partseq=nl;
if ∃ [alt] ∈ setalts | (atom alt and matches(alt,
node,partseq) then go to matched;end if ∃ ;
if n ∃ [altseq] ∈ setalts,1≤[j]≤if atom altseq then 0
else #altseq | matches(altseq(j),node,partseq)then
go to maynext;end if n;alt=altseq(j);
[matched:]rpak[preacts(alt)];
if nodnum ge 0 then mstak(#mstak+1)=<nodnum,
namesynt(synt)>;;
ok=t;typecount=nl;
(1≤∀ n≤#partseq)<subsynt,subnode,start,subalt,->=
partseq(n);if start eq 0 then
subnum=if nodnum lt 0 then nodnum else fixed
(subalt);ok=ok and postparse(subsynt,subnode,
subnum);else (start≤∀ j≤#desc{node})
subnum=ifx        (nodnum lt 0)?
              (nodnum), (typecount(subsynt)is tc  ne Ω)?
                    (tc+1),  (1);end ifx;
if nodnum gt 0 then typecount(subsynt)=subnum;;
ok=ok and postparse(subsynt,desc(node,j),subnum);
end start≤∀j;end else;end 1≤∀n;
if n ok then return f;;
if n lextry then [;(1≤∀j≤if postests(alt)is pt ne Λ then 0 else #pt)
<act,msg,->=pt(j);rpak(act);
```

```
[;iff                    (n ok)?
              (nodnum ge 0)?      (continue;),
          printfix,    (return f;);
printfix:/   print prefix(synt,nodnum)+msg;returnf;
end iff; end l≤∀j;
block oktest;-]  rpak[postacts(alt)];block endtest
(postests,postacts);-]else endtest(lextests,lextacts);end else;
(l≤∀n≤#tastack)<tests,acts,->=tastack(n);
(l≤∀j≤#tests)<act,msg,->=tests(j);rpak(act);
oktest;rpak[acts];end l≤∀n;if nodnum ge 0
then mstak(#mstak)=Λ;end if;return t;
end postparse;
definef matches(alt,node,partseq);
postparse external pretests,rpak;
if n matcher(alt,node)then partseq=nl;return f;;
(l≤∀j≤if pretests(alt)is pt ne Λ then #pt else 0)rpak(pt(j));
if n ok then partseq=nl;return f;;end l≤∀j;return t;
definef matcher(alt,node);
matches external partseq;postparse external desc,nodtype,
inodetype,idesc,minlast,oblig,literals,lexics,lextype;
if idesc(alt,1)eq Λ then go to twig;;
if nodetype(node) is  nty eq Λ then return f;;
if nty n∈ inodetype(alt) then return f;;
desreq =#idesc{alt}  is ndesca+if minlast(alt)
is min ne Λ then min-1 else 0;
if #desc{node}   is ndesc lt desreq or (min eq Λ  and
ndesc ne desreq) then return f;;
if l≤∀j≤ndesca-if min ne Λ then 1 else 0
n matcher(idesc(alt,j),desc(node,j)) then return f;;
if min eq Λ then return t;;
if idesc(alt,ndesca) is descalt ∈ oblig then if
```

```
ndesca≤∃j≤ndesc | n postparse(inodetype(descalt),
desc(node,j),-1) then return f;;end if idesc;
if descalt ∈ lexics then if ndesca≤∃j≤ndesc | n matcher
(descalt,desc(node,j)) then return f;else return t;;
end if descalt;
partseq(#partseq+1)=<inodetype(descalt),node,ndesca>;
return t;
[twig:]if alt ∈ literals then return node(2) ∈ inodetype(
alt)and if lextype(alt)is lext eq Ω then t else
node (1) ∈ lext;;
      if alt ∈ lexics then return node (1) ∈ inodetype(alt);;
      if alt ∈ oblig then if n postparse(inodetype(alt),
node,-1) then return f;;end if alt;
partseq(#partseq+1)=<inodetype(alt),node,0,alt>;
return t;
end matcher;
end matches;
definef partof(node,locator);postparse external idesc;
nod=node;(1≤∀n≤#locator)nod=desc(nod,locator(n)
);;return nod;end partof;
definef prefix(syntype,nodnum);postparse external mstak,namesynt;
res='the'+if nodnum eq 0 then nulc else (dec
nodnum+affix(nodnum))+namesynt(syntype)+
'of'+[+:#mstak≥ j≥1]('the'+if mstak(j)(1)is h
eq 0 then nulc else (dec h+affix(j))+mstak(j)(2)
+'of')+'this'+mstak(1)(2);mstak=Ω;return res;
end prefix;
definef affix(n);return if n lt 3 then {<1,' "st'>,
<2,' "nd'>,<3,' "rd'>}(n) else ' "th';end affix;
define guess(node,syntype,score);
postparse external sesor,threshold,nodenum,altname,
```

```
desc,altset,secaltset;
gather={<pickup(node)>};(1≤ ∀j≤#desc{node})
nodel=desc(node,j);gather=gather with <pickup(nodel),
j>;(1≤ ∀k≤#desc{nodel})gather=gather with <pickup
(desc(nodel,k)),j,k>;;end 1≤ ∀j;
score=0;synt=syntype;
(while synt ne Ω doing synt=sesor(synt);)
allalt=[u: x ∈ tl[altset{synt}]x u [u: x ∈ tl
[secaltset{synt}])x;
(∀alt ∈ allalt)if fit(alt,gather)is newscore gt
score then synkeep=synt;keep=alt;score=newscore;
end if;end ∀alt;end while;
if score lt threshold then return;;
message=prefix(synkeep,nodnum)+'is probably a ' +
altname(keep)+'but';
failmatch(keep,node);print message;return;
definef pickup(node);postparse external nodtype;
return if nodetype(node) is x ne Ω then x else node(2);
end pickup;
end guess;
definef fit(alt,gather);postparse external gathalt;
gath=gather;totscore=0;
(∀x ∈ gathalt(alt)) if ∃[g] ∈ gath | x(1) eq g(1) then
totscore=totscore+x(2);gath=gath less g;end if;end ∀x;
return totscore;
end fit;
define adm(string);guess external message;
charad=if message(#message-3:3) eq 'but' then ' '
else ' , ' ;message=message+charad+string;
return;
end adm;
```

```
define failmatch(alt,node);
postparse external desc,nodetype,idesc,inodetype,
oblig,minlast,literals,lextype,lexics,namesynt;
if idesc(alt,1) eq Ω then go to twig;;
if #desc{node} is ndesc eq 0 then ndesca=#idesc
{alt};go to numwrong;end if;
if n nodetype(node) is ntn ∈ inodetype(alt) is nta
then adm(ntn+'found where'+∃ nta+'or equi-
valent is expected');return;end if;
ndesca=#idesc{alt};mult=minlast(alt);
(1≤∀j≤(ndesca-if mult ne Ω then 1 else 0) min
ndesc) failmatch(idesc(alt,j),desc(node,j));;
if mult eq Ω then go to numwrong;;
if idesc (alt,ndesca) is descalt ∈ oblig then
(ndesca≤ ∀j≤ndesc | n postparse(inodtype(descalt),
desc(node,j),-1)) adm(dec(j-ndesca+1 is jx)
+affix(jx)+'part found not to be' + namesynt(
(inodetype(descalt))+'as required');end ndesca;
go to numwrong;end if;
if descalt ∈ lexics then (ndesca≤ ∀j≤ndesc)
failmatch(descalt,desc(node,j));end ndesca;;
[numwrong:]if ndesc lt ndesca then adm(if mult
ne Ω then 'at least' else nulc+dec(ndesca-ndesc)
+' parts missing');return;end if;
if mult eq Ω and ndesc gt ndesca then adm
(dec (ndesc-ndesca)+'superfluous parts');return;;
if ndesc lt (ndesca+mult-1)then adm('only'+
dec (ndesc-ndesca+1)+'parts found where'+
dec mult+'are required');;return;
[twig:]if alt ∈ literals or alt ∈ lexics and
desc(node,1) ne Ω then go to false twig;;
```

```
if alt ∈ literals and n node(2) is litfound ∈inodetype
(alt) is litneed then adm (litfound+'found where'
+∋ litneed+'or equivalent is required');return;;
if alt∈ literals and lextype(alt) is lext ne∩ and
n node(1) is typef ∈ lext then adm('literal'+
litfound+'of lexicaltype'+typef+'found where
lexical type'+∋ lext+'or equivalent is required');
return;;
if alt ∈ lexics and n node (1) is typef ∈ inodetype
(alt) is lext then adm ('lexical type' + typef+
'found where'+ ∋ lext+'or equivalent is required');
return;;
if alt ∈ oblig and n postparse(inodetype(alt),
node,-1) then adm('part found not to be'+
namesynt(inodetype(alt))+'as is required');
end if;return;
[falsetwig:]req=if alt ∈ literal then 'literallike'+
∋inodetype(alt) else if alt∈ lexics then 'lexical type'+
∋inodetype(alt) else nulc + 'or equivalent';
adm('composite structure containing'+nodtype
(node)+'found where'+req+'is required');return;
end failmatch;
```