Examination of the SETL algorithms of O.P.II reveals a number of cases in which the pursue construction suggested in Newsletter 133 allow algorithms to be written in substantially more succinct and/or transparent ways. This newsletter will record several such cases. In the algorithms to be presented, we use a notation somewhat different from that suggested in NL 133; specifically, we write

(1) 

(pursue)

as

(1')

(∀∀),

and write

(2) 

(pursue ∀x∈s | C(x))

as

(2')

(∀∀x∈s | C(x),

and write

(3) 

(pursue x ≤ ∀n ≤ y | C(x))

as

(3')

(x ≤ ∀∀n ≤ y | C(x)),

J. Schwartz
July 7, 1974
etc. For the original forms of the algorithms to be given, the reader is referred to O.P.II, pages to be cited. In many of these algorithms we will see that use of the $\forall \forall$ construction suppresses one level of control that would otherwise be explicit; this often allows a few auxiliary variables, and the operations which update them, to be dropped. The $\forall \forall$ construction proves to be useful in describing 'transitive closure' constructions of the most general sort, as well as systems of numerical or combinatorial equations to be solved by iteration. It is not useful for describing algorithms whose specific procedural structure is of importance either for efficiency or because the input to be processed has some specifically serial character. It may be observed that procedural viewpoints, whose ultimate root probably lies in the sequential nature of hardware, pervade many attempts to define algorithms. Note also that recursive routines can in some cases be replaced by $\forall \forall$ blocks.

A. Predominator finder (O.P.II, p 265)

```prolog
def define predoms(nodes, entry);
  /* the successor function cesor is assumed to be global */
  notfor = n; notfor(entry) = nodes less entry;
  ($\forall \forall$x $\in$ nodes, $y \in$ cesor(x))
      notfor(y) = notfor(x) less y + (notfor(y) $\ominus$ n);
  end $\forall \forall$;
  dom = n;
  ($\forall y$ $\in$ nodes) dom(y) = nodes - notfor(y) less y;
  return dom;
end predoms;
```


```prolog
def define nodparse (input, gram, root, syntypes, spans, divlis, em);
  spans = (\langle n, x, n+1 \rangle, 1 \leq n \leq \#input) in len. x $\in$ syntypes(input(n))$\setminus$
  divlis = n;
```
(∀ ∀ ε spans, spend ε spans {s(3)}, type ε gram(s(2) is typ1, spend(1) is typ2})

<s(1), type, spend(2)> is newsp in spans;
<newsp, typ1, s(3), typ2> in divlis;

end ∀ ∀;

/* check on grammaticality */
if n (< l, root, ilen + 1 > is topspan) ε spans then
    <spans, divlis, amb> = <nl, nl f>; return;
end if;

/* else clean up set of spans and determine ambiguity */
amb = f; goodspans = {topspan};
(∀ ∀ s ε goodspans)
    if #(divlis{s}) is dl) gt 1 then ambig = t ;;
    goodspans = goodspans + {[+d d l]}{<s(1),d(1),d(2)> + <d(2),d(3),s(3)>};
end ∀ ∀;
return;
end nodeparse;

(this is 17 lines excluding comments; the algorithm of pp. 161-162 is 34 lines).


define earleyparse (input,gram,root,syntypes,spans,divlis,amb);
spans = {<n,x,n+1>, l< n <(# input) is ilen,xε syntypes{input[n]};
divlis = nl; syms = {root};
ultbegin = {<x(3),x(1)>, xε gram};
descends = {<x(3),x(2)>, xε gram} + ultbegin;
(∀ ∀ y ε ultbegin) ultbegin = ultbegin + {<y(1),z>,z ε ultbegin{y(2)}}};
(∀ ∀ s ε syms) syms = syms + descends{syms};
startat = <ultbegin (root)>;
(∀ ∀ s ε spans, spend ε spans {s(3)},
    type ε gram(s(2) is typ1, spend(1) is typ1) * startat(s(1));
    <s(1) is newbeg, type, spend(2)> is newsp in spans;
    <newsp, typ1, s(3), typ2> in divlis;
    startat(newbeg) = startat(newbeg) + grammatical.
ultbegin[(x(1), xє gram[typl] | x(2) estartat(newbeg))];
end \forall \forall;
/* now check for grammaticality */
(the remainder of this algorithm is identical with the corresponding portions of the code given above for the simple nodal span parse)
end earleyparse;

D. Topological analysis algorithm. O.P.II, pp. 262-263

Here we recast our algorithm substantially, in a mathematically equivalent but very much less efficient form. Our new algorithm merely pushes the representative of each 'block' B in our collection to the smallest (earliest row, earliest column) possible position in the connected region similarly colored bricks containing B. This algorithm, as revised, extends easily to three and more dimensions. Attempts to optimise the algorithm are bound to raise interesting problems.

define a \texttt{smaller} b; /* lexicographic comparison for bricks */
\begin{align*}
<x,y> &= a; <u,v> = y; \\
\text{return } x &\lt u \text{ or } (x \equiv u \text{ and } y \lt v); \\
\end{align*}
and \texttt{smaller};
define a \texttt{touche\$} b; /* adjacency function for bricks*/
\begin{align*}
<x,y> &= a; <u,v> = b; \\
\text{return } (y &\equiv v \text{ and } \text{abs}(u-x) \equiv 1) \text{ or } \\
&\ (\text{abs}(v-y) \equiv 1 \text{ and } (x \equiv u \text{ or } \\
&\ \text{if } v \text{ mod } 2 \equiv 1 \text{ then } x \equiv (u-1) \text{ else } x \equiv (u+1));
\end{align*}
end touche;
define topanalyse (color, nxrows, nocols);
/* bricks is assumed to be global */
bricks = \{<nr,nc>, \lt nr \lt nxrows, \lt nc \lt nocols\};
repre\texttt{sents} = \{<b,b>, b \in \texttt{bricks}\};
\[(\forall \forall b \in \text{bricks}, b' \in \text{bricks})
\]
\[
(b \text{ touches } b' \text{ and } \text{color}(b') = \text{color}(b) \text{ and } \text{represents}(b) \text{ smaller } \text{represents}(b'))
\]
\[
\text{represents}(b') = \text{represents}(b);
\]
end \forall \forall;

primebricks = \{b \in \text{bricks} | \text{represents}(b) \equiv b\};
inside = n\emptyset;
(\forall p \in \text{primebricks} | (\exists b \in \text{bricks} | b \text{ smaller } p \text{ and } b \text{ touches } p))
\]
\[
\text{inside}(p) = \text{represents}(b);
\]
end \forall;
return inside;
end topanalyse;

(Excluding comments, this is 24 lines; the algorithm given on pp. 262-263 is 31 lines.)

In casting about for dictions allowing general purpose languages of higher level than SETL to be defined, the idea of using nonprocedural dictions which select a desired item out of some very large space of objects is bound to crop up. This is a technique which is used from time to time, often with spectacular success, in mathematics (consider, for example, the definition of homology groups by the use of singular homology.) In programming terms however this approach does not always seem convenient, and at any rate the transformation into usable algorithms of solutions defined using this technique raises problems which belong to mathematics rather than to routine optimisation.