J. Schwartz
July 2, 1974

SETL Newsletter # 133A

Additional *Pursue* Block Examples.

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 succint 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) $$(\text{pursue})$$

as

(1') $$(\forall\forall),$$

and write

(2) $$(\text{pursue } \forall x \varepsilon s \mid C(x))$$

as

(2') $$(\forall\forall x \varepsilon s \mid C(x),$$

and write

(3) $$(\text{pursue } x \le \forall n \le y \mid C(x))$$

as

(3') $$(x \le \forall\forall n \le y \mid C(x)),$$

etc. For the original forms of the algorithms to be given, the reader is refered to O.P.II, pages to be cited. In many of these algorithms we will see that use of the ∀ ∀ 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 ∀ ∀ 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 ∀ ∀ blocks.

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

```
definef predoms(nodes, entry);
/* the sucessor function cesor is assumed to be global */
notfor = nℓ; notfor(entry) = nodes less entry;
(∀ ∀x ε nodes, y ε cesor(x))
            notfor(y) = notfor(x) less y + (notfor(y) orm nℓ);
end ∀ ∀;
dom = nℓ;
(∀y ε nodes) dom(y) = nodes - notfor(y) less y;;
return dom ;
end predoms;
```

B. Nodal Span parse, Simple form. (O.P,II,pp. 161-162).

```
define nodparse (input, gram, root, syntypes, spans,divlis, amb);
spans = {<n,x,n+1>, 1 ≤ n ≤ (#input) is ilen, xε syntypes(input(n))};
divlis = nℓ;
```

```
(∀ ∀sεspans, 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 (< 1, 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
                    +[+:dεdℓ]{<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).

C. Nodal span parse, Earley form (O.P.II, pp. 163-164).

```
define earleyparse (input,gram,root,syntypes,spans,divlis,amb);
spans = {<n,x,n+1>, 1≤ n ≤(#input) is ilen,xε syntypes{input(n)}};
divlis = nℓ; symbs = {root};
uℓtbegin = {<x(3),x(1)>, xεgram};
descends = {<x(3),x(2)>, xεgram} + ultbegin;
(∀ ∀ y ε ultbegin) ultbegin = ultbegin + {<y(1),z>,z ε ultbegin{y(2)}};;
(∀ ∀ s ε symbs) symbs = symbs + descends [symbs];;
startat = <ultbegin {root}>;
(∀ ∀ s ε spans,spend ε spans {s(3)},
      type ε gram(s(2) is type1,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) orm nℓ +
```

```
ultbegin[{x(1), xε gram{typl} | x(2) εstartat(newbeg)}];
end ∀ ∀;
/* 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. Toplogical 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.

```
definef a smaller b; /* lexicographic comparison for bricks */
        <x,y> = a; <u,v> = y;
         return x lt u or (x eq u and y lt v);
and smaller;
definef a touches b; /* adjacency function for bricks*/
        <x,y> = a; <u,v> = b;
         return (y eq v and abs(u-x) eq 1) or
                (abs(v-y) eq 1 and (x eq u or
                     if v mod 2 eq 1 then x eq(u-1)  else x eq(u+1)));
end touches;
definef topanalyse (color, nrows, ncols);
/* Bricks is assumed to be global */
bricks = {<nr,nc>,1≤ nr ≤nrows,1≤ nc ≤ ncols};
represents = {<b,b>, b ε bricks};
```

```
(∀ ∀ b ε bricks, bprime ε bricks|
       b touches bprime and color(bprime) eq color(b) and
        (represents(b) smaller represents (bprime)))
                 represents (bprime) = represents(b);
end ∀ ∀;
primebricks = {b ε bricks | represents(b) eq b};
inside = nℓ;
(∀ p ε primebricks |(∃ b ε bricks | b smaller p and b touches p))
       inside(p) = represents(b);
end ∀;
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 gereral 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 suceess, 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.