GEHANI79
Gehani, N.; A High Level Data Structure - the Grid; Computer Languages, Volume 4, pp. 93-98.

This paper proposes a data structure called the grid, and claims that programs written using grids are smaller, clearer and more easily modifiable than those that are not. A notation for grids in Pascal is given, and some examples are used to illustrate the effectiveness of this data structure.

In order to achieve program clarity, the data structures we use should provide a natural representation of our data. But in many applications, we need to represent data which is not array-like. It may have an irregular shape, or even be unconnected. Clearly, an array or group of arrays could be used to simulate such data, but this would not be a natural representation, and would thus result in several problems. Generating and testing for valid indices would have to be done explicitly, clarity and modifiability would be lost, and parts of the area (for example borders) could not be referenced collectively.

The grid solves all of these problems. Grids are declared as a union of some component arrays (not necessarily connected), minus the union of some other arrays. For example,

```
GRID [1..50,1..100 MINUS 20..30,20..80] OF real
```

defines a rectangular ring, 20 units thick. In order to make certain grid definitions easier, the bounds of a component's dimensions can be integer functions of another dimension's coordinates. For example,

```
GRID [i in 1..30, i..30]
```

defines a triangle with corner points (1,1), (1,30), (30,30).

A FOR statement is defined which allows for easy iterations through the elements of a grid. The form of this statement is

```
FOR i1, i2, ... , in IN domain(g) DO S
        where g is a grid,
        i1...in are integer variables
                (loop counters),
        n is the number of dimensions of g, and
        S is a statement.
```

This FOR causes S to be executed once for every valid subscript of the grid, with each of the i variables holding the value of its associated dimension's coordinate. The iterations are done in an unspecified order, however, we can specify that the iteration over each dimension be ordered either in increasing or decreasing value by adding "inc" and "dec" clauses. For example

```
VAR  b :GRID [1..4,1..4 MINUS 2..3,2..3 PLUS 6..6,6..6]
                              of char
FOR i, j IN inc(1) dec(2) domain(g) DO S
```

has the same effect as

```
FOR i = 1 to 6 DO
  FOR j = 6 DOWNTO 1 DO
    IF indomain (g, i, j)
    THEN S
```

where indomain is a built in function which returns true if the subscript is valid for the given domain.

Finally, a notation is given for naming groups of grid components for easy reference to parts of a grid. When we wish to reference a section which cannot be described as a group of the grid's components, a notation for defining and referencing subgrids is also available. In both of these extensions, no extra space need be allocated.

It is clear from examples given that the grid reflects non array-like shapes naturally and allows shorter, more readable programs. More importantly however, changes in a grid declaration require no alterations to the code which accesses that grid.