A LANGUAGE FOR PARALLEL PROCESSING OF ARRAYS, EMBEDDED IN PASCAL

by

Leonard Uhr

Computer Sciences Technical Report #365

September 1979

A Language for Parallel Processing of Arrays, Embedded in PASCAL

Leonard Uhr
University of Wisconsin

Abstract

A new programming language is described for image processing, pattern recognition and scene description on parallel array computers. The language extends PASCAL to handle parallel procedures of the following general form:

```
||procedure {procedurename};
  {constants, types and variables declared here, as always}
begin
  ||dim {declaration of dimensions for parallel processing};
    ||set {arrays-to-be-assigned-results}
       := {structures in arrays, and compounds of structures};
    ||if  {structures and compounds} {conditional test}
      ||then {arrays-to-be-modified, and modifications}
        ||else {arrays and modifications};
||end;
```

Examples are given of such parallel code, and how it can be used. These extensions appear to fit reasonably well into standard PASCAL, allowing the programmer to intermix parallel and serial code. The parallel statements are presently executed by a serial computer. But, as the new parallel array hardware becomes available, they will execute (with orders of magnitude increases in speed) directly, in true parallel fashion.

Introduction

This paper describes an experimental language (called PascalPL) that extends PASCAL with constructs to process parallel arrays. PascalPL allows the programmer to code procedures that effect parallel operations over arrays of integer (or, optionally, boolean) values of the sort commonly used in image processing, pattern recognition and scene description programs. Statements can specify sets of relative locations (with respect to each cell of the array), and compounds of such sets, embedded in assignment statements, or in conditional statements.

PascalPL does not purport to handle efficiently a wide range of numerical operations. Nor does it attempt to handle the problems of parallel-serial processing by networks in general. Rather, arrays of information (e.g. as input by a television camera, and transformed by arrays of hardware [or virtual] processors) are the major type of parallel structures that it addresses. But since it is embedded in PASCAL all the facilities and power of PASCAL are available to the programmer.

PascalPL presently exists as a program (coded in PASCAL and running under the UNIX operating system on the University of Wisconsin Computer Sciences Department's VAX) that inputs a legal PascalPL program and outputs a legal PASCAL program that contains the code the PascalPL pre-processor output (which will now effect the parallel operations called for by the PascalPL code). Its special parallel constructs (to be described below) include parallel assignment (||set .. := ..;), conditional (||if .. ||then .. ||else ..;), input (||read(..);) and output (||write(..);) statement-types. These all execute operations, in true parallel fashion, on arrays of information. This is effected on the VAX, as on any other serial computer with only one processor (the "central processing unit" or "cpu"), by setting up temporary arrays (to keep processes parallel) and embedding the processes within nested do-loops. (See the Appendix for examples.)

The Need for Higher-Level Languages for Parallel Arrays

Several large arrays of parallel processors are today just beginning to become available. CLIP4, a 96 by 96 array (Duff,

1976, 1978), has each of its almost 10,000 processors fetch, in parallel, information from any subset of its 8 nearest neighbors plus its own memory and execute a logical instruction. ICL's DAP (Flanders et al, 1976, Reddaway, 1978) is a 64 by 64 array with a good bit of capability for numerical as well as for image-processing problems.

MPP (designed for NASA by Goodyear-Aerospace, to be delivered in 1982; see Fung, 1977) will be a very fast 128 by 128 array. Goodyear-Aerospace is also building ASPRO, a 2000 by 1 reconfigurable array with 32 processors and a flip network on a single chip, for delivery by 1981 (anon, 1979). (ASPRO is closely related to Goodyear's Staran-E [Batcher, 1974], essentially using lsi technology to miniaturize a Staran into a one cubic foot box.) Two pipelined and specially designed computers have also been built for image processing: PICAP (Kruse, 1976, 1978), and the Cytocomputer (Sternberg, 1879).

When actual array-processing hardware is connected to a serial computer that executes PASCAL programs then the parallel constructs in PascalPL can be translated to directly invoke the parallel hardware, which will execute them directly (and with two to five orders of magnitude increases in speed).

For the present it seems of interest to begin to develop parallel languages, to ease the burden of coding parallel programs (even if for a serial computer), to explore what kinds of parallel constructs are useful, and to examine what mixes of parallel and serial constructs programmers actually use.

It is not at all clear whether present-day languages that have been developed for serial computers should be extended to handle arrays and networks of parallel processors, or whether entirely new languages should be developed. In the long run we will inevitably see entirely new languages, that reflect the new hardware architectures of computers as well as our advances in understanding language design. But for the next 3 or 5 (or 10?) years it seems likely that the most powerful systems will use a serial computer (to serve as "host," file manipulator, compiler, simulator, etc., as well as to execute the serial portions of programs) along with any hardware arrays and networks of processors. It therefore seems appropriate to take advantage of the serial computer, and of the power of the newer languages that have been developed for serial computers.

PASCAL (Wirth, 1971, Jensen and Wirth, 1973) seems the best choice for the language-to-be-extended. It is already widely used and promises to be used increasingly, especially on microprocessors of the sort used to build networks and arrays. It is both powerful and relatively efficient. It has been extended, in concurrent PASCAL (Brinch-Hansen, 1973) and Modula (Wirth, 1977), to begin to handle networks of processors. It is being extended, in Telos (Travis et al, 1977) to handle a wide range of artificial intelligence and data base management tasks. At the first of what is expected to be a series of annual conferences on parallel architectures and higher-level languages for image processing the consensus of those who felt a modern language was needed was that it should be embedded in PASCAL (until a completely new language emerges), and a working group was formed

with that goal (see Duff, 1979). Finally, since it seemed desirable to implement an experimental version of such a language, to explore its problems and its uses, a decision had to be made as to the specific language to extend. PASCAL seemed a reasonable choice.

A Brief Background Review of Related Work

A large number of languages have been coded to handle parallel array processes (see Preston, 1979). These have tended to be at the assembly language level, or to be subroutine calls in Fortran. But they include a number of languages that are quite simple and straightforward to use, and offer the user great power (to a large extent because the parallel arrays, with their very large numbers of processors, are so powerful). Most of these languages execute entirely on a particular machine with parallel hardware (or on a simulation of parallel hardware), and are directed toward image processing. (The languages that have been developed for array processing of numerical problems, e.g. APL [Iverson, 1962] and Glypnir [Lawrie et al, 1975] and Actus [Perrott and Stevenson, 1978] for the Illiac-4 super-computer, will only be mentioned here.)

Kruse and his associates (see Gudmundsson, 1979) have developed a very nice algol-like language in which programs can be coded that call both the Picap parallel processor and the serial host computer. It contains two major parallel constructs, for 1) logical and 2) numerical operations, with the format:

```
(type)   1 2 3
         4 5 6  = result
         7 8 9
```

That is, the programmer is asked to specify a 3 by 3 array of 9 values (these may be boolean values, or integers specifying weights or inequalities), plus an operation and a result. This closely reflects the nearest-neighbor structure of the Picap-1 array (and of most other of today's hardware arrays).

Uhr (1979) has developed a higher-level language (also called PascalPL, but let's refer to it as PascalPL.0) for the CLIP parallel array that makes use of "compounds" and "implications" similar to, and precursors of, some of the constructs in the present PASCAL-based PascalPL.

Reeves (1979) has developed a system that extends APL to handle image processing as well as numerical processing of arrays. He argues (personal communication) that APL, since it handles arrays quite naturally, may well be preferable to PASCAL as the "host language" into which parallel constructs should be embedded. But since PASCAL is far more widely available, and far more consonant with the feelings that most people have today about what is "good sructure," it seems the better choice. APL is, however, a language that can suggest some useful constructs that might be embedded in a PASCAL-based system.

Levialdi and his associates (1978) are developing Pixal, which consists of parallel extensions embedded in Algol-60 (which they are using chiefly because PASCAL is not available on their computer). Pixal uses a "frame" construct that allows the programmer to specify a set of relative locations (not only nearest-neighbors) to be looked at everywhere (that is, relative to each cell in the array), and a "mask" construct with which the

programmer can specify a set of weights to be applied to the relative locations specified in the coordinate structure.

Douglass (1979) has proposed extensions to PASCAL to handle parallel arrays, building on earlier proposed extensions by Pratt and Ison (see Ison, 1977) to handle networks. These include a "fork" operation to set up new processors that are executing different sets of instructions in parallel, and a "split" operation, to invoke whole sets (e.g. arrays) of processors to execute the same set of instructions, but each on different data (e.g. different local regions of a visual image). The programmer can specify "windows" (sets of [relative] coordinate locations).

Schmitt (1979) has also described extensions to PASCAL to define and then execute operations on any arbitrary network structure, rather than limitting the programmer to arrays.

Almost all of these languages use as their key construct an operation on a set of relative locations. The languages designed for a specific hardware array will build in the interconnection pattern of that array (usually, as in the case of Picap-1, the 3 by 3 sub-array of the nearest neighbors). Pixal's "mask" and Douglass' "window" generalize this to any set of arbitrarily distant neighbors. (But when such a language is actually executed on a hardware-parallel array this gives excessively long sequences of nearest-neighbor shifting operations.) Uhr added constructs for the convenient compounding and implying of information over several different arrays. Douglass and Schmitt make suggestions for constructs to handle much more general networks, as well as SIMD (Single-Instruction-Multiple-Data-Stream) arrays.

A Description, with Examples, of PascalPL

PascalPL is designed to handle arrays, and sets of arrays, that contain binary, grey-scale and/or numerical values. It allows one to code procedures that look at and compound sets of relative locations (around each "pixel" cell) in a single array, and to compound sets of these sets across several arrays, using either arithmetic or boolean operations.

It seems best to introduce the reader to PascalPL by starting with very simple examples, gradually introducing its full set of constructs and features.

An Overview of PascalPL Constructs

A PascalPL program looks like a PASCAL program, except that it contains several new constructs (all announced and made visible by two vertical [parallel] bars, e.g.: ||procedure..; ||set..; ||if..; ||read..; ||write..;). This means that the programmer must declare all the necessary constants, data types and variables, and strictly follow all the conventions of PASCAL [e.g., a program must begin with "program {programname}(input,output{...});" and end with "end."]. The programmer can code as much as she/he desires in ordinary PASCAL. Only when parallel constructs are desired do any deviations occur. These parallel constructs are handled as follows:

The procedure that will contain these (one or more) parallel constructs must be declared:

```
||procedure {procedurename};
```

Any time after this procedure's "begin" statement, a "dimension declaration" statement must be placed, e.g.:

```
||dim    [0..127,0..127];
```

Then come, interspersed with ordinary PASCAL statements, parallel constructs of the sort:

```
||read{..};
||write{..};
||set  {array(s) assigned to} := {compound of array(s)..};
||if {compound of arrays) (ineq) }
    ||then {array(s) modified}
       ||else {arrays modified on failure - optional};
```

Procedures cannot be nested within parallel procedures (this restriction will be lifted when PascalPL is actually embedded in PASCAL, and can conveniently use stacks for declarations).

A Very Simple Example Program

The following program makes only the simplest use of PascalPL constructs. (See the Appendix for the PASCAL program into which the PascalPL preprocessor translates it.)

```
program simple(input,output);
{the programmer must declare the array data structures used}
{  for this program they are: image, negative, edgedimage  }

procedure sayhello;
begin
  writeln('hello');
end;

||procedure demonstrate;
begin
  ||dim    [0..2,0..2];
    ||read(image);
      ||write(image);
    ||read(negative);
  writeln('the image and the negative image have been input');
    ||set edgedimage := image - negative;
    ||write(edgedimage);
    ||set image, negative := 0;
||end;

begin {program}
  sayhello;
  demonstrate;
end.
```

An Example of Output From the Simple Program Above

```
hello
TYPE IN INTEGERS FOR  image[ 0.. 2, 0.. 2]
   1   2   3       were input to row  0
   4   5   6       were input to row  1
   7   8   9       were input to row  2
ARRAY =  image  contains:
   1   2   3 ;
   4   5   6 ;
   7   8   9 ;
 END OF ARRAY.
TYPE IN INTEGERS FOR  negative[ 0.. 2, 0.. 2]
   5   5   5       were input to row  0
   5   5   5       were input to row  1
   5   5   5       were input to row  2
ARRAY =  edgedimage  contains:
  -4  -3  -2 ;
  -1   0   1 ;
   2   3   4 ;
 END OF ARRAY.
```

This program first outputs 'hello' and then inputs a 3 by 3 array, naming it "image". (As it is presently implemented to run interactively, it outputs the message to TYPE an array, and outputs the inputs, to verify.)

The next statement "||write(image)" outputs the array stored in image, in array form. "||read(negative)" inputs the array that it names "negative". Now the program outputs that the two arrays have been input. (Note that this is a regular PASCAL "writeln(..);" command. It illustrates how regular PASCAL statements can be interspersed.)

Now, for each cell in the array "edgedimage" the program subtracts what negative contains from what image contains. Next it re-initializes the two arrays image and negative, so that each of their cells contains a zero. Finally, it outputs the array named edgedimage.

This is a trivial example, and doesn't begin to indicate any of the more powerful ways that the parallel-assignment construct (||set) can be used. But it shows how PascalPL's parallel constructs can be intermixed with standard PASCAL, and it does show how input and output are extended, in a straightforward but what appears to be satisfactory way, to handle arrays.

The Assignment Statement

An assignment statement contains a set of "arrays-to-be-assigned-results" (called "assignees") to the left of the assignment operator (":=") and a "compound-of-structures" (called "compounds") to the right of the ":=".

Now let's examine a sequence of more powerful assignment statements (See the Appendix for the PASCAL code output by PascalPL for selected statements.):

```
||set vert := image[+(0:1,0:0,0:-1)];
```

(This statement looks, for each cell in image, at the 3 relative locations specified, sums what it finds, and stores the result in the corresponding cell of vert.)

```
||set cross := vert[+(0:0,0:2,0:-2)] * hor[+(0:0,2:0,-2:0)];
```

(Sums the 3 relative locations in vert, does the same for hor, then multiplies these sums and stores the result in cross.)

(Note that:

```
||set array1 := array2 (op) array3;
```

is equivalent to:

```
||set array1 := array2[0:0] (op) array3[0:0];  )
```

```
||set vert, hor, image := 2 * image[+(0:1,1:0)] - #average;
```

(This illustrates how several arrays can be assigned the values computed by the compound, and how the compound can include constants and variable identifiers [each must be preceded by '#'].)

```
||set gradient := image[+(0:0*12,1:0*-2,0:1*-2,-1:-1*-1,...)];
```

[This will get a weighted difference between the center cell (weighted +12) and the 4 square neighbor cells (weighted -2 each) and 4 diagonal neighbors (weighted -1 each) (only the center and 3 of the 8 neighbors are shown).]

```
||set featurei, featurej, labelk
      := featurem[*(4:-3*2>5,-5:7*3>14,0:0*21>112)];
```

(Multiplies what is found in each relative location by the specified weight and accepts the result only if it exceeds the specified threshold; then stores the sum of these results in the corresponding cells of featurei, featurej and labelk.)

To summarize: An assignment statement consists of an arbitrarily long compound (to the right of the assignment operator ":=") of array-specifications. An array-specification consists of an array name followed (optionally) by a "structure". A structure consists of an (integer or boolean) operator followed by a set of relative locations. Each relative location in an integer array can, optionally, be followed by an operator and a weight and/or an inequality and a constant.

One or more arrays can be named to the left of the assignment operator. Each will be assigned the result of the set of operations on arrays specified in the compound. (An option that is probably not very useful also allows [for integer arrays] an ar-

ithmetic operator followed by an integer, to modify each result by the specified constant before storing it in its corresponding cell.)

The Conditional Statement

A conditional statement can also be constructed, with the form:

```
||if {compound} {optional inequality} ||then {modifications};
```

or

```
||if {compound} {opt. ineq.} ||then {modif.} ||else {modif.};
```

For example (see the Appendix for PascalPL's PASCAL output):

```
||if featurei[+(0:1,0:-1)] * featurej[+(1:0,-1:0)]  >  11
    ||then  labeli+19, labelj*2, labelk-33
      ||else  labell*2, labelm+27;
```

The conditional statement first computes the compound (for each cell in the arrays). Then, if an inequality is specified, it tests it and (only when it is satisfied or, when using boolean arrays, if the compound is true) makes modifications to the arrays following the "||then". If there is also an ||else, what follows it is modified (only when the conditional fails, or is false).

The modifications can specify an operator and an integer (e.g. labeli+19, indicating "add 19 to labeli", or vert*36, to multiply vert by 36). Reading and Writing Arrays

The two simple constructs:

```
||read({arrayname});  and
||write({arrayname});
```

input and output the specified array.

Constructs that Declare Parallel Procedures

The ||procedure {procedurename};
construct must declare each procedure that contains one or more ||if... or ||set... statements. (Its purpose is to signal the PascalPL pre-processor that a parallel procedure will have to be set up, so it can declare that procedure here, with a "forward". This avoids any problems that might arise if the programmer used the same name in the declarations that PascalPL uses in declaring the temporary lists and data structures that it must use. If these extensions were embedded in the PASCAL compiler itself this declaration could be eliminated or, probably better, combined with the ||dimension declaration.)

The ||dim statement declares the array-type and dimensions of the arrays to be used in the parallel constructs that follow in this procedure. Its shortest and simplest form is:
||dim;
(which declares the previously declared, or default, dimensions and integer arrays).

Its full form is:

```
||dim {arraytype}merge from  {from-setname}[{arraydimensions}]
            to {to-setname} [{shrink-convergence}];
```

The arraytype can be & (for boolean) or * (for integer). The from-setname and to-setname are optional. If used, they are concatenated in front of the names of arrays in a) compounds and in b) assignees or modifieds, respectively. (This is an option that may be useful when sets of arrays are collected into layers or characteristics, and if and when a feature that lets the programmer declare and use them as arrays of records, using the PASCAL "with" construct, is implemented.)

The arraydimensions must be given in the form: [minx..maxx,miny..maxy], optionally declared boolean or integer - e.g., [0..127,0..63 : integer]. (Note that this means the array-type can either be declared before or within the arraydimensions. Whichever seems more convenient and more natural will be retained in the future.) Thus still another alternative declaration is:

```
||dim [0..7,0..15 : boolean] {optional shrink};
```

The shrink-convergence indicates how information is converged from a from-array to a to-array. For example, if it is 2,3 then the x-coordinate of a from-layer cell is divided by 2, and the y-coordinate is divided by 3, to compute the coordinates of the to-layer's cell. (If no shrink-convergence is specified the program will use whatever was last specified. The default, if shrink-convergence was never specified, is 1,1 - that is, coordinates are divided by 1, so that no convergence occurs.)

An end statement: `||end;`
ends the parallel procedure.

The type of border to be used when a relative location-to-be-looked-at lies outside the array is specified by:

```
||border := {bordertype};
```

where bordertype can equal 0 (for 'false"); or 1 (for 'true'); or 2 (for 'what is contained by the nearest cell within the array'). (The default condition is border = 2.)

Suggestions for Possible Future Extensions

As noted above, PascalPL can be simplified and improved when it is embedded within PASCAL itself (this could have been done in the present pre-processor, but did not seem worth the effort in a first experimental system). The symbol table PASCAL builds up could be used to merge the names PascalPL must declare with those that the programmer declares (and also change names that are the same). PASCAL could get each array's type when it is declared; then PascalPL could generate the appropriate temporary arrays and stores (but it may be best to limit the programmer, since there will rarely be more than one physical hardware array present).

If declarations of dimensions, border-types, array-types and shrink-conversion were stacked parallel blocks could be embedded, and treated exactly like ordinary blocks (but this would violate the capabilities of most parallel hardware). Procedures could be generated and then called, to shorten the code (but the present code seems more appropriate for a first experimental system, since it makes clear exactly what serial PASCAL code must be executed to effect each parallel PascalPL procedure). Code could be made more efficient in several places. For example, the temp{orary}store is often not needed; the conditional test need be made only once; a compound could be stored immediately in TEMPARRAY0 when it contains only one element.

The specifications of a mask (what to look for in a set of relative locations) could be handled with 2-dimensional masks of the sort used by Picap. This would be especially convenient if implemented as part of an interactive prompting routine. Masks

should also be declarable, e.g. with ||mask := {mask}; so that the programmer could simply name them, and also could perform operations on them, to transform them.

In addition, the format for declaring array-types and shrink-convergence can be regularized, generalized, and brought closer to standard PASCAL format. One example of a promising version would replace the declarations:

```
||procedure {procedurename};
  .
  .
||dim &merge from [0..15,0..32] to [2,2];
     by
||procedure {name}(dim 0..15,0..15 : boolean; shrink 2,2);
```

(Now "shrink" would be only one of a number of conversion operations the programmer might designate are to be effected, along with, for example, "rotate" or "invert". The programmer should probably also be given the facility to write procedures that compute still other operations, and call them along with the built-in operations.)

PascalPL could also get the dimensions of arrays from the programmer's declarations, so that the procedure could be declared even more simply, e.g.:

||procedure {procedurename} (shrink 3,4); Alternately, shrink (and other operations) could be handled as simple assignment statements (either with or without "||").

The dimensions might be used to specify sub-arrays, and then the programmer be given facilities for coding different sequences of instructions to be executed over different sub-arrays. This

begins to give facilities for programming MIMD (Multiple-Instruction-Multiple-Data Stream) as well as SIMD arrays. In the extreme, the programmer could specify a sub-array of 1 cell, and different instructions for each such (1-cell) sub-array. It is not clear whether this would violate the parallel array structure (such programs could certainly not be executed except with enormous inefficiencies on actual arrays of physically parallel processors). But it seems an interesting step toward a language for more general networks, where the lock-step parallel processes of the array are relaxed. And it makes clear that it is not the MIMD vs. SIMD distinction that is important for efficiency in executing programs (as opposed to simplicity in building one program controller to drive all processors) but rather the need to keep all processors working for (close to) the same amount of time (which is guaranteed when all processors execute the same sequence of instructions).

A parallel version of the standard PASCAL Case statement would probably be desirable, to handle situations where a sequence of embedded '||if..||then..' statements would otherwise be needed. A slight extension should also be made, to allow inequalities over integer values. This would, for example, handle situations where one set of actions should be taken if a result of a compounding operation exceeded some threshold, while different sets of actions should be taken for intervals to successively smaller next threshold(s). Therefore the Case statement should accept inequalities as well as PASCAL character ("char") symbols, e.g.:

```
||case threshold of
  >57 :  modify(hor+23,vert-3,tree+5);
  >21 :  modify(noise+7,hor-3);
  >3  :  modify(noise+11,vert-7);
end  {case}
```

This would be another step toward handling more general MIMD processes.

A ||repeat...||until...; and a ||while...||do...; should be introduced, along with ||if...||then...||else...; statements that applied arbitrary blocks of code to the individual cells, rather than the relatively standard and therefore close-to-equal-in-time modifiers presently programmed.

Discussion

Again, these changes would move the language away from a language for physically parallel arrays (and may well violate the spirit of parallel processing on these arrays) and toward much more general networks of processors. But if the programmer took care to keep all eventualities relatively close in the time they needed for execution, and appriate hardware were available, this might be a good procedure for getting efficient programs. For the language would handle the allocation of processors and message passing automatically and efficiently. And the burden on the programmer would be relatively small - to formulate the program so that it executed through a parallel-window-like set of processes of a size commensurate with the size of the hardware-parallel system (which might be an array, or some other appropriate network).

Should a language for parallel arrays and, more generally, for networks, remain as an extension embedded in PASCAL? The present and projected extensions appear to cohabit rather congenially within PASCAL. PascalPL seems surprisingly simple in the extensions that were needed, and it makes use of standard PASCAL in many ways. A programmer should be able to code in this mixture with little interference between the two systems. And PASCAL is useful when the programmer codes at least some processes for a serial computer (as she/he certainly will, at least until all the very difficult problems of parallel arrays and networks are solved).

But it seems likely that the parallel aspects of computing are of overriding importance, that we are entering a completely new era of parallel networks of computers. We will need to develop completely new parallel algorithms and programs; we will find that whole new types of approaches to problems, and of problems themselves, are now amenable to attack, and invite attack. At some point, possibly quite soon, entirely new languages will be called for. But it seems best to move toward new languages by first extending those that exist, and also trying to combine the features of different types of languages that appear to be relevant (e.g. APL, array languages like Picap's PPL and Clip's CAP4 [Wood, 1977], and multi-processor languages like Concurrent PASCAL and Modula). These endeavours, along with the continuing design of and experience with arrays and networks, should give us the understanding needed to develop entirely new and more appropriate languages.

Appendix

Presently Implemented Options to PascalPL Constructs

A number of optional formats are presently allowed, so that, in this experimental version, they can be compared in terms of convenience and preference. The spirit of PASCAL suggests that optional forms not be given users. But those that prove to be useful, and do not lead to more programming errors, might be worth keeping.

Present options include:

1) Any number of vertical bars can identify a parallel construct, e.g. '|set' '|||||if..|then..|||else' (two bars, e.g. ||set, is recomended, since this is a standard symbol for parallel).

2) '||let' can be used instead of '||set'.

3) '&merge' or 'bmerge' or '&' or 'b' can be used to designate boolean arrays; to designate integer arrays.
The construct [xmin..xmax,ymin..ymax : boolean] can be used to declare a boolean (or .. : integer] for integer) array within the dimensions, as discussed above.

4) Because (when they are implemented) a period must precede a name of an array in an array of records, the options are given to use ',' or '+' or '@ to indicate that ordinary arrays are used.

5) ||dim & [0..7,0..7]; can be used instead of
||dim &from [0..7,0..7]; and, if the desired dimensions have previously been specified and the default condition of integer arrays is desired, then it is sufficient to write:
||dim; (which is equivalent to ||dim {integer} [{current dimensions}];).

6) ||border := {bordertype} is not needed;
border := {bordertype} will suffice.
But the former seems good practice, since border must be designated for the parallel array operations.

7) Similarly, ||end; is not needed: end; will suffice.

8) A sequence of arraynames for assignment or modification (in conditional statements) may be separated by mixtures of commas, end-brackets, or spaces, e.g.:
```
||set name1, name2, name3 :=
||set name1] name2, name3 :=
||set name1, name2 name3 ] name4 :=
```

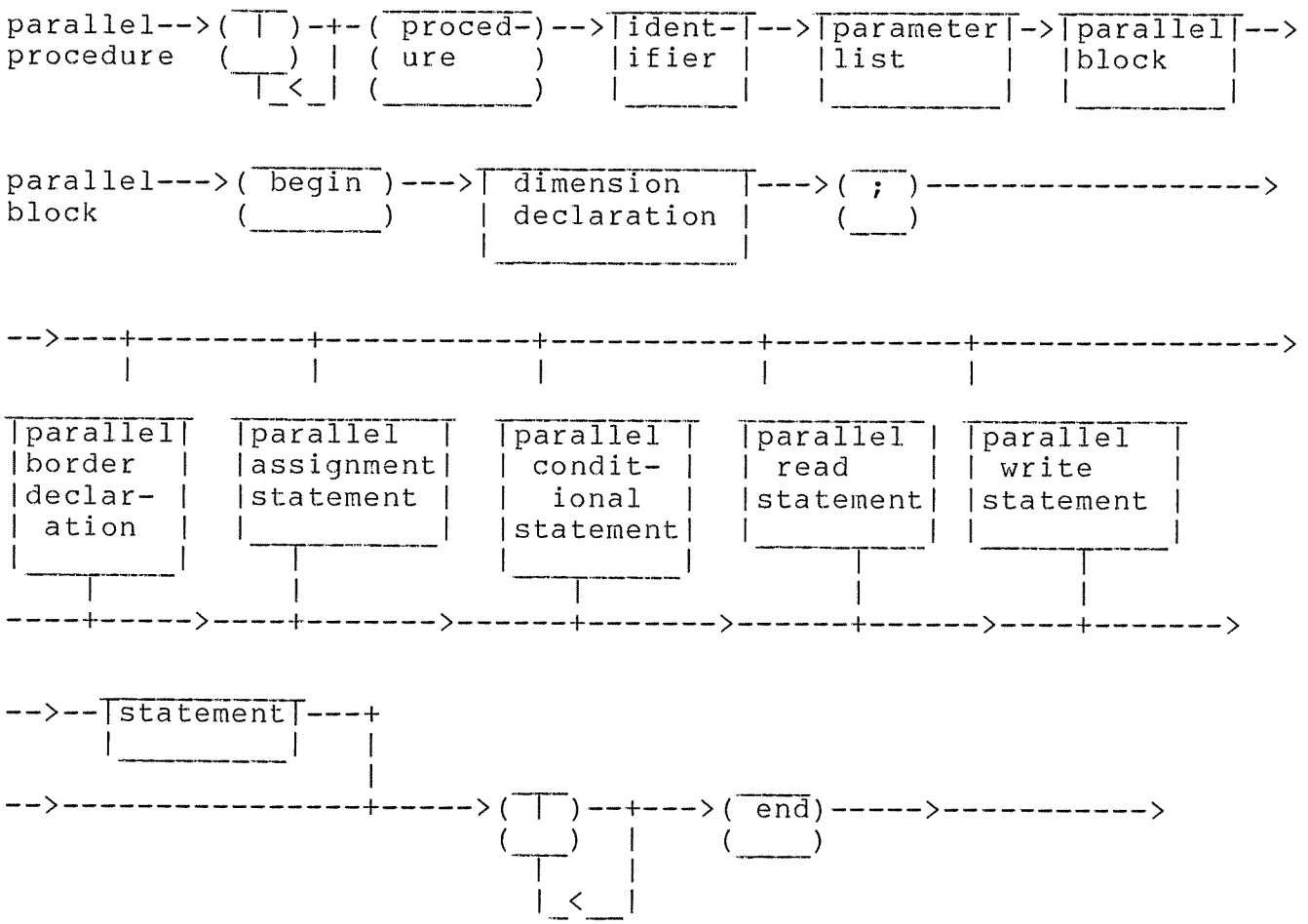
9) If no arithmetic operator follows a name to be modified in a conditional, that array is assigned the value of the expression preceding ||then (and is therefore handled exactly as it would be by a ||set assignment statement).

10) In assignment statements, names can be followed by arithmetic operators and integers. But these will be taken to specify modifications to the assigned value, rather than, as in the conditional statements, as modifications to the presently stored values.

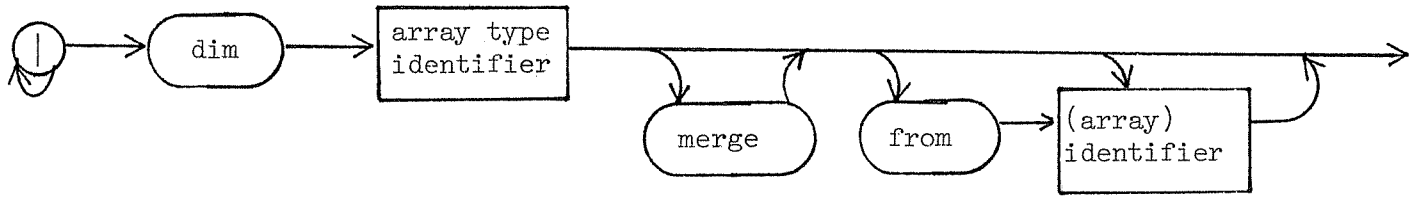
11) The ||read(..); and ||write(..); constructs can specify sub-arrays, e.g., ||read(image[0..3,2..7]);. But this must be done carefully in the present implementation, since these become the new array dimensions, and must lie within the previous dimensions.

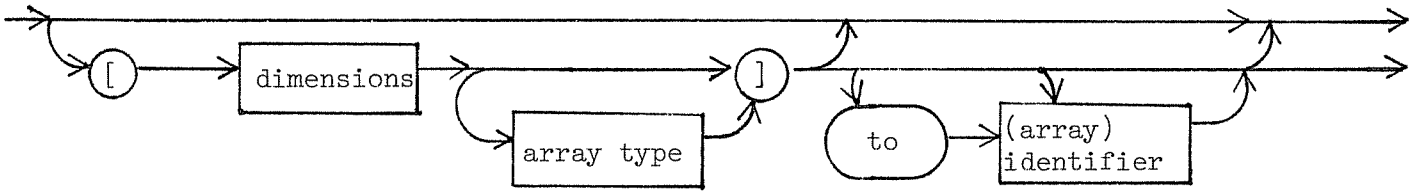
12) ||vari {integer variables} : integer; and
||varb {boolean variables} : boolean;
can be used as an alternative way of declaring array type. (This should be more useful when the program is extended to handle automatically the finding and using of the array's type in determining what type of operations to perform.)

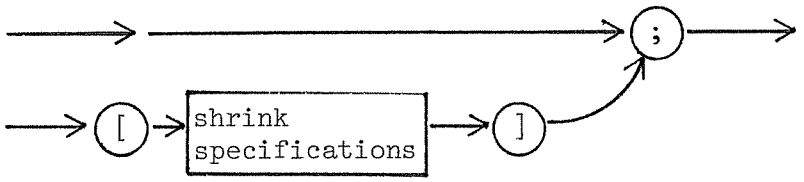
Syntax Diagrams for the PascalPL Extensions to PASCAL

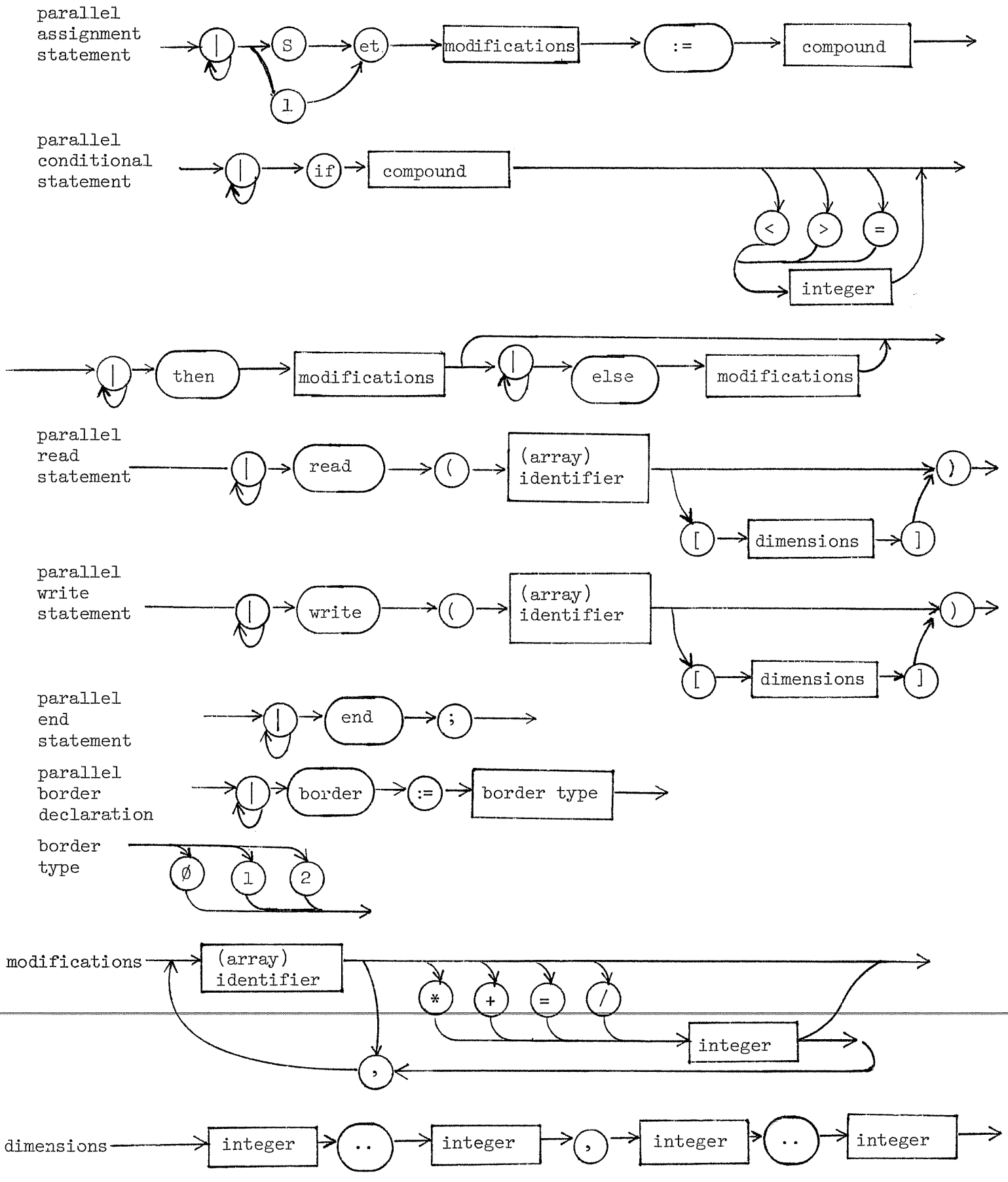


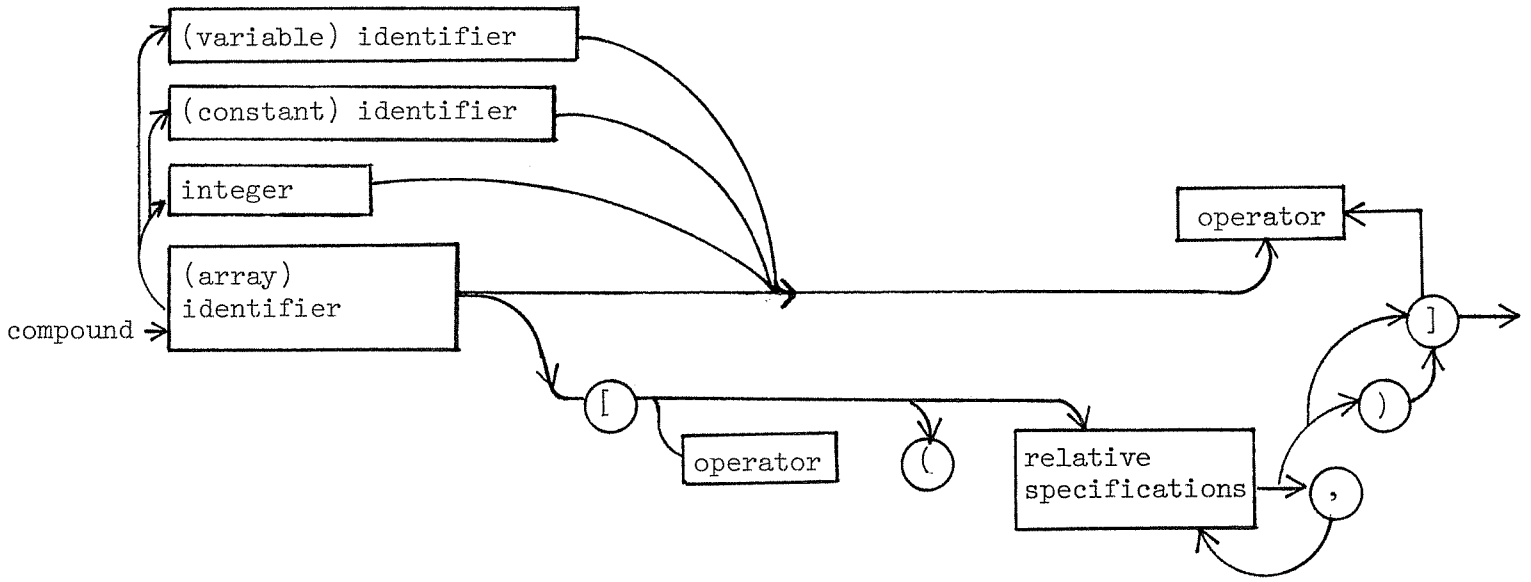
dimension
declaration

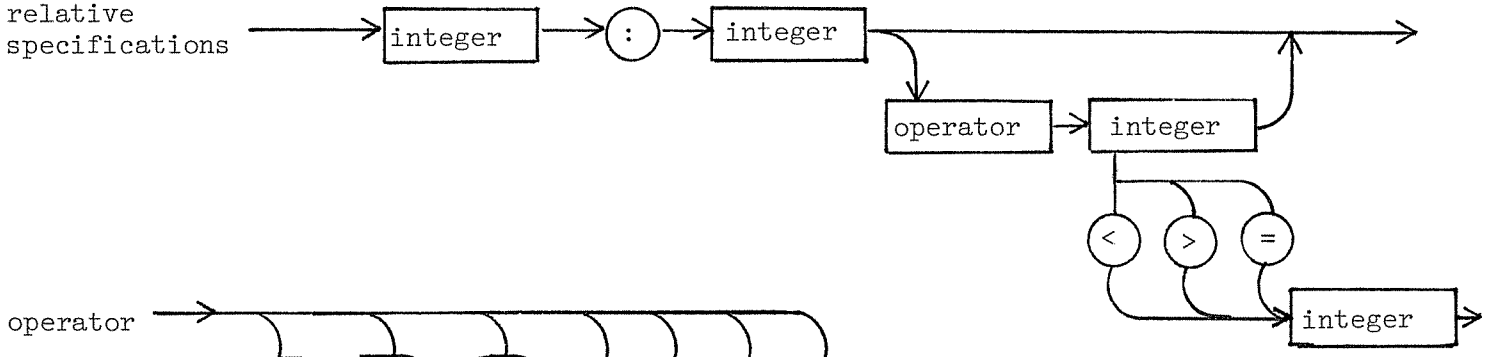


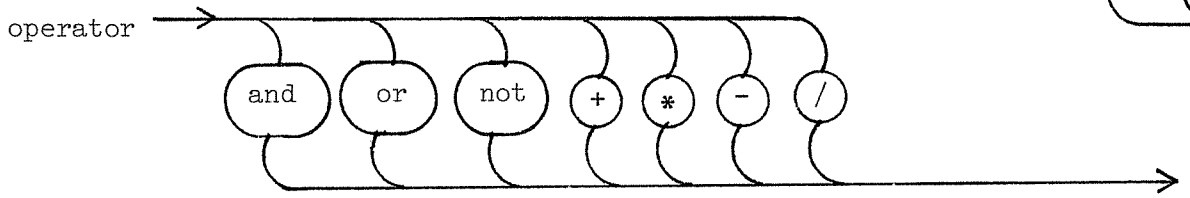


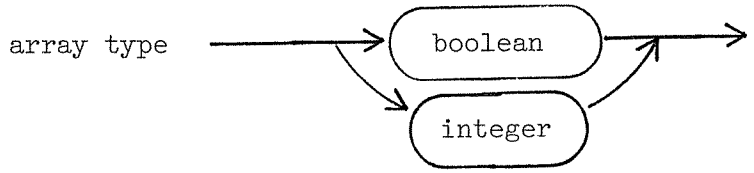


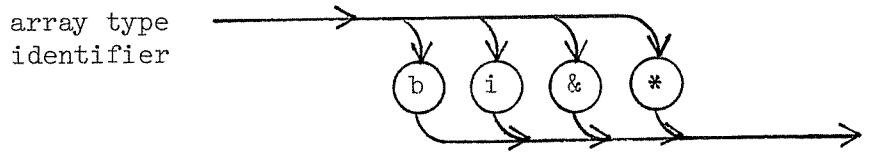

parallel assignment statement
|
S
et
1
modifications
:=
compound
parallel conditional statement
|
if
compound
<
>
=
integer
|
then
modifications
|
else
modifications
parallel read statement
|
read
(
(array) identifier
)
[
dimensions
]
parallel write statement
|
write
(
(array) identifier
)
[
dimensions
]
parallel end statement
|
end
;
parallel border declaration
|
border
:=
border type
border type
ø
1
2
modifications
(array) identifier
*
+
=
/
integer
,
dimensions
integer
..
integer
,
integer
..
integer

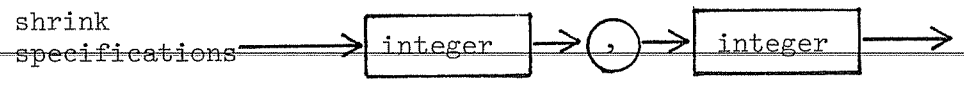

(variable) identifier
(constant) identifier
integer
(array) identifier
compound
operator
[
operator
(
relative specifications
,
)
]

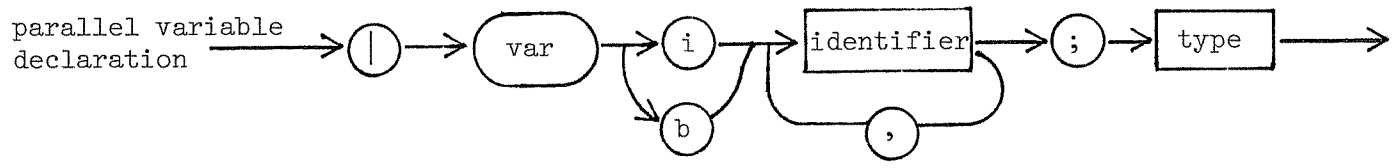

relative specifications
integer
:
integer
operator
integer
<
>
=
integer


operator
and
or
not
+
*
-
/


array type
boolean
integer


array type identifier
b
i
&
*


shrink specifications
integer
,
integer


parallel variable declaration
|
var
i
b
identifier
,
;
type

Examples of Program and Code Translated from PascalPL to PASCAL

The PascalPL Program Input to the PascalPL Preprocessor

```
program simple(input,output);
{the programmer must declare the array data structures used}
const
  xmin = 0; xmax = 7; ymin = 0; ymax = 7;
type
  xindex = xmin..xmax;
  yindex = ymin..ymax;
  arraytype = array [xindex,yindex] of integer;
var
  arrayx : xmin..xmax;
  arrayy : ymin..ymax;
  image, negative, edgedimage : arraytype;

procedure sayhello;
begin
  writeln('hello');
end;

||procedure demonstrate;
begin
||dim    [0..2,0..2];
  ||read(image);
  ||write(image);
    ||read(negative);
  writeln('the image and the negative image have been input');
  ||set edgedimage := image - negative;
  ||write(edgedimage);
  ||set image, negative := 0;
end;

begin {program}
  sayhello;
  demonstrate;
end.
```

The PASCAL Program Output for the Simple Program Above

```
          {****STARTING READIN****}
{****YOU MAY ASSIGN border := 0 (or 1 or 2)****}
{****DEFAULT border = 2 (for self - i.e., the nearest inner border)****}
{****DEFAULT arraydimensions are 0 .. 7, 0..7****}
{****DEFAULT xshrink, yshrink are 1 and 1 (no shrink)****}

(*INPUT1=program simple(input,output); *)
program simple(input,output);

(*INPUT2= {the programmer must declare the array data structures used} *)
 {the programmer must declare the array data structures used}

(*INPUT3= const *)
 const
```

```
(*INPUT4=   xmin = 0; xmax = 7; ymin = 0; ymax = 7; *)
   xmin = 0; xmax = 7; ymin = 0; ymax = 7;

(*INPUT5= type *)
 type

(*INPUT6=   xindex = xmin..xmax; *)
   xindex = xmin..xmax;

(*INPUT7=   yindex = ymin..ymax; *)
   yindex = ymin..ymax;

(*INPUT8=   arraytype = array [xindex,yindex] of integer; *)
   arraytype = array [xindex,yindex] of integer;

(*INPUT9= var *)
 var

(*INPUT10=   arrayx : xmin..xmax; *)
   arrayx : xmin..xmax;

(*INPUT11=   arrayy : ymin..ymax; *)
   arrayy : ymin..ymax;

(*INPUT12=   image, negative, edgedimage : arraytype; *)
   image, negative, edgedimage : arraytype;

(*INPUT13=  *)


(*INPUT14= procedure sayhello; *)
 procedure sayhello;

(*INPUT15= begin *)
 begin

(*INPUT16=   writeln('hello'); *)
   writeln('hello');

(*INPUT17= end; *)
 end;

(*INPUT18=  *)


(*INPUT19= ||procedure demonstrate; *)
 procedure plparp11;
       forward;

                {**          1||PARALLEL BLOCK starts.**}
procedure demonstrate;

(*INPUT20= begin *)
 begin

(*INPUT21= ||dim    [0..2,0..2]; *)
```

```
    plparpll;
{****  DIMENSIONS ARE=          0          2          0          2****}
end;
                    {***starting a block of PARALLEL CODE!!**}
procedure plparpll;
const
  xmin =  0;
  xmax =  2;
  ymin =  0;
  ymax =  2;
type
  xindex = xmin..xmax;
  yindex = ymin..ymax;
    temparraytype = array [xindex,yindex] of integer;
var
  beyondborder : boolean;
  partineq : char;
  valueread, border, tempstore, partthresh, arrayxloc,arrayyloc,xshrink,yshr
  arrayx : xmin..xmax;
  arrayy : ymin..ymax;
  TEMPARRAY0, TEMPARRAY1, TEMPARRAY2 : temparraytype;
begin
  border := 2;
{****SET NO SHRINK, so SHRINK = 1,1****}
  xshrink := 1;
  yshrink := 1;

(*INPUT22=   ||read(image); *)
   writeln("TYPE IN INTEGERS FOR  image[ 0.. 2, 0.. 2]");
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
             read(valueread);
             image[arrayx,arrayy] := valueread;
             writeln("[ ",arrayx," , ",arrayy,"] :=",valueread);
          end;
      end;

(*INPUT23=   ||write(image); *)
   writeln("ARRAY =  image  contains:");
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
            write(image[arrayx,arrayy]:4);
          end;
        writeln(" ;");
      end;
    writeln(" END OF ARRAY.");

(*INPUT24=     ||read(negative); *)
     writeln("TYPE IN INTEGERS FOR  negative[ 0.. 2, 0.. 2]");
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
```

```
          begin
             read(valueread);
             negative[arrayx,arrayy] := valueread;
             writeln("[ ",arrayx," , ",arrayy,"] :=",valueread);
          end;
       end;

(*INPUT25=   writeln('the image and the negative image have been input'); *)
   writeln('the image and the negative image have been input');

(*INPUT26=   ||set edgedimage := image - negative; *)
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
              {(*NO REL LOCS, SO NO BORDER TEST NEEDED*)}
            arrayxloc := arrayx;
            arrayyloc := arrayy;
              if beyondborder = true  then
                 TEMPARRAY1[arrayx,arrayy] :=   0
               else
                 begin
                   tempstore :=  image[arrayxloc,arrayyloc];
                   TEMPARRAY1[arrayx,arrayy] :=  tempstore;
                 end;
             TEMPARRAY0[arrayx,arrayy] :=
                    TEMPARRAY1[arrayx,arrayy];
              {(*NO REL LOCS, SO NO BORDER TEST NEEDED*)}
            arrayxloc := arrayx;
            arrayyloc := arrayy;
              if beyondborder = true  then
                 TEMPARRAY1[arrayx,arrayy] :=   0
               else
                 begin
                   tempstore :=  negative[arrayxloc,arrayyloc];
                   TEMPARRAY1[arrayx,arrayy] :=  tempstore;
                 end;
           TEMPARRAY0[arrayx,arrayy] :=
                  TEMPARRAY0[arrayx,arrayy]
                     -  TEMPARRAY1[arrayx,arrayy];
          end;
      end;
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
            edgedimage[arrayx div xshrink,arrayy div yshrink] :=
              TEMPARRAY0[arrayx,arrayy];
          end;
      end;

(*INPUT27=   ||write(edgedimage); *)
   writeln("ARRAY =  edgedimage  contains:");
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
```

```
          begin
            write(edgedimage[arrayx,arrayy]:4);
          end;
        writeln(" ;");
      end;
    writeln(" END OF ARRAY.");

(*INPUT28=   ||set image, negative := 0; *)
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
               TEMPARRAY0[arrayx,arrayy] :=   0;
          end;
      end;
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
            image[arrayx div xshrink,arrayy div yshrink] :=
              TEMPARRAY0[arrayx,arrayy];
            negative[arrayx div xshrink,arrayy div yshrink] :=
              TEMPARRAY0[arrayx,arrayy];
          end;
      end;

(*INPUT29= ||end; *)
 end; {END OF PARALLEL BLOCK=  1}

(*INPUT30=  *)


(*INPUT31= begin {program} *)
 begin {program}

(*INPUT32=   sayhello; *)
   sayhello;

(*INPUT33=   demonstrate; *)
   demonstrate;

(*INPUT34= end. *)
 end.

(*INPUT35=  *)


(*INPUT36=  *)
```

```
{****READIN HAS FINISHED.****}
```

Examples of ||Set Statements Translated from PascalPL to PASCAL

PASCAL code follows for the following two PascalPL statements:

```
||set cross := vert[+(0:0,0:2,0:-2)] * hor[+(0:0,2:0,-2:0)];
||set featurei, featurej, labelk
    := featurem[*(4:-3*2>5,-5:7*3>14,0:0*21>112)];

(*INPUTx=   ||set cross := vert[+(0:0,0:2,0:-2)] * hor[+(0:0,2:0,-2:0)]; *)
{****ASSIGNMENT STATEMENT HANDLED HERE.****}
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
            arrayxloc := arrayx +  0;
            arrayyloc := arrayy +  0;
                          {(**KEEPS IT IN BOUNDS!**)}
              beyondborder := false;
                if arrayxloc <   0  then beyondborder := true;
                if arrayxloc >   2  then beyondborder := true;
                if arrayyloc <   0  then beyondborder := true;
                if arrayyloc >   2  then beyondborder := true;
              if beyondborder = true  then
                 TEMPARRAY1[arrayx,arrayy] :=   0
               else
                 begin
                   tempstore :=  vert[arrayxloc,arrayyloc] *  1;
                   TEMPARRAY1[arrayx,arrayy] :=  tempstore;
                 end;
            arrayxloc := arrayx +  0;
            arrayyloc := arrayy +  2;
                          {(**KEEPS IT IN BOUNDS!**)}
              beyondborder := false;
                if arrayxloc <   0  then beyondborder := true;
                if arrayxloc >   2  then beyondborder := true;
                if arrayyloc <   0  then beyondborder := true;
                if arrayyloc >   2  then beyondborder := true;
              if beyondborder = true  then
                TEMPARRAY2[arrayx,arrayy] :=
                     0   +  TEMPARRAY1[arrayx,arrayy]
               else
                 begin
                   tempstore :=  vert[arrayxloc,arrayyloc] *  1;
                       TEMPARRAY2[arrayx,arrayy] :=
                              TEMPARRAY1[arrayx,arrayy]
                                 +   tempstore;
                   if border < 2  then
                     if TEMPARRAY2[arrayx,arrayy]  > 1  then
                       TEMPARRAY2[arrayx,arrayy] := 0;
                 end;
            arrayxloc := arrayx +  0;
            arrayyloc := arrayy + -2;
                          {(**KEEPS IT IN BOUNDS!**)}
              beyondborder := false;
```

```
    if arrayxloc <   0  then beyondborder := true;
    if arrayxloc >   2  then beyondborder := true;
    if arrayyloc <   0  then beyondborder := true;
    if arrayyloc >   2  then beyondborder := true;
  if beyondborder = true  then
    TEMPARRAY1[arrayx,arrayy] :=
         0   +  TEMPARRAY2[arrayx,arrayy]
   else
     begin
       tempstore :=  vert[arrayxloc,arrayyloc] *  1;
           TEMPARRAY1[arrayx,arrayy] :=
                  TEMPARRAY2[arrayx,arrayy]
                     +   tempstore;
       if border < 2  then
         if TEMPARRAY1[arrayx,arrayy]  > 1  then
           TEMPARRAY1[arrayx,arrayy] := 0;
    end;
 TEMPARRAY0[arrayx,arrayy] :=
        TEMPARRAY1[arrayx,arrayy];
arrayxloc := arrayx +  0;
arrayyloc := arrayy +  0;
              {(**KEEPS IT IN BOUNDS!**)}
  beyondborder := false;
    if arrayxloc <   0  then beyondborder := true;
    if arrayxloc >   2  then beyondborder := true;
    if arrayyloc <   0  then beyondborder := true;
    if arrayyloc >   2  then beyondborder := true;
  if beyondborder = true  then
     TEMPARRAY1[arrayx,arrayy] :=   0
   else
     begin
       tempstore :=  hor[arrayxloc,arrayyloc] *  1;
       TEMPARRAY1[arrayx,arrayy] :=  tempstore;
     end;
arrayxloc := arrayx +  2;
arrayyloc := arrayy +  0;
              {(**KEEPS IT IN BOUNDS!**)}
  beyondborder := false;
    if arrayxloc <   0  then beyondborder := true;
    if arrayxloc >   2  then beyondborder := true;
    if arrayyloc <   0  then beyondborder := true;
    if arrayyloc >   2  then beyondborder := true;
  if beyondborder = true  then
    TEMPARRAY2[arrayx,arrayy] :=
         0   +  TEMPARRAY1[arrayx,arrayy]
   else
     begin
       tempstore :=  hor[arrayxloc,arrayyloc] *  1;
           TEMPARRAY2[arrayx,arrayy] :=
                  TEMPARRAY1[arrayx,arrayy]
                     +   tempstore;
       if border < 2  then
         if TEMPARRAY2[arrayx,arrayy]  > 1  then
           TEMPARRAY2[arrayx,arrayy] := 0;
    end;
arrayxloc := arrayx + -2;
```

```
            arrayyloc := arrayy +  0;
                          {(**KEEPS IT IN BOUNDS!**)}
              beyondborder := false;
                if arrayxloc <   0  then beyondborder := true;
                if arrayxloc >   2  then beyondborder := true;
                if arrayyloc <   0  then beyondborder := true;
                if arrayyloc >   2  then beyondborder := true;
              if beyondborder = true  then
                TEMPARRAY1[arrayx,arrayy] :=
                     0   +  TEMPARRAY2[arrayx,arrayy]
               else
                 begin
                   tempstore :=  hor[arrayxloc,arrayyloc] *  1;
                       TEMPARRAY1[arrayx,arrayy] :=
                              TEMPARRAY2[arrayx,arrayy]
                                 +   tempstore;
                   if border < 2  then
                     if TEMPARRAY1[arrayx,arrayy]  > 1  then
                       TEMPARRAY1[arrayx,arrayy] := 0;
                end;
           TEMPARRAY0[arrayx,arrayy] :=
                  TEMPARRAY0[arrayx,arrayy]
                    *  TEMPARRAY1[arrayx,arrayy];
          end;
      end;
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
            cross[arrayx div xshrink,arrayy div yshrink] :=
              TEMPARRAY0[arrayx,arrayy];
          end;
      end;

(*INPUTy1=   ||set featurei, featurej, labelk *)
{****ASSIGNMENT STATEMENT HANDLED HERE.****}

(*INPUTy2=      := featurem[*(4:-3*2>5,-5:7*3>14,0:0*21>112)]; *)
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
            arrayxloc := arrayx +  4;
            arrayyloc := arrayy + -3;
                          {(**KEEPS IT IN BOUNDS!**)}
              beyondborder := false;
                if arrayxloc <   0  then beyondborder := true;
                if arrayxloc >   2  then beyondborder := true;
                if arrayyloc <   0  then beyondborder := true;
                if arrayyloc >   2  then beyondborder := true;
             if beyondborder = true  then
                TEMPARRAY1[arrayx,arrayy] :=   0
              else
                begin
                  tempstore :=  featurem[arrayxloc,arrayyloc] *  2;
                   if not (  tempstore  >    5 )  then
```

```
                tempstore := 0;
              TEMPARRAY1[arrayx,arrayy] :=  tempstore;
            end;
          arrayxloc := arrayx + -5;
          arrayyloc := arrayy +  7;
                    {(**KEEPS IT IN BOUNDS!**)}
            beyondborder := false;
              if arrayxloc <   0  then beyondborder := true;
              if arrayxloc >   2  then beyondborder := true;
              if arrayyloc <   0  then beyondborder := true;
              if arrayyloc >   2  then beyondborder := true;
            if beyondborder = true  then
              TEMPARRAY2[arrayx,arrayy] :=
                   0   *  TEMPARRAY1[arrayx,arrayy]
             else
               begin
                 tempstore :=  featurem[arrayxloc,arrayyloc] *  3;
                  if not (  tempstore  >   14 )  then
                     tempstore := 0;
                     TEMPARRAY2[arrayx,arrayy] :=
                            TEMPARRAY1[arrayx,arrayy]
                               *   tempstore;
                 if border < 2  then
                   if TEMPARRAY2[arrayx,arrayy]  > 1  then
                     TEMPARRAY2[arrayx,arrayy] := 0;
               end;
          arrayxloc := arrayx +  0;
          arrayyloc := arrayy +  0;
                    {(**KEEPS IT IN BOUNDS!**)}
            beyondborder := false;
              if arrayxloc <   0  then beyondborder := true;
              if arrayxloc >   2  then beyondborder := true;
              if arrayyloc <   0  then beyondborder := true;
              if arrayyloc >   2  then beyondborder := true;
            if beyondborder = true  then
              TEMPARRAY1[arrayx,arrayy] :=
                   0   *  TEMPARRAY2[arrayx,arrayy]
             else
               begin
                 tempstore :=  featurem[arrayxloc,arrayyloc] * 21;
                  if not (  tempstore  >  112 )  then
                     tempstore := 0;
                     TEMPARRAY1[arrayx,arrayy] :=
                            TEMPARRAY2[arrayx,arrayy]
                               *   tempstore;
                 if border < 2  then
                   if TEMPARRAY1[arrayx,arrayy]  > 1  then
                     TEMPARRAY1[arrayx,arrayy] := 0;
               end;
          TEMPARRAY0[arrayx,arrayy] :=
                TEMPARRAY1[arrayx,arrayy];
      end;
  end;
for arrayx :=   0 to   2 do
  begin
    for arrayy :=   0 to   2 do
```

```
        begin
          featurei[arrayx div xshrink,arrayy div yshrink] :=
            TEMPARRAY0[arrayx,arrayy];
          featurej[arrayx div xshrink,arrayy div yshrink] :=
            TEMPARRAY0[arrayx,arrayy];
          labelk[arrayx div xshrink,arrayy div yshrink] :=
            TEMPARRAY0[arrayx,arrayy];
        end;
    end;
```

An Example of PascalPL Output for a Simple Conditional Statement

The following shows the PascalPL conditional statement (labeled INPUTz), and then the PASCAL code output by PascalPL, for:

```
||if featurei[+(0:1,0:-1)] * featurej[+(1:0,-1:0)]  >  11
  ||then  labeli+19, labelj*2, labelk-33
    ||else  labell*2, labelm+27;
```

```
(*INPUTz1=   ||if featurei[+(0:1,0:-1)] * featurej[+(1:0,-1:0)]  >  11  *)
      {IF STATEMENT BEING HANDLED HERE.}
  for arrayx :=   0 to   2 do
    begin
      for arrayy :=   0 to   2 do
        begin
          arrayxloc := arrayx +  0;
          arrayyloc := arrayy +  1;
                       {(**KEEPS IT IN BOUNDS!**)}
            beyondborder := false;
              if arrayxloc <   0  then beyondborder := true;
              if arrayxloc >   2  then beyondborder := true;
              if arrayyloc <   0  then beyondborder := true;
              if arrayyloc >   2  then beyondborder := true;
            if beyondborder = true  then
               TEMPARRAY1[arrayx,arrayy] :=   0
             else
               begin
                 tempstore :=  featurei[arrayxloc,arrayyloc] *  1;
                 TEMPARRAY1[arrayx,arrayy] :=  tempstore;
               end;
          arrayxloc := arrayx +  0;
          arrayyloc := arrayy + -1;
                       {(**KEEPS IT IN BOUNDS!**)}
            beyondborder := false;
              if arrayxloc <   0  then beyondborder := true;
              if arrayxloc >   2  then beyondborder := true;
              if arrayyloc <   0  then beyondborder := true;
              if arrayyloc >   2  then beyondborder := true;
            if beyondborder = true  then
              TEMPARRAY2[arrayx,arrayy] :=
                   0   +  TEMPARRAY1[arrayx,arrayy]
             else
               begin
                 tempstore :=  featurei[arrayxloc,arrayyloc] *  1;
```

```
                    TEMPARRAY2[arrayx,arrayy] :=
                          TEMPARRAY1[arrayx,arrayy]
                             +   tempstore;
                  if border < 2  then
                    if TEMPARRAY2[arrayx,arrayy]  > 1  then
                      TEMPARRAY2[arrayx,arrayy] := 0;
                end;
               TEMPARRAY0[arrayx,arrayy] :=
                     TEMPARRAY2[arrayx,arrayy];
              arrayxloc := arrayx +  1;
              arrayyloc := arrayy +  0;
                           {(**KEEPS IT IN BOUNDS!**)}
                beyondborder := false;
                  if arrayxloc <   0  then beyondborder := true;
                  if arrayxloc >   2  then beyondborder := true;
                  if arrayyloc <   0  then beyondborder := true;
                  if arrayyloc >   2  then beyondborder := true;
                if beyondborder = true  then
                   TEMPARRAY1[arrayx,arrayy] :=   0
                 else
                   begin
                     tempstore :=  featurej[arrayxloc,arrayyloc] *  1;
                     TEMPARRAY1[arrayx,arrayy] :=  tempstore;
                   end;
              arrayxloc := arrayx + -1;
              arrayyloc := arrayy +  0;
                           {(**KEEPS IT IN BOUNDS!**)}
                beyondborder := false;
                  if arrayxloc <   0  then beyondborder := true;
                  if arrayxloc >   2  then beyondborder := true;
                  if arrayyloc <   0  then beyondborder := true;
                  if arrayyloc >   2  then beyondborder := true;
                if beyondborder = true  then
                  TEMPARRAY2[arrayx,arrayy] :=
                       0   +  TEMPARRAY1[arrayx,arrayy]
                 else
                   begin
                     tempstore :=  featurej[arrayxloc,arrayyloc] *  1;
                         TEMPARRAY2[arrayx,arrayy] :=
                               TEMPARRAY1[arrayx,arrayy]
                                  +   tempstore;
                     if border < 2  then
                       if TEMPARRAY2[arrayx,arrayy]  > 1  then
                         TEMPARRAY2[arrayx,arrayy] := 0;
                   end;
             TEMPARRAY0[arrayx,arrayy] :=
                    TEMPARRAY0[arrayx,arrayy]
                      *  TEMPARRAY2[arrayx,arrayy];
          end;
      end;

(*INPUTz2=          ||then  labeli+19, labelj*2, labelk-33  *)
    for arrayx :=   0 to   2 do
      begin
        for arrayy :=   0 to   2 do
          begin
```

```
                (**modify if found greaterthan value**)
          if  ( TEMPARRAY0[arrayx,arrayy] > ll )  then
              labeli[arrayx div xshrink,arrayy div yshrink] :=
                labeli[arrayx div xshrink,arrayy div yshrink]  +  19;
                (**modify if found greaterthan value**)
          if  ( TEMPARRAY0[arrayx,arrayy] > ll )  then
              labelj[arrayx div xshrink,arrayy div yshrink] :=
                labelj[arrayx div xshrink,arrayy div yshrink]  *   2;

(*INPUTz3=          ||else  labell*2, labelm+27; *)
                (**modify if found greaterthan value**)
          if  ( TEMPARRAY0[arrayx,arrayy] > ll )  then
              labelk[arrayx div xshrink,arrayy div yshrink] :=
                labelk[arrayx div xshrink,arrayy div yshrink]  -  33;

          else
                          (**else  do the following:**)
              labell[arrayx div xshrink,arrayy div yshrink] :=
                labell[arrayx div xshrink,arrayy div yshrink]  *   2;
                          (**else  do the following:**)
              labelm[arrayx div xshrink,arrayy div yshrink] :=
                labelm[arrayx div xshrink,arrayy div yshrink]  +  27;
        end;
    end;
```

References

anon, advertising brochure and personal communication, Goodyear-Aerospace, Akron, Ohio, 1979.

Batcher, K.E., STARAN parallel processor system hardware, Proc. AFIPS National Computer Conf., 1974, 43, 405-410.

Brinch Hansen, P., Operating System Principles, Englewood-Cliffs: Prentice-Hall, 1973.

Douglass, R.J., Extensions to PASCAL for parallel image processing, paper presented at Workshop on Higher-Level Languages for Image Processing, Windsor, England, June, 1979.

Duff, M. J. B., CLIP4: a large scale integrated circuit array parallel processor, Proc. IJCPR-3, 1976, 4, 728-733.

Duff, M. J. B., Review of the CLIP image processing system, Proc. National Computer Conf., 1978, pp. 1055-1060.

Duff, M.J.B., Final Report on Workshop on Higher-Level Languages for Image Processing, University College London, 1979.

Flanders, P.M., Hunt, D.J., Reddaway, S.F., Parkinson, D., Efficient high speed computing with the Distributed Array Processor, In: High Speed Computer and Algorithm Organization, New York: Academic Press, 1977, pp. 113-128.

Fung, L., A massively parallel processing computer. In: High Speed Computer and Algorithm Organization, Kuck, Lawrie and Someh, Eds., New York: Academic Press, 1977.

Gudmundsson, B., An interactive high-level language system for picture processing, Paper presented at Conference on Higher-Level Languages for Image Processing, Windsor, England, June, 1979.

Ison, Unpublished paper on extensions to PASCAL for parallel networks, Univ. of Virginia, 1977.

Iverson, K.E., A Programming Language, New York: Wiley, 1962.

Jensen, K. and Wirth, N., PASCAL User Manual and Report (Second Edition), Berlin: Springer-Verlag, 1976.

Kruse, B. The PICAP picture processing laboratory, Proc. IJCPR-3, 1976, 4, 875-881.

Kruse, B. Experience with a picture processor in pattern recognition processing, Proc. National Computer Conf., 1978.

Lawrie, D.H., Layman, T., Baer, D. and Randal, J.M., GLYPNIR - a programming language for ILLIAC IV, CACM, 1975, 18, 157-164.

Levialdi, S., Maggiolo- Schettini, A., Napoli, M, and Uccella, G., PIXAL: a high level language for image processing, working paper, Japan-U.S. Seminar on Real-Time Parallel Image Analysis and Recognition, 1978.

Perrott, R. and Stevenson, D., ACTUS - a language for SIMD architectures, Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors, Los Alamos, 1978, pp. 212-218.

Preston, K., Image manipulative languages: a preliminary survey, paper presented at Workshop on Higher-Level Languages for Image Processing, Windsor, England, June, 1979.

Reddaway, S.F., DAP - a flexible number cruncher, Proceedings of the 1978 LASL Workshop on Vector and Parallel Processors, Los Alamos, 1978, pp. 233-234.

Reeves, A.P., An array processing system with a Fortran-based realization, Computer Graphics and Image Processing, 1979, 9, 267-281.

Schmitt, L., Unpublished paper on parallel languages for general networks, Univ. of Wisconsin, 1979.

Sternberg, S.R., Cytocomputer real-time pattern recognition, paper presented at Eight Pattern Recognition Symposium, National Bureau of Standards, April, 1978.

Travis, L., Honda, M., LeBlanc, R. and Zeigler, S., Design rationale for TELOS, a PASCAL based AI language, ACM SIGPLAN, 1977, 12, no. 8, 67-76.

Uhr, L., A language for programming scene description and pattern recognition systems on a parallel array computer, paper presented at Workshop on Higher-Level Languages for Image Processing, Windsor, England, June, 1979. (also U. Wisconsin Comp. Sci. Dept. Tech Rept. 354)

Wirth, N., Design of a PASCAL compiler, Software: Practice and Experience, 1971, 1, 309-333.

Wirth, N., Toward a discipline of real-time programming, Comm. ACM, 1977, 20, 577-583.

Wood, A., CAP4 Programmers Manual, University College London, Image Processing Group, 1977.

$