DATA TYPES FOR VERY HIGH
LEVEL PROGRAMMING LANGUAGES+

Narain Gehani

TR 75-258

August 1975

Department of Computer Science
Cornell University
Ithaca, New York 14853

+This research was supported in part by the National Science Foundation under grant GJ-42512.

DATA TYPES FOR VERY HIGH

LEVEL PROGRAMMING LANGUAGES

Narain Gehani Ph.D.

Cornell University, 1975

ABSTRACT

Very high level programming languages (higher than PL/I, Algol 60 etc) attempt to free the programmer from providing details and let him concentrate on the algorithm for the problem at hand. The importance of very high level programming languages is further emphasized by decreasing machine costs, increased programming costs and the desire to have programs that are well structured, easy to understand and prove correct. Very high level languages provide powerful control structures and data structures that allow the problem to be specified in a natural manner.

In this dissertation we propose several ways of raising the level of a language. The different types of for iteration statements are consolidated into one general for statement. This along with a new type, the domain of an array, provides us with an easy way of processing arrays; nested iteration statements are no longer necessary. The syntactic list and array generators and the concept of overloading make programming more flexible.

The current notion that a data type is a set of values together with basic operations on that set leads us to conclude that formal parameter types need not be explicitly stated. Given a formal parameter X with operations $z_1, z_2, \ldots, z_n$ being performed on it within the procedure, one should be able to supply, as an actual parameter in a call, a variable of any type that has the operations $z_1, z_2, \ldots, z_n$ defined on it. For example, this concept allows us to write <u>one</u> procedure that finds the maximum value of the elements of an array of <u>any</u> dimension and <u>any</u> element or index type.

Grids are arrays that can have any shape. Grid elements need not be contiguous i.e. grids can have holes in them. For example, grids can be trapezoidal, parabolic, rectangular with a hole or pyramid-like. Programs written using grids are more general than those written using arrays and/or functions to simulate non-array shapes. To alter an existing program to work for another grid shape one need only modify the grid declaration suitably, leaving the rest of the program intact. Programs are smaller, semantically clearer and have a more natural problem representation. Grids may be used to represent sparse matrices. Data security is achieved by allowing parts of grids pass as parameters to be read only or completely masked out. <u>Grids have been implemented as an extension to Fortran</u>.

Using Pascal as the base language, we show by series of examples from numerical analysis, data processing, engineering etc. how the above concepts raise the level of a programming language and how they blend together naturally and systematically. Efficient ways of implementing them are also discussed.

BIOGRAPHICAL SKETCH

Narain Gehani was born at New Delhi, India on [redacted] 2, 1947. On graduating from St. Columba's High School, New Delhi, in 1964, he won the National Science Talent Scholarship and was admitted to the Indian Institute of Technology, Delhi, on the basis of All-India competitive examinations. Graduating from the Indian Institute of Technology with a Bachelor of Technology (Mechanical Engineering) he proceeded to the U.S.A. where he joined the Steven Institute of Technology, Hoboken, N.J. In the fall of 1971, after finishing M. Eng. (Mechanical Engineering) at Stevens, he was admitted for the Ph.D. program in Computer Science at Cornell University. He was awarded M.S. in Computer Science in June 1975.

to my parents and grandparents

"The thing can be done," said the butcher. "I think.
The thing must be done, I am sure
The thing shall be done! Bring me paper and ink,
The best there is time to procure."

--- Lewis Carroll in The
Hunting of the Snark.

ACKNOWLEDGEMENTS

I must thank Dr. David Gries, my thesis advisor, for his guidance, patience, constant encouragement and friendship. Mere words, however, cannot fully express my profound graditude.

Professors J. Moré, K. Torrance and F. Gouldin served on my Special Committee and I thank them for being helpful in more ways than one.

My colleagues Ricardo Duhne, Charlie Fischer, Janos Simon and Susan Owicki are, in one way or another, responsible for my being able to complete this thesis.

Finally, I would like to thank Debra Fields without whose help this thesis could not have appeared in this form.

TABLE OF CONTENTS

Chapter I INTRODUCTION

VHL programming languages are those that are 'higher' than the current high level languages Fortran IV, Algol 60, PL/I, Pascal, etc. The ultimate in high level programming will have arrived when we are able to build a language processor that accepts the problem stated in a natural language like English and solves it (the user would not have to supply the algorithm to solve the problem). Whether we will ever be able to build such a processor is a debatable question. In any case, the arrival of programming in English is far from imminent. In this dissertation we propose some new data types and some new concepts related to data types which, if incorporated into current high level languages, will raise their level (when talking about high level languages, we mean general purpose languages and not specialized or problem oriented languages).

The higher the level of programming language, the easier it should be to program in it. Programs should be semantically clearer, easier to read, understand and prove correct. The desire to have very high level languages reflects Perlis's [9] opinion when, in his Turing lecture, he says "Programmers should never be satisfied with languages which permit them to program everything but to program nothing of interest easily".

The level of a programming language may be raised by the introduction of more powerful data structures and their access functions, control statements and operations. The language designer must be careful to avoid the pitfall of making the proposed VHL language an agglomeration of 'neat' features or many problem oriented or specialized languages. The language should be systematically designed, simple, syntactically and semantically well defined, without glitches, and conducive to top down structured program description.

The data structures of a programming language should provide for a natural representation of the data of the problem in question. Otherwise, as Standish [12] says, "Indirectness of data representation and manipulation may cause a loss of both clarity of structure and also of efficiency since operations on the representations which could be expressed or performed once prior to writing an algorithm may be distributed through it and since the primitive data structures used to represent the data of interest may not be well adapted to the task".

We now give a brief summary of the contents of the ensuing chapters in which we propose different ways of raising the level of a language.

In the first part of chapter II we reintroduce the concept of overloading and show that it is an elegant way of defining operations for the elements of a user defined type. We also discuss implicit conversion; type t_1 should be implicitly converted to type t_2 only if t_1 is a 'subtype' of t_2. A notation for specifying this kind of implicit conversion is described.

In the second part of chapter II the different types of for statements are consolidated into one general for statement which along with a new type, the domain of an array, considerably simplifies iterating over arrays. Now one does not need to write nested for statements to 'visit' the elements of an array with more than one dimension. We next discuss the array and syntactic list generators and show how they make programming flexible.

We put forth the idea of allowing parameter types to be parameters themselves in chapter III. This allows us to write procedures† that accept different types of actual different types of actual parameters. For example, we can write one procedure to multiply arrays of any element type (real integer, boolean, complex, etc.) provided the operators + (addition), * (multiplication) and := (assignment) are defined for that type. We also discuss why this is good in programming, how it helps the programmer, etc. The structured type grid is introduced in chapter IV. Grids are arrays that can have any shape and whose elements need not be contiguous. Portions of grids may be named and used

† When talking about procedures we also imply functions and operators.

in aggregate operations. The indices of elements of a grid or portions of it may be generated by using the general <u>for</u> statement discussed in chapter II. Grids are needed in a lot applications since they enable us to easily represent real world concepts which are often not array-like (e.g. the cross-section of a square pipe).

Using Pascal as the base language, we show by series of examples how the above concepts make programming higher level and how they blend together naturally and systematically. We also discuss efficient ways of implementing procedures that accept different types of parameters and grids.

Here we interject a word of caution. We do not propose the immediate incorporation of all these concepts into a high level language as their <u>full</u> impact on <u>all</u> parts of the language has not been considered. These are ideas for future high level languages and further analysis might result in their being changed or restricted somewhat.

CHAPTER II. GOOD EXTENSIONS AND CHANGES TO PASCAL

The programming language language Pascal, because of its simplicity and clarity of definition, both formal Hoare [5] and informal Jensen [4] and Wirth [14] is chosen as the vehicle to explain the concepts proposed by us.

We will rely on the reader's experience with programming languages to accept certain changes to Pascal without a formal description, such as the use of the break chapter '_' in identifiers, the assumption of Algol like block structure or that a function may return a value of any type (except <u>file</u>).

When specifying the type of a formal parameter, we allow the use of the actual type (e.g. <u>array</u>[1,50] <u>of</u> real, <u>set</u> <u>of</u> integer) instead of just the type identifier. For an array formal parameter, the character '*' may be substituted for the bounds of an array as in PL/I. The bounds of a dimension with a '*' are received from the procedure call. This allows procedures with formal parameters of type array to be called at different places with arrays having different bounds.

We need these and other changes described later to convey our new ideas in a systematic fashion.

1. Defining Operations on Data Types.

Historically, data types were generally considered to be sets of values with no attention being focussed on what operations can be performed on these values. The programmer might know that it does not make any sense to multiply two variables of type social_security_number (defined to be a subrange of the integers) but most languages do not allow him to specify this. Computer scientists Morris [7], Organick [8] and others now view a data type as

i. set of values, together with
ii. operations that can be performed on these values.

A language that allows the definition of new types should provide the user with a mechanism for specifying the operations to be performed on their elements. For example, if we define a new data type complex, we should be able to define the operations of assignment :=, equality =, addition +, multiplication *, etc. for complex values. The above is in complete agreement with Hoare [6] who says that 'In order to fully embed a type language, a facility must be provided for defining the normal arithmetic operations for quantities of this new type; so that subsequently the arithmetic operations +, -, *, etc may be used to operate on this type...'.

Consider the program segment given below:

```
{calculate the greatest common divisor of u & v
   both > 0}
        while v ≠ 0 do
                begin r:=u mod v;
                      u:=v;
                      v:=r;
                      end
        {the GCD is contained in u}
```

This program segment calculates the greatest common divisor (GCD) of u and v if the variables u, v and r are of type integer. Since the algorithm for calculating the GCD of polynomials is identical to that of integers, it would be nice if the above program segment also worked for variables u, v and r of type polynomial. This assumes that the operators :=, ≠, and mod can be defined for values of type polynomial.

Simula 67's (Dahl [2]) class mechanism allows the association of procedures with user defined types i.e. they can only be used with variables of the type they are associated with. The notation is inelegant and restrictive compared to what we are going to propose. Also infix and prefix operators cannot be defined.

In Pascal, we are allowed to define new types as sets of values. The operators :=, =, and ≠ are automatically provided by Pascal, as should all high level languages, for all types including those defined by the user. The relational operators $\geq$, >, < and $\leq$ are defined for scalar types and a subrange of type t has all the operators and functions of type t defined for it.

1a. Operator Overloading.

The concept of operator overloading i.e. the definition of existing operators for additional operand types is not new, having been used in the MAD language [8] and discussed by Hoare [6]. We extend Pascal by providing an overloading mechanism similar to that of Hoare. Since operators are unary prefix or binary infix functions we have tried to keep our operator overloading syntax similar to that of function and procedure definitions. Only existing operators may be overloaded. New operators cannot be defined. Operator priorities, associativity etc. are not altered by overloading. An operator that is unary only (e.g. not) may not be overloaded as a binary operator. Similarly, the pure binary operators (e.g. *) may not be overloaded as unary operators.

Informal syntax

i. operator to be overloaded as a function

```
function operator (1 or 2 formal parameters):
                                   result_type;
         {the body is similar to a function
```

body except that the value is
returned by using the identifier
result instead of the function name}.

ii. operator to be overloaded as a procedure

procedure operator (1 or 2 formal parameters);

{the body is similar to a
procedure body}.

Example

The operators +, - and * are overloaded below for the type complex which is defined as:

```
type complex = record i,r:real end
```

i).
```
function +(var a,b:complex):complex;
  begin result.r:=a.r+b.r;
        result.i:=a.i+b.i;
        end
```

ii).
```
function -(var a,b:complex):complex;
  begin result.r:=a.r-b.r;
        result.i:=a.i-b.i;
        end
```

iii).. Unary minus is defined as:

```
function -(var a: complex):complex;
  begin result.r:=-a.r;
        result.i:=-a.i;
        end
```

iv).

```
function *(var a,b:complex):complex;
  begin result.r:=a.r*b.r-a.i*b.i;
        result.i:=a.r*b.i+a.i*b.r;
        end
```

If the application of an operator does not return any value, as in the case of the assignment operator, we must then use the procedure definition to overload the operator.

Example

Suppose we want a boolean variable to acquire the value false if it is assigned the integer value 0 and the value true if it is assigned any other integer value. This is accomplished by overloading the assignment operator as follows:

```
procedure :=(var a:boolean; var b:integer):boolean;
      if b=0 then a:=false else a:=true;
```

1b. Procedure Overloading.

Assume we have defined the function

```
function PROD(var x:array[*] of integer; n:integer):
                                          integer;
  var p:integer;
  begin p:= 1;
        for i:=1 to n do p:=p*x[i];
        PROD:=p
        end
```

then we should also be able to define

```
function PROD(var x:array[*] of complex;
                              n:integer):complex;
   var i:integer; p:complex;
   begin p:=1;
        for i:= 1 to n do p:=p*x[i]
        PROD:=p
        end
```

because both definitions of PROD conceptually do the same thing, i.e. calculate the product of the elements of an array. We, therefore, allow names of procedures that have been defined before, to be used in defining additional procedures, having either different formal parameter types or a different number of formal parameters. This is called <u>procedure overloading</u>. For a call to an overloaded procedure, the selected procedure definition is the one in which the number of formal parameters equals the number of actual parameters and whose formal parameter types match the actual parameter types.

The overloaded definitions of operators and procedures should be consistent with the actions performed by the existing definitions. For example, it would be confusing and extremely bad programming if the integer or real addition operator + was overloaded to perform the multiplication of complex operands.

Defining an operator or procedure for the same set of parameter types more than once is of course illegal.

lc. Subtypes.

A problem exists in the second definition of the function PROD in the last section (II.1b); the assignment statement p:=1 is incorrect because p is a complex variable while 1 is an integer constant. Integers can be considered to be complex numbers with their imaginary parts being equal to 0; we should be able to inform the language processor of this so that it should convert integers to complex numbers when necessary. Of course we will have to supply information on how the conversion is to be performed. Pascal does some type conversion for us; for example, the integers are converted to reals because the integers are a subset of the reals.

There are several solutions to the above problem. We could insist that the programmer explicitly specify any type conversion needed. This would require him to use the complex constant complex(1,0) instead of the integer 1 thus writing the above statement as p:=complex(1,0). The statement will now not work for p of type integer or real. This solution is too rigid for the concepts introduced in the next chapter.

Another solution is to overload the assignment operator as

```
procedure := (var a:complex; x:integer);
   begin a.r := x;
         a.i := 0 end
```

Though this solves the problem of type conversion for the assignment operator, making p:=1 legal, it does not solve the general type conversion problem. For example, the expression p+1 would still be illegal.

A general solution would be to allow the programmer to specify implicit type conversion from one type to another, Fischer [3]. However, <u>implicit conversion should occur only when one type is a subset of another</u> - for example, from 1..20 to integer, from integer to real and from real to complex. Otherwise, roundoff errors and runtime efficiencies creep in. We propose the notation

```
function subtype (identifier:type_1):type_2;
           {statement which indicates how to.
             convert type 1 to type 2}
```

for implicit conversion subtype is a standard function that can be overloaded. For example,

```
function subtype(x:real):complex;
   begin subtype.r:=x;
         subtype.i:=0  end
```

This at least gives an indication that one is defining how types interact and not just how to convert one type to another. It is easy to check for and warn against circularity as for example in

```
function subtype (x:real):complex;
   begin subtype.r:=x;
         subtype.i:=0  end

function subtype (x:complex):real;
         subtype:=x.r
```

However, such circularity is not always a mistake although for purposes of clarity and efficiency we may not want to allow it. Consider the two types, rectangular coordinates and polar coordinates, defined as

```
rect_coord = record x,y:real  end
polar_coord = record r,theta:real  end
```

These two types are equivalent in that you can transform a value from one type to another and back again; ease of mathematical formulae manipulation necessitates such transformations. However, some information (accuracy) will be lost during these transformations on any computer.

2. Programming with Arrays.

Except for files, the array is the most commonly used data structure. In the rest of this chapter we shall focus on how to use and manipulate arrays with ease.

2a. A General for Statement

There has been a proliferation of different types of for statements in high level languages. Some of these are:

1. <u>for</u> x ∈ set <u>do</u> S
2. <u>for</u> y ∈ ordered_set <u>do</u> S
3. <u>for</u> i := 1 <u>to</u> n do S
4. <u>for</u> i := n <u>downto</u> 1 <u>to</u> S

Each <u>for</u> statement provides a way of specifying a set of values, an index variable that acquires each one of these values (in a pre-determined order if the set is ordered) and a statement S that is executing once for each value acquired by the index variable.

It is quite likely that in the future we shall see the development of additional types of <u>for</u> statements. For example, Wulf [15] talks about the disadvantages of not being able to write a <u>for</u> statement of the form

<u>for</u> <u>all</u> a ∈ array A <u>do</u> S

Consequently, it would be wise to consolidate these different types of <u>for</u> statements into one general <u>for</u> statement of the form

<u>for</u> <index variable> <u>in</u> <set of values> <u>do</u> S

The index variable type must be same as the base type of the set or a <u>super</u> <u>range</u> of it (type t is a super range of type s if s is a subrange of t).

The programmer no longer has to keep track of the syntax of the different <u>for</u> statements. All he needs to know is how to specify the set of values in the general <u>for</u> statement. We shall deal with this question after axiomatically defining the semantics of the general for statement using Hoare's [5] notation. Instead of having to give an axiom for each type of for statement, we now need only two axioms for the general for statement, one each for the ordered and unordered set of values.

Semantics of the General for Statement

a) Let

1. A and B be unordered sets such that $B \subseteq A$.
2. P(E) be an assertion about the unordered set E.

The semantics of the general for statement with an unordered set of values are

$$\frac{i \in A-B \wedge P(B)\ \{S\}\ P(B \vee [i])}{P([\ ])\ \{\underline{\text{for}}\ i\ \underline{\text{in}}\ A\ \underline{\text{do}}\ S\}\ P(A)}$$

b) Let

1. X be a ordered set such that $X = [x_1, x_2, \ldots, x_n]$ and $x_i < x_{i+1}$, $1 \le i < n$.
2. Q(F) is an assertion about the ordered set F.
3. [a..b) be the set $[y|a \le y < b]$
4. [a..b] be the set $[y|a \le y \le b]$

The semantics of the general for statement with an ordered set of values are

$$\frac{x_1 \le y \le x_n \wedge Q([x_1..y))\ \{S\}\ Q([x_1..y])}{Q([\ ])\ \{\underline{\text{for}}\ y\ \underline{\text{in}}\ X\ \underline{\text{do}}\ S\}\ Q(X)}$$

Specifications of the set of values for the general for statement

By consolidating the different types of for statements into one general one, we have reduced the problem of defining different kinds of iteration to a simpler problem of specifying different set of values. These sets of values may be specified in any one of the following ways:

i. Any Pascal expression that evaluates to set, Pascal sets are unordered.

Examples

```
type set10 = set of 1..10;
type weekday = (monday,tuesday,wednesday,thursday,
                friday);
var p,q,:set10; i,s:integer;
    w:set of weekday; day:weekday;
1. for i in p v q do s:= s+i;
2. for day in w do write(day);
3. for day in w-[friday] do workhard(day);
```

ii. Data types define a set of values (along with the operations allowable on these values). We, therefore, allow

a) finite scalar or subrange data types to be used in specifying the set of values in a for statement. The set of values represented by a scalar data or a subrange type is ordered.

Examples

```
type suit = (club, diamond, heart, spade];
var i:integer; j:suit;
1. for i in 1..10 do statement1
2. for j in suit do statement2
```

b) The data type tuple with components that are of finite scalar or finite set types. Consider Pascal records without variants. A record is a tuple. Let r be a record type with components $c_1, c_2, \ldots, c_n$ and corresponding types $t_1, t_2, \ldots, t_n$. Let x be a variable of type r. Then type r represents the unordered set of tuples

$$[c_1, c_2, \ldots, c_n \mid c_i \in t_i]$$

The corresponding index variable must be a record with n components $i_1, i_2, \ldots, i_n$ such that the type of i_k is the same as or a super range of t_k, $1 \le k \le n$.

Examples

```
type complex100 = record r,i: 1..100 end;
type month = (January, February, March, April,
              May, June, July, August, September,
              October, November, December);
type day = record d:1..31; m:month,
                  year:1976..1976 end;
```

```
var k:complex100; j:record r,i:integer end;
    a:array[1..75,1..80] of real; sum:real;
    total_sales:integer; data:day;
```

1. for k in complex100 do write(k.r, k.i)
2. for j in complex100 do a[j.r, j.i]:= 0.0
3. for j in record e: 1..75; m:1..80 end do sum:=sum+a[j.r, j.i]
4. for date in day do total_sales:= sales(date)+total_sales

where the function sales returns the amount of sales for a particular date. From example (3) we see that there is no need for specifying field names in record types used to specify sets of values as well as the keywords record and end. It would be more pleasing to be able to write the for statement of example (3) as

for j in 1..75, 1..80 do sum:=sum+a[j.e, j.m]

To enable the above, we define a new kind of record data type, of the form

$t_1, t_2, \ldots, t_n$

which is an abbreviation for the record type

record comp1:t_1; comp2:t_2;..;compn:t_n end

c) An array may be thought of as a function whose domain is the set of all valid subscripts and whose range is the of values represented by the element type. To iterate over all the elements of an array, we need to generate all the valid subscripts. So as to be able to do this easily, we introduce a new type called the domain of an array.

The domain of an n-dimensional ($n \geq 1$) array A, written as domain(A), is $t_1, t_2, \ldots, t_n$ (i.e. record compl:t_1; comp2:t_2;..;compn:t_n end) where t_i is the type of the index of the i^{th} dimension of A (the domain of an array cannot be an integer array because Pascal arrays can have non-numeric indices).

Examples

```
type suit = (club, diamond, heart, spade);
var a:array[1..75,1..75] of integer;
     deck: array[suit,1..13] of integer;
     i:domain(a); card:domain(deck);

1. {set diagonal elements to 1. others to 0.}
     for i in domain(a) do
        if i.compl=i.comp2
          then a[i.compl,i.comp2] := 1
          else a[i.compl,i.comp2] := 0

2. for card in domain(deck) do deal(card)
```

The first thing one notices is that nested for statements are no longer needed to iterate over the elements of an n-dimensional ($n>1$) array.

In example (1), it would be elegant to be able to refer to the array element a[i.compl,i.comp2] as a[i]. We, therefore, allow the element of a n-dimensional array x to be referred to as x[r] where r is a record with n components. If the type of r is <u>record</u> $i_1{:}t_1;\ i_2{:}t_2;\ldots;$ $i_n{:}t_n$ <u>end</u> then x[r] is equivalent to $x[r.i_1, r.i_2,\ldots, r.i_n]$. Example (1) may now be written as:

```
{set diagonal elements to 1. Others to 0.}
   for i in domain(a) do
        if i.compl=i.comp2
             then a[i] := 1
             else a[i]. := 0
```

We also allow, for the sake of convenience, user preference and conventionality, the replacement of an index variable of type record with n components $c_1, c_2,\ldots, c_n$ by a list of variables $k_1, k_2,\ldots, k_n$ such that the type of k_i is the same as that of c_i or a super range of it, $1 \leq i \leq n$. Example (1) may now be written as:

```
{set diagonal elements to 1. Others to 0.}
   for j, k in domain(a) do
      if j = k
          then a[j,k] := 1
          else a[j,k] := 0
```

The variables j and k are type integer.

<u>Changing the order of scalar types and the ordering of the set of values represented by tuple (i.e. Pascal record) types.</u>

i. Any ordered type may have its order reversed by being prefixed by rev. The resulting type is also ordered. Suppose

type s = rev t

The set of elements represented by type s is the same as that represented by type t. The elements of type s have the same properties as the elements of type t except the functions pred, succ and the relations $<$, $\le$, $>$ and $\ge$. Their new definitions are given below.

Let

1. a and b be elements of type s.
2. x and y be elements of type t such that x = a and y = b.

Then

1. pred(a) = succ(x)
2. succ(a) = pred(x)
3. $a \le b$ if $x > y$
4. $a < b$ if $x \ge y$
5. $a \ge b$ if $x < y$
6. $a > b$ if $x \le y$

ii. A tuple type whose components are finite scalar types may be ordered by prefixing it with either row or col. The ordering produced is defined below.

Definition

Reverse lexicographic order is defined as lexicographic order in which the i^{th} component of the tuple from the right hand side is considered to be the i^{th} most significant (in lexicographic order the i^{th} component from the left hand side is considered to be the i^{th} most significant).

Let

1. r be a record type with components $c_1, c_2, \ldots, c_n$ having finite scalar types $t_1, t_2, \ldots, t_n$.
2. x be a variable of type r.

The set of tuples S represented by type r is

$$S = [c_1, c_2, \ldots, c_n \mid c_i \in t_i]$$

1. The ordered type row r represents the set S ordered in increasing lexicographic order.
2. The ordered type col r represents the set S ordered in increasing reverse lexicographic order.

The ordering of row r or col r may be reversed by prefixing them with rev.

3. The ordered type rev row r represents the set S ordered in decreasing lexicographic order.

4. The ordered type rev col r represents the set S ordered in decreasing reverse lexicographic order.

All the ordered tuple types described above have the functions succ and pred and the operators $<$, $\leq$, $>$ and $\geq$ defined for their elements.

Examples

```
var z:array[1..100,boolean] of real;
    i:domain(z)
    j,k,m:integer;
    b:boolean;
    q;record i:boolean; d:char end;

1.   for i in row domain(z) do statement_1
2.   for i in rev row domain(z) do write(z[i])
3.   for j,b in col domain(z) do
                    if b then z[j,b]:=1.0 else z[j,b] :=0.0
4.   for j,k in 1..100, rev 1..75 do statement_a
5.   for j,k,m in rev col 1..10,rev 10..70;rev 15..90 do s3
6.   for q in row false..true, 'a'..'z' do s4
```

A final note on the general for statement

For user convenience, we allow more than one set of values to be specified in the for statement. Let v_1, v_2,..., v_n be sets of values, Then the for statement for i in v_1; v_2;..; v_n do S

is an abbreviation for

for i in v_1 do S;
for i in v_2 do S;
.
.
for i in v_n do S

Examples

var i,j:integer;

1. for i in 1..20; 30..40; rev 70..80 do S
2. for i,j in 1..10, 1..10; rev 60..65, 60..65 do S2

2b. Some Standard Functions

The following functions are added to Pascal as we feel that they will be very useful in light of the extensions we have already made and the concepts to be introduced in the next chapter.

1. ℓbound

ℓbound(x,i) returns the lower limit of the i^{th} dimension of the m-dimensional array x.

ℓbound(x) returns a record with m components comp1, comp2,.., compm such that ℓbound(x).compi = ℓbound(x,i)

2. ubound

ubound(x,i) returns the upper limit of the i^{th} dimension of the m-dimensional array x.

ubound(x) returns a record with m components compl, comp2,.., compm such that ubound(x).compi = ubound(x,i)

3. rank

The rank of an m-dimensional array is defined to be m. In the next chapter we discuss procedures that accept parameters of different types including arrays of any rank; it will be useful to have a function that returns the rank of a parameter.

Examples

```
var a:array[1..10,1..10,1..10] of real;
    b:array[1..20] of array[boolean] of char;
    c:array[10..95] of record x,y:real end;
    e,f:real;

    rank(a) = 3
    rank(b) = 2
    rank(c) = 1
    rank(e) = rank(f) = 0
```

4. min

min(t) returns the smallest value of the ordered finite type t (in case of reals and integers the smallest value representable is returned).

5. <u>max</u>

max(t) returns the largest value of the ordered finite type t (in case of reals and integers the largest value representable is returned).

6. <u>base_type</u>

In the next chapter we develop a syntax for allowing parameter types to be parameters themselves. Procedures will then be able to accept arrays and sets of any element type. To be able to use the type of array or set elements in declaring local variables or in conditional statements, we introduce the function base_type. We illustrate base_type using the variables declared in the preceding section (II.2b.3).

<u>Examples</u>

i. base_type(a) = real
ii. base_type(b) = char (not <u>array</u>[boolean] <u>of</u> char)

2c. The Syntactic List Generator, The Array Generator and Array Sections.

Often we need to write procedures that accept

i. a variable number of actual parameters, or
ii. a list of expressions as actual parameters instead of an array and vice versa.

Variable number of actual parameters are allowed by high level languages when using pre-defined language procedures. For example, the function MAX in PL/I accepts a variable number of actual parameters and returns the value of the maximum actual parameter. Similarily the Pascal procedure write accepts a variable number of actual parameters and writes them onto an output file. However, high level languages do not allow the user to define his own procedures that accept a variable number of actual parameters. Programmers have to use explicitly defined arrays if they want to simulate a procedure that accepts a variable number of arguments.

Also, the high level languages do not allow the substitution of an one dimensional array for a syntactic list of expressions and vice versa.

These features would be very convenient to have. In the 'calculator mode' of an interactive language (in which the value of an expression is returned immediately after it is typed in), it would be very convenient to be able to write the expression

```
standard_deviation(...variable number of actual
                                    parameters...)
```

instead of the sequence of statements

```
declare x to be an array;
assign values to the elements of x;
standard_deviation(x)
```

In case array x already exists or if the data values are to be stored in x for later use or manipulation, then the second alternative is desirable.

The above mentioned capabilities are provided in the language by the introduction of

1. The syntactic list generator
2. The array generator
3. Array sections (PL/I array cross sections particular cases of array sections)

1. The Syntactic List Generator.

list is a unary operator that takes an array as an argument and returns a syntactic list of the array elements. Syntactic lists are not data types.

Let a be of type array[p..q] of t where t is any type. Then list a is an abbreviation for

a[p], a[succ(p)], a[succ(succ(p))],.., a[q]

Examples

i) One dimensional arrays can be used to index arrays a[list x] is an abbreviation for a[x[p], x[succ(p)],.., x[q]] where

a. type(x)=array[p..q] of t
b. t is integer or a type convertible to integer
c. number of elements of x $\leq$ rank(a)

In case the operator list is left out a[x] will be coerced to a[list x].

ii) Procedures that expect lists of expressions as actual parameters instead of an array of these expression values can now be made to work easily for arrays.

a. Consider the PL/I built-in function MAX that returns the maximum actual parameter vale out of a list of a variable number of actual parameters. It can now easily be used to find the maximum element of an array.

```
var h,i,j:integer;
    a:array[1..50] of integer;
    b:array[40..100] of integer;
```

```
MAX(h,i)       returns the maximum of h and i
MAX(list a)    returns the maximum element value
               in a
MAX(list a,h,i)   returns MAX(h,i,MAX(list a)
MAX(list a,list b)   returns MAX(MAX(list a,
                                 MAX(list b))
```

b. Similarily, the Pascal procedures read and write can be used to input and output arrays.

2. The Array Generator.

The array generator allows the user to construct expressions of type array[a..b] of t using expressions of type t. The array generator proposed by us is similar to that of Hoare [6]. In addition to the advantages of array generators mentioned by Hoare, like being able to assign different values to the elements of an array in one assignment statement. We shall show how the array generator can be used to simulate passing a variable number of actual parameters to procedures. The array generator has the form

$$[a_1, a_2, .., a_n] \ \{(p..q)\}^1$$

where

i. $\underline{\{\ \}}$ are meta brackets. $\underline{\{x\}}^1$ means that x may or may not be included.

ii. $a_1, a_2, .., a_n$ are expressions

iii. p and q are expressions of the same type such that $p \leq q$.

iv. In case (p..q) is left out, default values of

$p = 1$ and $q = n$ are assumed.

Let

i) x be the array generated by the above array generator expression. Then type(x) = <u>array</u>[p..q] <u>of</u> t, where t = type(a_1).

ii) $c_1, c_2, .., c_m$ be the elements p, succ(p), succ(succ(p)),.., q. The elements of x have values as defined below.

a) $x[c_i] = a_i$ $\quad 1 \le i \le$ mininum of m and n
b) $x[c_i] = a_n$ $\quad n + 1 \le i \le m$

Some array generation examples are

1. ⦃1.0⦄(1..100)
2. ⦃b,e⦄(100..110)
3. ⦃list i, list j⦄ where i and j are integer arrays
4. ⦃1,1,1,1⦄('a'..'b')
5. ⦃1,10,k,1⦄(2..3)

Suppose A is of type array[1..4] of integer. Then A:=⦃5,7,4,6⦄ is a valid statement.

Consider the procedure plot and the function standard_deviation given below:

```
procedure plot(var x,y:array[*] of real);
  .
  .
  begin
      .
      .
      plot x_i versus y_i
      .
      .
      end

  function standard_deviation(var x:array[*] of real):
                                          real;
     var xmean, s:real, i:integer;
     begin xmean:=sum(x)/(ubound(x,1)-ℓbound(x,1)+1);
           s:=0
           for i in domain(x) do
             s:=sqr(x[i]-xmean)+s;
           standard_deviation:=sqrt(s/ubound(x,1)
                                    -ℓbound(x,1)+1));
           end.
```

where sum returns the sum of the elements of a real array.

Skipping the usual way of calling standard_deviation and plot with arrays, we show some calls that make use of the syntactic list generator and the array generator. The e_i's and the f_i's are real expressions

```
var a,b:array[1..50] of real;
    c.d:array[30..100] of real;
```

1. standard_deviation(ǂe_1,e_2,..e_{30}ǂ) — we simulate passing a variable number of actual parameters

2. standard_deviation(ǂlist a,list b,list c,list dǂ) — returns the standard deviation of the elements of a,b,c and d taken together

3. plot(ǂe_1,e_2,..,e_{40}ǂ,ǂf_1,f_2,..,f_{40}ǂ — Simulates passing a variable number of actual parameters.

4. plot(ǂlist a,list cǂ,ǂlist b,list dǂ) — plot a and c versus b and d.

3. Array Sections.

If A is array then the array section A[k1..k2] refers to the elements A[k1], A[succ(k1)],..,A[k2] where k1 $\leq$ k2. If k2 < k1 then the section is empty. Array sections may be used in aggregate operations.

The section A[k1..l2][m1..m2] refers to the elements

A[k1..k2,m1..m2] = A[k1][m1..m2], A[succ(k1)][m1..m2],..., A[k2][m1..m2]
= A[k1,m1..m2], A[succ(k1),m1..m2],..., A[k2, m1..m2]

i.e. the portion of A that contains the elements A[i,j] such that $k1 \leq i \leq k2$, $m1 \leq j \leq m2$.

The PL/I cross section notation is also allowed. Cross sections are special cases of array sections.

a[*] is an abbreviation for A[ℓbound(A,1)..ubound(A,1)].

Examples

1. The convenience of cross sections is well known and numerous examples are readily available.

2. The following function bs uses binary search to find the index of x in an array a. If x does not occur in a then max(integer) is returned.

```
function bs(var a:array[*] of integer;
                    var x:ingeter):ingeter;
    var p,q,k:integer;
    begin p:=ℓbound(a,1); q:= ubound(a,1); k:=(p+q)/2;
      if a[k] = x
         then bs:=k;
         else if q = p
            then bs:= max(integer)
            else if x < a[k]
               then if p = k
```

```
            then {no elements with values
                 smaller than a[k]}
                 bs:= max(integer);
            else bs:=bs(a[p..k-1]);
          else if q = k
            then {no elements with values >
                 a[k]}
                 bs:= max(integer);
            else bs:= bs(a[k+1..q]);
    end
```

<u>Some Properties of the Syntactic List and Array Generators</u>

The array generated by the array generator has the base element type which is the same as that of the first operand. If we want an array of another base element type we can implicitly or explicitly force the first operand of the array generator to be of that type. For example:

```
ǂtrunc(4.39+x),b,cǂ  produces an array of integers
ǂ1/3,5,7,5,6,8ǂ      produces an array of reals
```

The identities given below make use of the following declarations:

```
var a:array[pa..qa] of t;
    b:array[pb..qb] of t;
```

Let d_1, d_2,..., d_n be expressions of type t and p and q be expressions of type s.

1. <u>list</u> a = a[pa], a[succ(pa)],..., a[qa]

2. list list a = list a[pa], list a[succ(pa)],..., list a[qa]

3. $\{d_1, d_2, \ldots, d_n\}$ = T where i) type(T) = array[1..n] of u

ii) u = type(d_1)

iii) T[i] = d_i $1 \le i \le n$

4. $\{d_1, d_2, \ldots, d_n\}$ (p..q) = T where

i) type(T) = array[p..q] of u

ii) u = type(d_1)

iii) Let $c_1, c_2, \ldots, c_m$ be the elements p, succ(p),..., q

a) T[c_i]=d_i $1 \le i \le n$
minimum of m be n

b) T[c_i]=d_n $n+1 \le i \le m$

5. list a[p..q] = list(a[p..q])

6. Array concatenation

$\{$list a, list b$\}$ =

$\{$a[pq], a[succ(pa)],..., a[qa], b[pb], b[succ(pb)],..., b[qb]$\}$

Chapter III TYPES OF PARAMETERS AS TYPES

1. Introduction.

Current programming languages with types require that the type of a formal parameter be determined from the procedure of which it is a parameter. For example, if we want to sort an array of integers, an array of reals and an array of characters we must write three different procedures in Pascal.

i)
```
procedure sort_integer_array(var x:array[*] of
                                        integer);
   var t:integer; {temporary variable for
                                     swapping}
   begin {bubble sort}
       .
       .
       .
       end
```

ii)
```
procedure sort_real_array(var x:array[*] of real);
   var t:real;  {temporary variable for swapping}
   begin  {bubble sort}
       .
       .
       .
       end
```

iii)
```
procedure sort_char_array(var x:array[*] of char);
   var t:char;  {temporary variable for swapping}
   begin  {bubble sort}
       .
       .
       .
       end
```

In essence, when writing the sort procedure we are not concerned with the type of the values being sorted, we are only interested in the fact that a relation '>' (comparision) and an operator ':=' (assignment) are defined on the type.

This kind of thing happens quite often in programming; the programmer does not want to deal with values of a particular type, instead he is concerned with the <u>operations to be performed on the values</u>.

This abstraction from a data type to the operations performed on the type is so important that we feel that the language should support it. This part of the thesis is devoted to exploring a notation for it, to giving many examples of its use, and to discussing an efficient implementation. We will see that the concept is quite useful and flexible, so much so that we will be able to write below a single sort procedure which will sort any array of values on which the operations := and > are defined, effectively collapsing three procedures into one.

```
iv)   procedure sort(var x:array[*] of <p>);
         var t:p;  {temporary variable for swapping}
         begin  {buble sort}
            .
            .
            .
            end
```

The angle brackets < and > specify that the type of the array elements p is determined by the procedure call; note how the type identifier is used in the declaration of variable t.

If sort is called at different places with arrays of real, integer and char then p correspondingly assumes the 'values' real, integer and char. The type p is said to be variable or a parameter since it is deduced from the procedure call.

Another example is the procedure swap defined below:

```
procedure swap(var x,y:<q>);
   var t:q;
   begin t:=x;
         x:=y;
         y:=t   end
```

The procedure swap may be called with two actual parameters of any type provided they both have the same type; it works for all types that have the assignment operator := defined for their elements. The call swap(A,I) leads to a type error if A is real and I is integer.

As a final example consider

```
procedure x(var r:record p:<z>; q:real  end;
                    r:set of <y>);
   var q:array[1..10] of z
   begin
       .
       .
       .
       end
```

Procedure x may be called with two parameters A and B such that A is a record with two components and B is a set of any element type. The first component of A can be of any type but the second must be real. z acquires the 'value' which is the same as the type of A's first component while y gets the value that is element type of B.

In letting formal parameter types be parameters we shall allow the following to be variable:

i) the type of a formal parameter

ii) the component types of a structured type formal parameter

iii) the rank of an array formal parameter

We now give examples of (i) and (ii); examples of (iii) are given later.

	formal parameter type	meaning
i.	<q>	The type q is to be determined from the procedure call.
ii.	array[*,*] of <a>	The corresponding actual parameter must be a two dimensional array. The array component type is to be determined from the procedure call.

formal parameter type	meaning
iii. record a:<y>; b:array[*] of <z>; c:real end	The corresponding actual parameter must be a record with 3 components. The type of the first component is variable; the second component must be a one dimensional array of any type; the third component must be of type real.

In the ensuing sections we discuss the advantages of letting parameter types be parameters themselves, develop for this by series of examples, a syntax that is clean, natural, systematic, similar to Pascal's syntax and at the same time allows an efficient implementation. We also show the powerful capabilities of this concept.

2. Advantages of Letting Parameter Types be Parameters.

i) Reduced programming effort

Procedures that are systematically identical in all respects except formal parameter types and related local variable types may now be collapsed into one procedure. The programmer does not have to worry about matching the precision and other attributes of the formal and actual parameters. The procedure will work as long as the operations in it are defined for the actual parameter types. (In PL/I if one tries to swap two FIXED DECIMAL variables of precision (4,0) by calling a procedure that expects FIXED DECIMAL actual parameters of precision (5,0), there will be either a type error or a call by value resulting in no change in the values of the two variables).

ii) Programs become more managable

Program listings become smaller and less storage will be required to store the source code since there will be fewer procedures. Program correctness proofs become smaller but now we must also specify the types of actual parameters for which the proofs hold.

iii) Abstraction

By letting parameter types be parameters, we are really abstracting the control structure behind ordinary procedures. If a formal parameter type has the form <x> then identifier x represents an abstraction of the formal parameter type just as variables represent an abstraction of their value. The current 'value' of x is left to the language processor to figure out. This is in accordance with the concept of programming by refinement which encourages us to concentrate on the algorithm without worrying about the details.

iv) Portability across types

If a programmer changes the types of one or more of the actual parameters he may not have to change his procedure at all provided the procedure was written to accept different types of actual parameters. In case of ordinary procedures a change in actual parameter types results in changing formal parameter types. For example, suppose initially an integer array is used to represent a stack and a procedure push is defined for it. If the programmer decides to use a real array to represent the stack at some later stage, he will have to modify procedure push. However, had he defined a procedure that accepted actual parameters of different types he would not have to change anything.

3. Restricting the acceptable types of actual parameters

Consider the following function overage that returns the average value of the elements of an array:

```
function average(var a:array[*] of <z>): z;
  var i:domain(a);
      number:integer;
      sum:z;
  begin
      number:=0;
      {compute sum}
         sum:=0;
         for i in domain(a) do
                 begin sum:=sum+a[i];
                       number:=number+1
                       end
      average:=sum/number
      end
```

Average accepts an array of any index and element type and returns a value which is of the same type as the array elements and is their average. The function average will only work for real and integer arrays (assuming no overloading has been done for other types). If average is called with arrays of any other type say char the language processor detects an error, but only when it tries to generate code for the function body by finding that '+' and '/' are not defined for char actual parameters.

We would like a way of indicating to the reader of the program and to the compiler (to enable easy compile time checking) what types of actual parameters are permissible.

This restriction of actual parameter types is desirable for the following reasons:

1. The operators, functions or procedures in a procedure may not be defined for arbitrary actual parameter types.

2. A legal call (as the procedure writer views it) to a procedure may be made only with certain types of actual parameters even though (1) may not be true.

Consider the follwoing procedure print that accepts arrays of any index and element types and writes them onto text file f.

```
procedure print(var b:array[*] of <x>)
   var i:domain(b);
   for i in row domain(b) do write(F,b[i])
```

We want to restrict the use of print to write only integer or real arrays onto file F even though print will work for char arrays because user's of file F expect it to contain only numerical data.

The function average and the procedure print may be restricted to accepting only real and integer arrays by changing their headings to

```
i)  function average(var a:array[*] of <z> in
                                   (real, integer)):z;

ii) procedure print(var b:array[*] of <x> in
                                   (real, integer));
```

In general if a type is of the form <p> then the identifier p is restricted to being one of the types $t_1, t_2, \ldots, t_n$ by modifying <p> to <p> in $(t_1, t_2, \ldots, t_n)$.

Some examples of calls to average and print are:

```
var i:array[1..50] of integer;
    a,b,c,d,e:integer;

1. average(i[5..25])
2. average(ǂlist i[4..20],a,b,c,d,eǂ)
3. print (i)
```

The first call returns the average of the 5^{th} through 25^{th} elements of i while the second returns the average of the 4^{th} through 20^{th} elements and a,b,c,d and e.

4. Types in Conditional Statements

Till now we have discussed procedures which do the 'same' thing for different types of actual parameters. However, we would like to be able to do different things depending upon the actual parameter types. This variation of action is made possible by allowing the use of type identifiers, and actual types (e.g. array[*] of x, array[1..100] of real) in boolean expressions.

This allows the programmer to perform conditional actions depending upon the type represented by a generic type. For example, we can have statements of the form

```
if book=real then statement1
if z=array[*] of real and q=real
                  then statement1
                  else statement2
```

We now give a few examples to illustrate the usefulness of the above.

Example 1

In constructing a symbol table a different hash routine is to be used for integers, reserved_words, identifiers etc. The following is a procedure that inserts different types of elements into a symbol table using different hash routines for different symbol types.

```
procedure insert_into_symbol_table(var s:symbol_table;
                                   var x:<symbol>);
  var i:integer;
  begin
      .
      .
      .
      if symbol=integer
           then i:=hash_integer(x);
           else if symbol=reserve_word
                then i:=hash_rw(x);
                else if symbol=string
                     then i:=hash_id(x);
                     else i:=hash(x);
      .
      .
      .
      end
```

Note: reserved_word is assumed to be a user defined type

Example 2

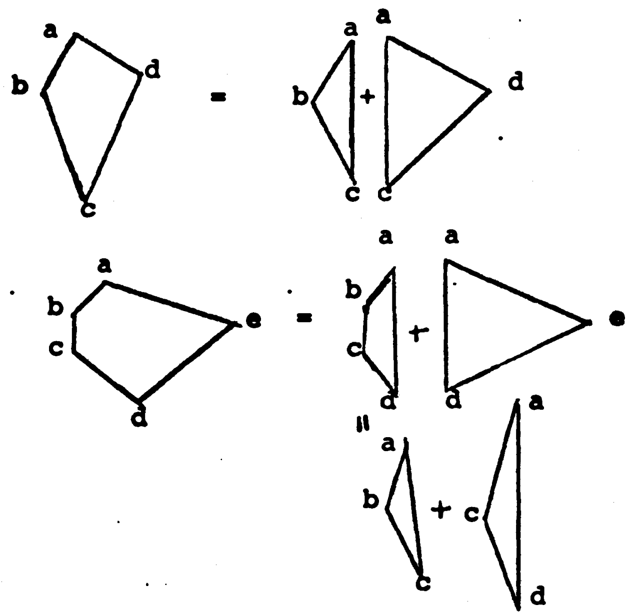
The following procedure multiplies real, integer, and boolean matrices with numerical indices. For boolean values, the logical operations and and or are used instead of

* and +. The conditional statements could have been avoided by suitably overloading the operators :=, + and * for boolean types.

3. Given the coordinates of the points of an arbitrary two dimensional convex shape in cyclic order, write a function to calculate its area. The area of a triangle is given by

area=abs(x1*y2+y1*x3+y3*x2-y2*x3-y1*x2-x1*y3)/2.0

where xi, yi i=1,3 are the coordinates of the vertices of the triangle. We use recursion as shown below to calculate the area of an arbitrary two dimensional convex shape. The area of a convex shape with n vertices=area of a convex shape with n-1 vertices+area of a triangle.



```
type complex=record x,y:real  end
type int_complex=record x,y:integer  end
function area(var a:array[*] of <z> in(complex,
                                        int_complex,
  array[1..2] of real, array[1..2] of integer)):real

var p,q:integer;
begin p:=ℓbound(a,1); q:=ubound(a,1);
  if q-p+1=3
     then {calculate the area of the triangle}
            if z=complex or z=int_complex
               then area:=abs(a[p].x*a[p+1].y+a[p].y
                         *a[q].x+a[q].y*a[p+1].x-a[p+1].y
                         *a[q].x-a[p].y*a[p+1].x-a[p].x
                         *a[q].y)/2.0;
               else area:=abs(a[1,1]*a[2,2]+a[1,2]*a[3,1]
                         +a[3,2*a[2,1]-a[2,2]*a[3,1]
                         -a[1,2]*a[2,1]-a[1,1]*a[3,2])
                                                  /2.0;
     else area:=area(a[p..q-1])+area(⸢a[1],a[q-1],a[q]⸣);
end
```

5. Procedures that accept Arrays of any Number of Dimensions

Our notation for allowing parameter types be parameters allows us to write procedures that accept arrays of any number of dimensions, i.e. any rank, but it is clumsy and inelegant as illustrated by the following example.

AMAX is a function that accepts an array of <u>any rank</u>, <u>any index and element types</u>.

```
function AMAX(var A:array[*] of <q>): base_type(A);
  var i:domain(A);
  begin  {set AMAX to the first element}
         i:=lbound(A);
         AMAX:=A[i];

         {compute the maximum}

         for i in domain(A) do
            if AMAX < A[i] then AMAX:=A[i]
         end
```

The formal parameter declaration specifies that the actual parameter corresponding to the formal parameter A must be a one dimensional array of any type thus accepting arrays of any rank. The value returned has a type which is the same as the base type of the actual parameter array. AMAX computes the maximum of the elements of the array provided the operator < is defined for the base type of the array.

We are unable to specify directly to the language processor by using in( ), that the actual parameter A is to be restricted to being arrays of some component type (e.g. array[*] of real, array[*,*] of real, array[*,*,*] of real, etc). As a consequence we introduce a new notation that allows us to do this.

notation for specifying formal parameter types	meaning
1. $\underline{\text{array}}^1$ of real	array[*] of real
2. $\underline{\text{array}}^2$ of char	array[*,*] of char

notation for specifying formal parameter types	meaning
3. array^2 of <z>	The actual parameter corresponding to a formal parameter of this type may be an array of rank 2 and any element type. This type is not equivalent to array[*,*] of <z> since this specifies that the corresponding parameter must be an array of rank at least 2 and of any element type (because z may be an array type)
4. array <k> of real	The corresponding actual parameter may be a real array of rank at least one. In the procedure k behaves like a local integer variable and has a value equal to the rank of the actual parameter.
5. array <k> of <z>	same as above except that the array may be of any element type.

The function AMAX may now be elegantly written as:

```
function AMAX(var A:array <P> of <x> in (real,integer)):x;
  var i:domain(A);
  begin {set AMAX to the first element}
        AMAX:= A[ℓbound(A)];
        {compute the maximum}
        for i in domain(A) do
             if AMAX < A[i] then  AMAX:=A[i];
        end
```

Instead of setting AMAX to the first element of A we could have set it to the smallest element of type x with the statement AMAX:=min(X).

6. Examples to illustrate the Versatility of the Concepts introduced so far

i. The following is some computation involved in Fourier Transforms.

$$c_{i1} = \sum_{j1} a_{j1} * r^{i1+j1}$$

$$c_{i1,i2..,in} = \sum_{j1}..\sum_{jn} a_{j1;j2,..,jn} * r^{i1*j1+i2*j2+..+in*jn}$$

$$= \left\{ r^{i1} \sum_{j2}..\sum_{jn} a_{1,j2,..,jn} * r^{i2*j2+i3*j3+..+in*jn} \right.$$

$$\left. +..+ r^{m*i1} \sum_{j2}..\sum_{jn} a_{m,j2,..,jn} * r^{i2*j2+i3*j3+..+in*jn} \right\}$$

where $1 \le ik, jk \le m$ for $k=1,n$

The recurrence relationship is as follows:

$c = F^n(a,r)$ c,a being n-dimensional arrays

$$c[i] = \sum_j r^{i*j} * F^{n-1}(a[j],r)$$

$$F^0(a[j],r) = a[j]$$

Using present day programming languages (with types) one can easily write a procedure to calculate c for any particular value of n. But one cannot easily and directly write a procedure for any value of n. However, the concepts intorduced by us make this possible.

```
procedure F(var a,c:array[*] of <z>; var r:<x> in
                                        (real,integer));
   var t:z; i,j:integer;
   if z=real or z=integer
      then for i:=1 to ubound(c,1) do
           begin c[i]:=0;
             for j:=1 to ubound(a,1) do
                 c[i]:=c[i]+a[j]*r**(i*j);
             end
   else for i:=1 to ubound(c,1) do
      begin c[i]:=0;
          for j:=1 to ubound(a,1) do
            begin F(a[j],t,r);c[i]:=c[i]+t*r**(i*j);
                  end
            end
```

Note: We have assumed that the operators *,+ and := are defined for arrays.

The procedure F, if written iteratively is much simpler.

```
procedure F(var a,c:array<N> of <z> in (real, integer);
                                                var r:z);
  var i,j:array[1..N] of integer;
  for list i in domain(c) do
      begin c[list i]:= 0;
            for list j in domain(a) do
              c[list i]:=c[list i]+a[list j]*r** sum(i*j)
            end

function sum(var a:array<N> of <z> in (real, integer)): z;
  var i:domain(a);
  begin sum:= 0;
        for i in domain(a) do
          sum:=sum+a[i];
        end
```

ii. The following subtype definition allows the use of any value i as an array of any rank, all of whose elements contain i. For example, one can write A:=20 or A:=A+35 where A is an integer array.

```
function subtype(x:<type>):array<N> of type;
  var I:domain(subtype);
  for I in domain(subtype) do subtype[I]:=X;
```

iii. The operator + may be overloaded for arrays of any rank and any element or index type as follows:

```
function+(var a,b:array<k> of <z>): array^k of z;
  var I:domain(a);
  for I in domain(a) do result[I]:=a[I]+b[I];
```

7. Correctness Proofs of Procedures that accept Actual Parameters of Different Types.

Allowing parameter types to be parameters themselves reduces the amount of proving required as we now have fewer procedures to prove correct. The proof of a procedure that accepts parameters of different types follows the same pattern as that of an ordinary procedure except that now we must also assert for what parameter types the procedure will work. There should be only one proof of correctness for each procedure regardless of the types of parameters it accepts.

Consider the informal proofs of the procedures swap and bubble sort of given below. Procedure swap will work only for those parameter types which have the assignment operator defined while bubble sort will work for those array types that have the assignment and the relational operators := and > defined and for which the procedure swap works. The assignmen operator is defined if the assignment axiom holds

$$\text{i.e.} \quad P_E^x \; \{x := E\} \; P$$

where the variable x and the expression E have the same type. We have used the notation of Hoare [5]. The relational operator > is defined if

i) $\neg(x > x)$
ii) $(x > y) \wedge (y > z)$ implies $x > z$
iii) $\forall x \forall y ((x > y) \vee \neg(x > y))$

In the above x, y and z have the same type.

```
        {swap will work only for those parameter
         types that have the assignment operator :=
         defined}

    procedure swap(var x,y:z);
      var temp:z;
      begin
          temp := x;
                    {temp = a and y = t}
          x := y;
                    {temp = a and x = t}
          y := temp
                    {y = a and x = t}
          end

        {bubble sort works for all array types that
         have the assignment and relational operators
         defined for which procedure swap works}

    procedure bubble_sort(var a:array[*] of z);
      var r,i,j:integer;
      {let s(i,j) be the assertion that the elements
       i through j of the array are sorted in non
       decreasing order}
      begin
        n:=ubound(a,1):
        j:=1;
        {s(1,j) is the invariant relation.  Upon
         termination s(1,j) and j=n implies s(1,n)}
          while j < n do
            begin
```

```
if a[j]>a[j+1] then {bubble up the element
                                        j+1}
   begin
     i:=j+1
     {invariant rel. is s(1,i-1) ~
                              s(i,j+1)}
     repeat
       swap(a[i],a[i-1];
       i:=i-1;
       until (i=1 v a[i-1]≤a[i])
     {s(1,j+1)}
     end
j:=j+1;
end
end
```

8. Implementation.

The concepts introduced in this chapter and the preceding one can be implemented efficiently only if the compiler can determine the exact type of every reference to every variable at compile time. If it can, then it is a straight forward matter to generate the right code for overloaded operators and procedures, determine implicit conversions and generate code for them etc. Otherwise, information about the type must be interpreted at runtime. We shall show that types of variables can be determined at compile time in most cases, the exceptions being pathological. Before doing so, we discuss cases where interpretation is actually more efficient.

Consider the function to sum the elements of an integer array.

```
function sum(var A:array<N> of integer):integer;
  var J:domain(A); s:integer;
  begin s:=0;
        for J in domain(A) do s:=s+A[J];
        sum:=s
        end
```

As the set domain(A) is unordered, it is left to the compiler to select the most efficient order for generating the subscript values for J. Independent of the rank of the array i, the compiler should implement this as a single loop treating A as a 1-dimensional array and J as an integer variable. The necessary information to do this (the number of elements of A, the address of the first element etc.) would appear in the dope vector passed along with the array actual parameter.

Our notation allows an intelligent compiler to avoid generating code for subscript checking by realizing that J is always a valid subscript for A. Of course, the compiler would have to make sure that J is used as a subscript for A only, its components are not explicitly referenced etc.

Let us now consider procedures that accept parameters of different types.

When the compiler encounters a call to such a procedure a copy of the procedure is made, the formal type identifier 'values' deduced from the call and these values are then used to textually replace the corresponding type through out the procedure copy (similar to macro expansion). The original call is replaced by a·call to this modified procedure which is then compiled. During compilation all easily evaluatable <u>if</u> statements, like those with just types in their boolean expressions are evaluated (this is just for the sake of efficiency; we could postpone the evaluation to run time if so desired). The fact that this procedure has been compiled for a certain set of actual parameter types is noted. If another call to this procedure is encountered and if its actual parameter types are the same as those for which the procedure has been compiled before then this call is simply replaced by a call to the corresponding modified procedure. No new version of the procedure need be compiled.

Consider the procedure S and the calls on it shown below:

```
procedure s(var x,y:<z> in (real,integer,char,complex))
  var t:z;
  statement

var a,b,c:real;
    l,m:string;
    u,v:complex;
```

1. s(a,b)
2. s(u,v)
3. s(ℓ,m)
4. s(b,u)
5. s(b,c)

During compilation s(a,b) is replaced by the call sl(a,b) where sl is a modified version of s with z being replaced by real and the if statement evaluated to select the proper alternative.

```
procedure s1(var x,y:real);
  var t:real;
  statement'
```

Similarly, s(u,v) is replaced by the call s2(u,v) where s2 is defined as

```
procedure s2(var x,y:complex);
  var t:complex;
  statement
```

As mentioned before, calls s(ℓ,m) and s(b,u) will result in type errors. s(b,c) is simply replaced by a call to sl i.e. sl(b,c).

The reader's initial reaction might be to protest against the extra processing done by the compiler; but one must remember that without the concept of letting parameter types be parameters, the user would have to write the procedures sl and s2 himself.

Can we always process procedures that accept different types of parameters in the 'macro like' fashion described above? We will now show that compile time processing of certain pathological cases of recursive procedures leads to the problem of infinite type generation. All other procedures, however, can be processed at compile time.

Infinite Type Generation

Consider the procedure P given below:

```
procedure P(var a:<k>; n:integer);
  var b:array[1..10] of k;
  begin
      .
      .
      if n < 5 then P(b,n+1);
      .
      .
      .
      end
```

Suppose P is called with the first parameter being real. Compile time processing leads to the generation of a modified copy of P (in which p has the type array[1..10] of real) which is then compiled.

Inside the modified copies of P there is a recursive procedure call to P with the first parameter being of type array[1..10] of real. Processing this leads to another recursive call to P with the first parameter being of type array[1..10] of array[1..10] of real. Further processing leads to an infinite number of copies of P being generated in which the first formal parameter has types real, array[1..10] of real, array[1..10] of real and so on.

Processing at run time (i.e. interpretation) is clearly a finite process since the second parameter has its value increased by 1 for every additional recursive call.

Is it possible to construct an algorithm which can detect infinite type generation?

We will answer this question after going through some preliminary remarks about the language G_3 and the Halting Problem.

G_3 is a universal programming language like Pascal, Fortran, Algol etc. . Any program written in these languages can be translated into G_3 and vice versa. G_3 has the following features:

1. An infinite number of variables x_i $i=1,2,\ldots\ldots$
2. The variables can have values ≥ 0, i.e. the positive integers.
3. A variable can be incremented by 1, i.e. $x_i := x_i + 1$
4. A variable can be decremented by 1, i.e. $x_i := x_i - 1$ where $0-1=0$
5. A conditional branch, i.e. <u>if</u> $x_i \neq 0$ then L, where L is the label of any G_3 program statement.

The G_3 programs can be enumerated.

ϕ_1
ϕ_2
.
.
. G_3 program list
ϕ_i
.
.
ϕ_k

It has been shown that the Halting Problem (HP) in general and the Halting problem on the G_3 program list index (HP_i) in particular are not decidable for G_3 programs Constable [1].

We will use $\phi_i(i)\downarrow$ to mean that program ϕ_i halts on input i and $\phi_i(i)\uparrow$ to mean that program ϕ_i diverges on input i (i.e. goes into an infinite loop).

<u>Theorem I</u>

There does not exist an algorithm A which when given a procedure that accepts actual parameters of different types and all related procedures that involve a recursive call sequence can decide whether the recursion leads to infinite mode generation or not.

<u>Proof</u>

Assume there exists such an algorithm A. We shall show that this leads to the decidability of the HP_i for G_3 programs.

Consider the G_3 list of programs. To see if $\phi_i(i)\downarrow$ or $\phi_i(i)\uparrow$ for a particular i, we construct the following program P.

```
program P;
  var c:real; y:integer;
  begin c:=0;
        s₀(c);

        procedure s₀(var a:<z>);
          var b:array[1..15] of z;
          begin
              Execute procedure Q for 1 second
              (or 1 statement) starting from its
```

```
saved state, if any.  At the end
save its final statement;

if y=0 then s.(b);

end

procedure Q;
  begin y:=0;
        φi(i);
        y:=1
        end

end
```

Notes i. y is not one of the variables of ϕ_i

ii. The first time we execute procedure Q, we will not have a saved state. We execute from the beginning (i.e. initial state).

If algorithm A tells us that there is infinite mode generation, then this implies that $\phi_i(i)\uparrow$. If it tells us that there is no infinite type generation, then this implies that $\phi_i(i)\downarrow$. Thus, our assumption of the existence of A leads to the decidability of HP_i for G_3 programs which implies that our assumption was wrong.

Theorem II

Even if we restrict the recursive call sequence (of Theorem I) to depend only upon if statements that can be evaluated at compile time (e.g. if x≠real then statement1), there does not exist such an algorithm A.

Proof

First we will show how to simulate G_3 programs using

1. <u>if</u> statements evaluatable at compile time
2. Procedure calls
3. Definition of procedures
4. Variable declarations

Then we will show that the existence of A implies the decidability of the HP_i for G_3 programs.

G_3	Pascal simulation
1. positive integers	arbitrary rank arrays of real elements
2. integer value 0	real
3. integer value i(> 0)	array of rank i
4. ≠ 0	≠ real
5. +1	construction of an array of one higher rank
6. -1	construction of an array one rank lower

We modify the G_3 programs as follows:

1. Add a clock variable. This counts the number of statements executed in the original program if the program loops then this will have an infinite value.

2. Relabel the statements in sequential order 1,2,3,... This might involve changing some labels in the existing program.

3. Change every statement of the form x:=x-1 to if $x \neq 0$ then x:=x-1. This conditional will be simulated using G_3 statements. We do this to avoid passing an actual parameter of type real (corresponding to 0) when the procedure is expecting an array (corresponding to values > 0) in the Pascal simulation.

G_3 program	Pascal program simulating G_3 program
1. Let $x_1, x_2, \ldots, x_n$ be the G_3 program variables.	procedure $P_0(a_1{:}\langle x_1\rangle; a_2{:}\langle x_2\rangle; ..; a_n{:}\langle x_n\rangle)$; simulate statement 1
2. $k{:}x_i{:}{=}x_i{+}1$ where k is the statement label.	var b:array[1..15] of real; $P_k(a_1, a_2, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_n)$; where P_k is defined as procedure $P_k(a_1{:}\langle x_1\rangle; a_2{:}\langle x_2\rangle; ..; a_n{:}\langle x_n\rangle$; simulate statement k+1
3. $k{:}x_i{:}{=}x_i{-}1$	$P_k(a_1, a_2, \ldots, a_n)$; where P_k is defined as procedure $P_k(a_1{:}\langle x_1\rangle; a_2{:}\langle x_2\rangle; ..; a_{i-1}{:}\langle x_{i-1}\rangle$; b:array[*] of $\langle x_i\rangle$; $a_{i+1}{:}\langle x_{i+1}\rangle; ..; a_n{:}\langle x_n\rangle$; var $a_i{:}x_i$; simulate statement k+1
4. k:if $x_i \neq 0$ then L	if $x_i \neq$ real then $P_L(a_1, a_2, .., a_n)$; simulate statement k+1.

G_3 program	Pascal program simulating G_3 program
5. Empty statement (last statement)	Empty statement

The main program P is simply

```
program P;
  var a1, a2, a3,..., an:real;
  P0(a1,a2,...,an)
```

Assume algorithm A exists. Now if algorithm A tells us that program P and all the related procedures lead to infinite mode generation, then there exists at least one variable x_i in the corresponding G_3 program which is going to have an infinite value $\Rightarrow$ the G_3 program will not halt.

On the other hand suppose A tells us that there will not be any infinite mode generation $\Rightarrow$ there is no variable in the G_3 program that will acquire an infinite value $\Rightarrow$ the G_3 program halts. (note if the G_3 program does not halt, then the clock variable will acquire an infinite value) $\Rightarrow$ HP $\Rightarrow$ solved $\Rightarrow$ our assumption about the existence of A was incorrect.

Note $\Rightarrow$ means implies

The infinite type problem arises in several different cases; in essence this is due to a particular parameter of a recursive procedure being called with either

i) an array of higher and higher nesting (i.e. array[1] of array[1] of . .),

ii) records where types are of higher and higher complexity,

iii) subrange types of higher and higher cardinality (e.g. 1..3, 1..4, 1..5, ..)

iv) combinations of the above.

Some of these problems can be resolved by considering similar types to be equivalent - treating all subrange types using a dope vector, and treating arrays of different dimensions but the same element type as the same, using a dope vector. The other problems can be avoided by

a) restricting the use of the 'variable' types (such as not allowing a variable type z to be used in constructing a structured type array[1..10] of z or record i,r:z end) and

b) having the compiler recognize the situations that might lead to infinite mode generation and postponing processing to sum time. i.e. interpretation.

9. Formal Definition of Processing a Call to a Procedure that Accepts Parameters of Different Types.

The effect of processing a call to a procedure G with actual parameters $a_1,a_2,\ldots,a_n$ is described as

$$G(a_1,a_2,\ldots,a_n)\ \{Compiler(G,a_1,a_2,\ldots,a_n)\}$$

$$G^{[p_1,a_1],\ldots,[p_n,a_n]}(a_1,a_2,\ldots,a_n)$$

where the p_i s are the formal parameters of G. The call $G(a_1,a_2,\ldots,a_n)$, as a consequence of the processing by the compiler is replaced by the call

$$G^{[p_1,a_1],[p_2,a_2],\ldots,[p_n,a_n]}(a_1,a_2,\ldots,a_n).$$

The procedure $G^{[p_1,a_1],[p_2,a_2],\ldots,[p_n,a_n]}$ is the procedure G modified by the following type reduction algorithm.

```
for i:=1 to n do
  begin ES:=[type(p_i) = type(a_i)];
    while  equation e in ES not completely reduced do
           ES:=ES + [equations e reduces to];
    for each equation e in ES do
           Textually replace the identifier in the
           angle brackets on the left hand side of e
           by its right hand side in G;
    end
```

<u>Note</u> type(x) returns the type of x

Intutively, type reduction is finding the type corresponding to the generic type identifiers by pattern matching. Reduction rules for type equations follow:

Reduction rules for type equations

equation e	equations to which e reduces
1. set of x = set of y	x = y
2. file of x = file of y	x = y
3. array[a] of x = array[b] of y where a,b ='*' or p..q if a and b are both not = '*' then the bounds p and q must be the same on both sides	x = y
4. record $s_1:x_1;s_2:x_2;..;s_n:x_n$ = record $t_1:y_1;t_2:y_2;..;$end $t_n:y_n$ end	$x_i = y_i \;\; 1 \le i \le n$
5. <y> = x	completely reduced
6. identifier_1 = identifier_2	(i.e. empty)

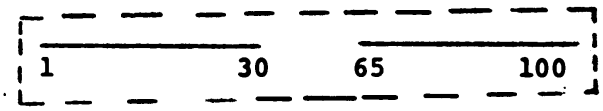
If a type equation is not in one of the above forms then there is an error.

CHAPTER IV: GRIDS

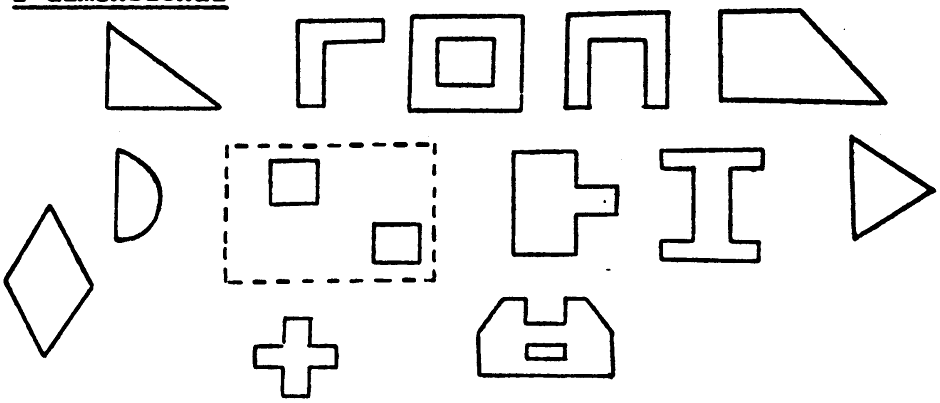
1. Introduction.

Most objects in the real world do not have array like shapes. Consequently, in many computer applications like engineering, numerical analysis, inventory control, data processing etc., problems arise that require only a portion of the elements of an array. Some examples of these 'partial' arrays are given below, where the dashed boxes indicate that the enclosed data structures are components of the same data structure even though they are not connected to one another.

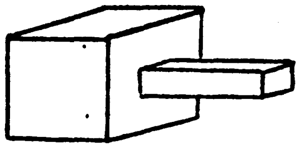
1-dimensional

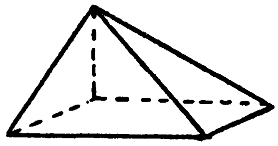


2-dimensional



3-dimensional





We introduce a new unpacked structured type called the grid to represent the 'partial arrays'. A grid is an array that can have any shape. Conventional arrays are a particular case of grid shapes and will eventually be replaced by grids in programming languages. We will show that the use of grids makes programs shorter, easier to understand, more general and easier to prove correct.

In current programming languages, grids are simulated using one of the methods described below, making programs not only harder to read and understand but also more complex.

a. Each grid is subdivided into portions that can be represented by arrays, Each portion is declared to be an array, each with a different name, and they collectively represent the grid. This method has the following disadvantages: (i) The grid cannot be collectively referred to by one name. (ii) Extra programming effort is required to make sure that the right array name is used when referring to an element. (iii) Grid aggregate operations are not possible.
(iv) There is no flexibility; slight changes in a grid shape often result in extensive

changes in the statements which process them.

b. A one to one function is used to map the grid onto a linear array. The disadvantages of this method are: (i) A simple function may not exist or may be hard to find. (ii) Function evaluation cost may be prohibitive. (iii) Grid aggregate operations are not possible. (iv) Explicit programming to check for invalid grid indices may be required. (v) A change in the grid shape might involve finding a new function and changing program statements.

c. The grid is represented by the smallest encompassing array (SEA); non-grid points could be marked by some special values in the SEA. The disadvantages of this method are: (i) Storage may be wasted. (ii) When iterating over the SEA, explicit programming to avoid non-grid points is required. (iii) Grid agregate operations (in this case array aggregate operations) are not possible in general since the special values in the SEA may be destroyed.

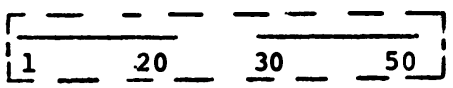
Apart from the disadvantages mentioned above for each representation, portions of grids, for example the boundary points, cannot be collectively referred to.

2. Examples and syntax of grid declarations

We shall try to develop a syntax for grid declarations that reflects the shape of a grid and conforms with the syntax of Pascal. Portions of grids (e.g. the boundary points, or the points inside the boundary) may be given a name for use in the for statement and in aggregate operations. The grid is a structured type similar to an array and may be used just like array types for declaring variables of type grid and in the definition of new types. An element of an n-dimensional grid is referred to exactly like an n-dimensional array element.

The syntax of grids is illustrated by the examples given below. In these examples we often give alternate ways of defining the same grid shape.

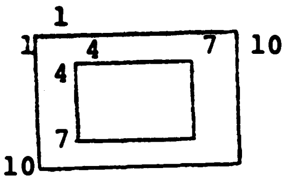
2a. One dimensional grids

1. 

i. grid[1..20; 30..50] of real

ii. grid[1..50; minus 21..29] of real

2b. Two dimensional grids

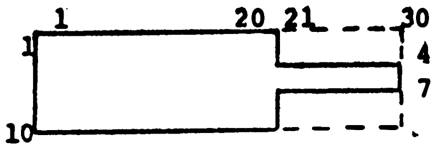


i. grid[1..4,1..10; 7..10,1..10;
5:6,1:4; 5:6,7:10] of
integer

ii. grid[1..10,1..4;1..10,7..10;
1..4,5..6; 7..10,5..6] of
integer

iii. <u>grid</u>[1..10,1..10; <u>minus</u> 5..6, 5..6] <u>of</u> integer

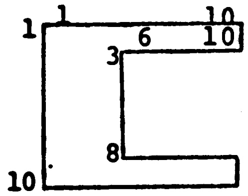
2.



i. <u>grid</u>[1..10,1..10; 4..7,21..30] <u>of</u> complex

ii. <u>grid</u>[1..10,1..30; <u>minus</u> 1..3,21..30; <u>minus</u> 8..10, 21..30] <u>of</u> complex

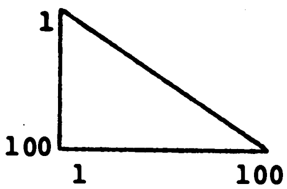
3.



i. <u>grid</u>[1..3,1..10; 8..10,1..10; 4..7,1..3] <u>of</u> char

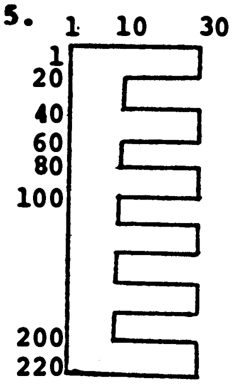
ii. <u>grid</u>[1..10,1..10; <u>minus</u> 4..7,7..10] <u>of</u> char

4.



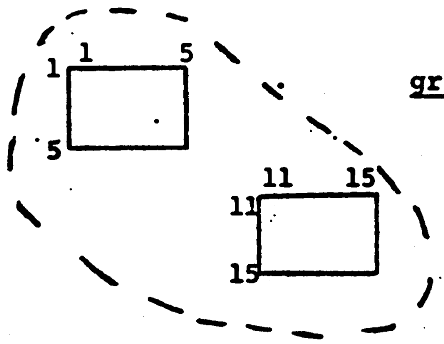
<u>grid</u>[1..1,1..1;2..2,1..2; 3..3,1..3; .. 100..100,1..100] <u>of</u> real

5.



<u>grid</u>[1..220,1..10; 1..20,1..30;
41..60,1..30;
.
.
.
201..220,1..30]
<u>of</u> real

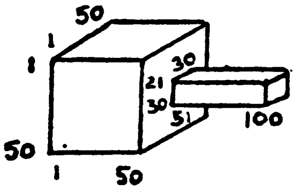
6.



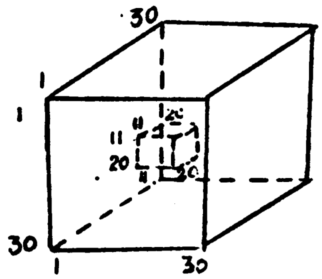
<u>grid</u>[1..5,1..5;11..15,11..15] <u>of</u>
real

2c. <u>Three dimensional grids</u>

1.



<u>grid</u>[1..50,1..50,1..50;
21..30,51..100,21..30]
<u>of</u> complex

2. 

grid[1..30,1..30,1..30; *minus* 11..20,11..20, 11..20] *of* complex

Notes

1. The *minus* feature makes it easier to describe grids of certain shapes.
2. For examples 2b.4 and 2b.5 our notation becomes cumbersome. In example 2b.4 the triangular grid has to be declared as consisting of a hundred linear 2-dimensional arrays.

3. Functional notation for grid bounds

As we saw in the last section, the notation for grids becomes clumsy and long for some grid shapes. Allowing the bounds of a grid dimension to be functions of the coordinates of other dimensions alleviates this problem. In other words, we allow the upper and lower bounds of the i-th dimension to be functions of the coordinates of the 1,2,..,i-1,i+1,..n dimensions in a n-dimensional grid. If the i-th dimension bounds are functions of the coordinates of the j-th dimension ($i \neq j$) then the j-th dimension bounds cannot be functions of the coordinates of the i-th dimension to avoid circularity.

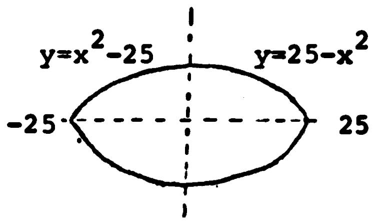
i. The triangular grid of example 2.b.4 can now be be defined as follows; the upper bound of the second dimension is now a function of the coordinate of the first dimension (the identity function).

grid[i in 1..100,1..i] of real

ii. The multipronged E of example 2.b.5 can now be declared as

grid[j in 1..220,1..30+(-1)**((j-1)div 20)*10] of real

iii. The shape



is declared as

grid[i in -5..5,i*i-25..25-i*i] of integer

One way of avoiding the circularity problem is to restrict the bounds of the i-th dimension to be functions of the coordinates of the 1,2,..,i-1 dimensions only (this reduces the number of possible shpaes but we can simulate all shapes by interchanging the coordinates). The bounds of a functional grid declaration would then look like

$$i_1 \ \underline{\text{in}}\ L_1..U_1,\ i_2\ \underline{\text{in}}\ L_2(i_1)..U_2(i_1),\ldots,$$
$$i_{n-1}\ \underline{\text{in}}\ L_{n-1}(i_1,i_2,\ldots,i_{n-2})..U_{n-1}(i_1,i_2,\ldots,i_{n-2}),$$

$$L_n(i_1,i_2,\ldots,i_{n-1})\,..\,U_n(i_1,i_2,\ldots,i_{n-1})$$

4. The partial grid syntax

To aid the reader in understanding grids better, we now give the formal grid syntax. It is incomplete as the concepts discussed in later sections are not included. The complete syntax is given in appendix B.

```
<unpacked structured type>::=<array type>|<record type>
                    <set type>|<file type>|<grid type>
<grid type>::= grid[<regular part> {i<regular part>}]
                              of  component type
<regular part>::= <positive part>|{ minus }1<piece>
<piece>::= <bounds>  {,<bounds>}
<positive part>::= {<identifier>}  [<piece>{i{minus}1
                                      <piece>}]
<bounds>::= {<identifier> in}1  <simple bounds>
<simple bounds>::= <expression>..<expression>
```

We have used the notation <u>{A}</u> to mean that A may be included as many times as desired and $\underline{\{A\}}^1$ to mean that A may be included at most once. The grid component type is the same as the Pascal array component type.

5. Generating the indices of the grid elements

The utility of the data structure grid would be severely limited if no simple way of generating the indices of all or parts of the grid was provided. The general <u>for</u> statement introduced earlier is used for this purpose.

The grid, like an array, may be thought of as a function whose domain is the set of all valid subscripts of its elements.

Let g be a n-dimensional grid that consists of $j (j>0)$ n-dimensional pieces $p_1, p_2, \ldots, p_j$ (the positive pieces) minus the union of $k (k \geq 0)$ n-dimensional pieces $q_1, q_2, \ldots, q_k$ (the negative pieces). The index types of the i-th dimension of all the pieces must either be the same scalar type or subranges of the same scalar type. The domain of grid g is defined as:

$$\underline{\text{domain}}(g) = (\underline{\text{domain}}(p_1) \vee \underline{\text{domain}}(p_2) \vee \ldots \vee \underline{\text{domain}}(p_j)) - (\underline{\text{domain}}(q_1) \vee \underline{\text{domain}}(q_2) \vee \ldots \vee \underline{\text{domain}}(q_k))$$

The pieces are either array like or have functional bounds. The domain of an array has been defined before while the domain of a piece with functional bounds is defined as follows.

Let z be a component of the form

$$i_1 \ \underline{\text{in}} \ a..b, \ i_2 \ \underline{\text{in}} \ f_2(i_1)..g_2(i_1), \ldots, i_n \ \underline{\text{in}} \ f_n(i_1, i_2, \ldots i_{n-1})..g_n(i_1, i_2, \ldots, i_{n-1})$$

Then the domain of z is the set of tuples

$$\text{domain}(z) = [\langle z_1, z_2, \ldots, z_n \rangle \mid a \leq z_1 \leq b, \ f_2(z_1) \leq z_2 \leq g_2(z_1), \ f_3(z_1, z_2) \leq z_3 \leq g_3(z_1, z_2), \ldots, \ f_n(z_1, z_2, \ldots, z_{n-1}) \leq z_n \leq g_n(z_1, z_2, \ldots, z_{n-1})]$$

Note that from the definition of the domain of a grid we see that

i) The tuple $<i_1,i_2,..,i_n>$ representing the indices of the grid element $g[i_1,i_2,..,i_n]$ may belong to the domain of more than one piece of the grid g.

ii) If the tuple $<i_1,i_2,..,i_n>$ belongs to the domain of at least one piece being minused out, then the grid g has no element with subscripts $i_1,i_2,..,i_n$.

We define a new type called the domain of a grid (similar to the domain of an array). It is equivalent to the record type $t_1,t_2,..,t_n$ where the index types of the i-th dimension of the grid components are the same as or subranges of t_i, with the restriction that the set of values represented by the record type is the same as that belonging to the domain of the grid. The domain of a n-dimensional grid g, domain(g), may be used in the <u>for</u> statement to specify a set of values. The ordering operators <u>row</u>, <u>col</u> and <u>rev</u> may be used to order <u>domain</u>(g) in a fashion similar to that for record types as discussed before. The corresponding index variable must be

i) a variable of type record with n components of type t_i $1 \leq i \leq n$

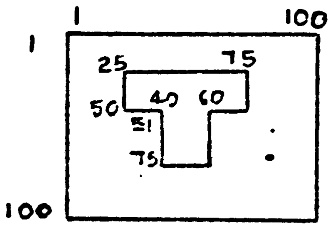
ii) a list of n variables of type t_i $1 \leq i \leq n$

where the index type of each component of the grid is t_i or a subrange of it $1 \leq i \leq n$.

Examples

The following program segments visit the grid points in increasing row order. In case of Pascal without the data structure gird the SEA is used to represent grids.

a.



Pascal

```
var a:array[1..100] of real; i,j:integer;
for i:=1 to 100 do
  for j:=1 to 100 do
    if not(i>25 and i<50 and j>25 and j<75 or
           i≥50 and i<75 and j>40 and j<60)
                            then visit(a[i,j]);
```

Pascal/g (Pascal with grids)

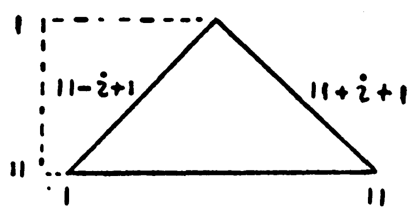
```
var g:grid[1..100,1..100; minus 26..49,26..74;
                          minus 50..74,41..59] of real;
                          i,j:integer;
```

We give two alternate program segments.

```
i)  var i:domain(g);
    for i in row domain(g) do visit(g[i]);

ii) var j,k:integer;
    for j,k in row domain(g) do visit(g[j,k]);
```

b.



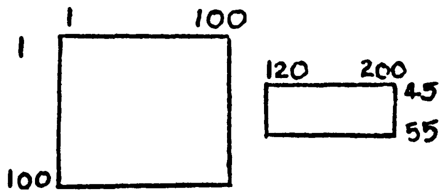
Pascal

```
var a:array[1..21,1..21] of real; i,j..integer;
for i:=1 to 11 do
  for j:=11-i+1 to 11+i-1 do visit[a[i,j]);
```

Pascal/g

```
var g:grid[i in 1..11,11-i+1..11+i+1] of real; i,j:integer;
for i,j in row domain(g) do visit(g[i,j]);
```

c.



Pascal

```
var a:array[1..100,1..200] of real; i,j:integer;
i.  for i:=1 to 100 do
      for j:=1 to 200 do
        if j<100 or i>45 and i<55 and j>120
                              then visit(a[i,j]);
```

or the following more efficient alternative.

```
ii. for i:=1 to 44 do
      for j:=1 to 100 do visit[a[i,j]);
    for i:=45 to 55 do
      for j:=1 to 200 do
        if j<100 or j>120 then visit[a[i,j]];
    for i:=56 to 100 do
      for j:=1 to 100 do visit(a[i,j]);
```

Pascal/g

```
var g:grid[1..100,1..100; 45..55,120..200] of real;
                                    i,j:integer;
for i,j in row domain(g) do visit(g[i,j]);
```

An analysis of the above examples reveals the following points in favour of grids:

1. The grid declarations, unlike the array declarations, correspond to the shape being represented.
2. i. Pascal/g segments are shorter than the corresponding Pascal segments.
   ii. Pascal/g program segments are easier to understand.
   iii. There are more chances of making errors in the Pascal segments because we have to explicitly make sure that illegal grid indices are not generated.
   iv. In Pascal, the user has to decide which iteration method will be more efficient (example c). In Pascal/g this is left to the compiler.
3. A change in the order of visiting the grid points, say from increasing row order ot decreasing row order, requires only minor modifications to the grid iteration statement while in Pascal it may involve changing the <u>for</u> statement limits and the sequential order of the statements. If we do not care about the order of

iteration we can omit some key words in the grid iteration statement allowing the language processor the freedom to perform the iteration in the most efficient order. In Pascal we must specify some iteration order.

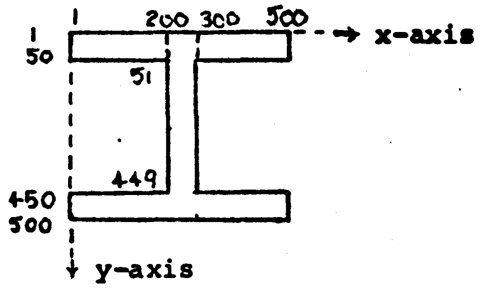
4. Most important is the fact that the Pascal/g program segments are identical for all the different shapes; this is not the case for the Pascal segments. The ramification of this is that a Pascal/g program can be altered to work for another grid shape by just altering the grid declaration, while the rest of the program remains intact.

5. We would like to use the most efficient representation for grids. In Pascal the programmer has to figure out which grid representation will be the most efficient while in Pascal/g an intelligent compiler will do it for the programmer.

The following is a complete practical problem involving the use of grids.

Calculate the moment of inertia (MI) of the unit I-beam, shown below, about the y-axis. The discrete weight at coordinate i,j is given by $m_{i,j}$ and the x-coordinate by $x_{i,j}$. The MI is given by the formula

$$MI = \sum_{ij}\sum m_{i,j} * x_{i,j} ** 2$$



In the programs given below storage of the weights $m_{i,j}$ could have been avoided.

Pascal

```
var mi:real; i,j:integer; m:array[1..500,1..500] of real;
{read in data}

        for i:=1 to 50 do
          for j:=1 to 500 do read(m[i,j]);
        for i:=51 to 449 do
          for j:=200 to 300 do read(m[i,j]);
        for i:=450 to 500 do
          for j:=1 to 500 do read(m[i,j]);

{compute mi}

      mi = 0;
      for i:=1 to 50 do
        for j:=1 to 500 do mi:=mi+m[i,j]*(j-1)**2;
      for i:=51 do 449 do
        for j:=200 to 300 do mi:=mi+m[i,j]*(j-1)**2;
      for i:=450 to 500 do
        for j:=1 to 500 do mi:=mi+m[i,j]*(j-1)**2;
              write(mi);
```

Pascal/g

```
var mi:real; i,j:integer; m:grid[1..50,1..500; 51..449,
                                 200..300; 450..500,
                                 1..500] of real

{read in data}

  for i,j in row domain(m) do read(m[i,j]);
{compute mi}

  mi:=0;
  for i,j in domain(m) do mi:=mi+m[i,j]*(j-1)**2;
write(mi);
```

6. Parts and subgrids

6a. Parts.

As mentioned before, grids may be thought of as consisting of a union of pieces (the positive pieces) minus the union of some other pieces (the negative piece). A part of a grid is defined to be the union of m pieces minus the union of n other pieces belonging to the grid ($m \geq 0$, $n \geq 0$). A positive part has at least one positive piece. We allow the naming of positive parts to enable easy traversal of portions of the grid. For example, one might want to be able to visit the boundary points of a grid or just the inner region points. The part names must be qualified by the grid name whenever there is any ambiguity as follows:

<grid name> . <part name>

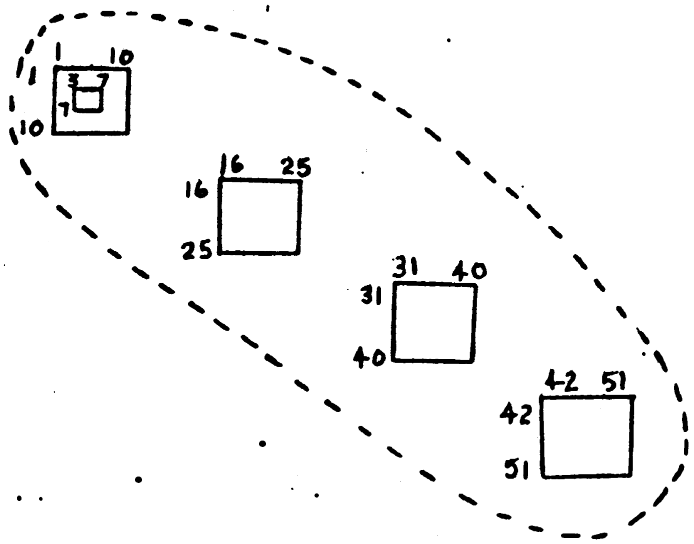
The domain of a part of grid g is the same as that of a grid comprising of just that part and all the negative parts of g.

The domain of a union of m grids or parts minus the union of n grids or parts is a finite set of values and therefore we can consider it to be a type. Let $a_1, a_2, \ldots, a_m, b_1, b_2, \ldots, b_m$ be grids or named parts. Then

$$\underline{\text{domain}}(a_1, a_2, \ldots, a_m, \underline{\text{minus}}\ b_1, \underline{\text{minus}}\ b_2, \ldots, \underline{\text{minus}}\ b_m)$$
$$= \underline{\text{domain}}(a_1) \vee \underline{\text{domain}}(a_2) \vee \ldots \vee \underline{\text{domain}}(a_m) - \underline{\text{domain}}(b_1)$$

$- \underline{\text{domain}}(b_2) - .. - \underline{\text{domain}}(b_n)$

Example



The above grid g with three named parts a, b and c and one unnamed part is declared as follows:

<u>var</u> g:<u>grid</u>[a[1..10,1..10; <u>minus</u> 4..6,4..6]; b[16..25,16..25];
c[31..40,31..40]; 42..51,42..51] <u>of</u> real;

Let i and j be integer variables and s be any statement.

Some examples of the use of parts in the <u>for</u> statement are

i) <u>for</u> i,j <u>in</u> <u>domain</u>(g.a) <u>do</u> s
ii) <u>for</u> i,j <u>in</u> <u>rev</u> <u>col</u> <u>domain</u>(g.a,g.b) <u>do</u> s
iii) <u>for</u> i,j <u>in</u> <u>rev</u> <u>row</u> <u>domain</u>(g, <u>minus</u> g.a, <u>minus</u> g.c) <u>do</u> s

Some examples of the improper use of part names in the <u>for</u> statement are:

i) <u>for</u> i,j <u>in</u> <u>domain</u>(<u>minus</u> g.a) <u>do</u> s

ii) <u>for</u> i,j <u>in</u> <u>domain</u>(g.a, <u>minus</u> g) <u>do</u> s

Both the above statements are null statements. s is never executed,

The naming of non-positive i.e. negative parts has not been allowed for the following reasons:

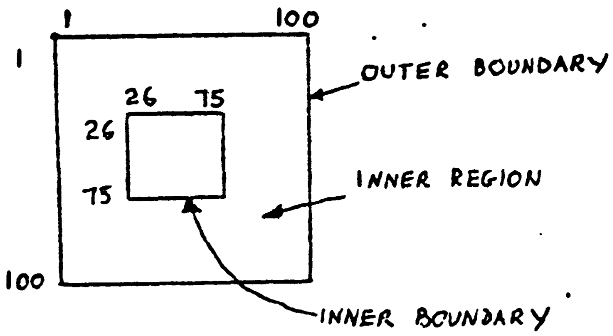
1. It does not make any sense to be able to refer to or iterate over elements that do not belong to the grid.
2. The grid iteration statements do not generate the subscripts of the elements that belong to the negative parts. Hence the negative parts do not have to be explicitly subracted out of the positive parts in the grid iteration statement.

An element of a grid may be referred to by using the part name. In the above example g[1,1] is the same as a[1,1] (if unambiguous) or g.a[1,1].

b. <u>Subgrids</u>.

Ability to name parts of a grid allows us to traverse their elements. We would like a way of traversing portions of parts composing a grid. One solution to this would be to break up the parts into smaller pieces, name each piece, and then use them in various combinations. However, semantically it may not be good programming practice to break up parts. We allow the naming portions of parts by the definition of subgrids:

Example



The above grid with its inner and outer boundaries and the inside region defined as subgrids could be declared as follows:

```
var g:grid[1..100,1..100; minus 27..74,27..74] with subgrids
          outer_boundary[1..100,1..100; minus 2..99,2..99],
          inner_boundary[26..75,26..75; minus 27..74,27..74]
          inside_region[2..99,2..99; minus 26..75,26..75]
                                                  of real;
```

Subgrids can be used in a fashion similar to parts. In case of ambiguity the subgrid name must be qualified by the grid name as shown below:

<grid name> . <subgrid name>

Subgrids are different from parts in that they do not cause allocation of storage. The subgrid elements must be a subset of the corresponding grid elements.

The domain of a subgrid is similar to the domain of a part. Subgrids are used just like parts. Some examples of their useage are:

```
var k:domain(g:inside_region); i,j:integer;
i)   for i,j in col domain(inner_boundary] do s
ii)  for i,j in domain(g, minus g.inside region] do s
iii) for k in domain(g.inside_region) do s
```

<u>Note</u>

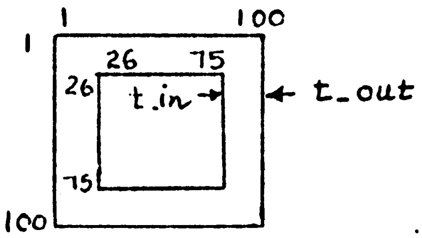
As an extension of subgrids, the use of the grid name and of existing part and subgrid names could be allowed in the definition of additional subgrids. For example, the subgrid inside_region may be defined as

inside_region[g, <u>minus</u> inner_boundary, <u>minus</u> outer_boundary]

<u>Example</u>

The following problem is taken from Heat Transfer (Mechanical Engineering).

Find the steady state temperature distribution across the walls of the duct (or chimney) shown below, the inside wall temperature being t_in and the outside wall temperature being t_out.



A heat transfer analysis of the above problem reduces it to the simpler problem of solving the Laplace equation $\nabla^2 t=0$ (t is the temperature) over the same cross-section. The method used to solve the Laplace equation is Jacobi iteration. It is defined as follows:

```
guess a temperature distribution for the cross-section
repeat
    calculate the new temperature distribution
    from the old one by letting the new
    temperature of each point be the average of
    the old temperatures of the surrounding
    points;
    until (the maximum change in the temperature
           of all the points is < epsilon).
```

where epsilon is << 1.0 and is the desired accuracy of the result.

<u>Pascal</u>

In the Pascal program, we use a boolean array region initialised as follows:

$$\text{region}[i,j] = \begin{cases} \underline{\text{true}} \text{ if the point } i,j \text{ is inside the duct boundaries} \\ \underline{\text{false}} \text{ if the point } i,j \text{ is on the duct boundaries or outside the duct.} \end{cases}$$

```
var t_in,t_old,eps,maxchange:real; i,j:integer;
    t_new,t_old:array[1..100,1..100] of real;
   region:array[2..99,2..99] of boolean;

initialise t_in,t_out,eps;
{initialise region}
   for i:=2 to 99 do
     for j:=2 to 99 do if i≤25 or i≥76 or j≤25 or j≥76
                               then region[i,j]:=true;
                               else region[i,j]:=false;

{initialise t_old}
   for i:=1 to 100 do
     for j:=1 to 100 do
       if i=1 or i=100 or j=1 or j=100
         then t_old[i,j] := t_out;
         else if region[i,j]
             then t_old[i,j]:=(t_in+t_out)/2;
             else t_old[i,j]:=t_in;

repeat
   maxchange:=0;
   {calculate new temperature distribution}
      for i:=2 to 99 do
        for j:=2 to 99 do
          if region[i,j]
            then begin t_new[i,j]:=(t_old[i,j-1]+
                                t_old[i,j+1]+t_old[i+1,j]+
                                t_old[i-1,j])/4.0;
                       maxchange:=max(maxchange,abs(
                                  t_old[i,j]-t_new[i,j]));
                       end
```

```
        for i:=2 to 99 do
          for j:=2 to 99 do if region[i,j] then
                              t_old[i,j]:=t_new[i,j];
        until(maxchange<eps);
    print t_old;
```

Pascal/g

```
    var t_in,t_out,eps,maxchange:real; i,j:integer;
        t_new,t_old:grid[1..100,1..100; minus 27..74,27..74]
                                               with subgrids
                inner_boundary[26..75,26..75;minus
                                               27..74,27..74],
                outer_boundary[1..100,1..100;minus
                                               2..99,2..99],
                inside_region[2..99,2..99; minus
                                               26..75,26..75]
                                                     of real;

    initialize t_in,t_out,eps;
    {initialize t_old}
      for i,j in domain(t_old.outer_boundary) do
          t_old[i,j]:=t_out;
      for i,j in domain(t_old;inner_boundary) do
          t_old[i,j]:=t_in;
      for i,j in domain(t_old.inside_region) do
          t_old[i,j]:=(t_in+t_out)/2;

    repeat
      maxchange:=0;
      {calculate new temperature distribution}
        for i,j in row domain(t_old.inside_region) do
          begin t_new[i,j]:=(t_old[i,j-1]+t_old[i,j+1]+
                            t_old[i-1,j]+t_old[i+1,j])/4.0;
              maxchange:=max(maxchange,abs(t_old[i,j]-
                                              t_new[i,j]));
```

```
        end
for i,j in domain(t_old.inside_region) do
     t_old[i,j]:=t_new[i,j];
print t_old;
until(maxchange<eps);
```

Notes

1. If the symmetry of the duct is taken into account, then the whole temperature distribution can be contained by just considering 1/8th of the duct.
2. The Pascal/g program holds for any cross-sectional shape provided just the grid declaration is changed accordingly.
3. Since we never use t_new's subgrids and we could have declared it without subgrids. However, for convenience's sake we declare it along with t_old.

7. Aggregate Operations

7a. The grid aggregate assignment statement

The grid aggregate assignment statement (corresponding to the array aggregate assignment statement in PL/I) allows the grid elements to be assigned values resulting from similar operations. The left hand side of the grid aggregate statement can be a grid name, part name or subgrid name. The right hand side may be an expression involving the following:

1. grids, parts or subgrids.
2. simple variables
3. constants

4. functions returning grids or values of type corresponding to elements of the aggregate on the left hand side.

The shape of all the grids, parts and subgrids in the assignment statement must be the same and the bounds identical.

The grid aggregate statement reduces the amount of coding to be done making the programs smaller, easier to read and prove correct.

<u>Examples</u> (using the duct cross-section of section 6.b)

```
1.  t_old.inner_region:=5;
2.  t_new:=t_old+5;
3.  t_new:=t_in;
```

If we had used grid aggregate assignment statements then the imperative part of our last example (the steady state temperature one) would look like

```
initialize t_in,t_old,eps;
{initialize t_old}
   t_old.outer_boundary:=t_out;
   t_old.inner_boundary:=t_in;
   t_old.inside_region:=(t_in+t_out)/2;
repeat
   maxchange:=0;
   {calculate new temperature distribution}
      for i,j in row domain(t_old.inside_region) do
        begin t_new[i,j]:=(t_old[i,j-1]+t_old[i,j+1]+
                          t_old[i-1,j]+t_old[i+1,j]/4.0;
```

```
                  maxchange:=max(maxchange,abs(t_old[i,j]-
                                              t_new[i,j]));
                  end
        t_old:=t_new;
        print t_old;
        until(maxchange<eps);
```

7b. Grid sections

Let i. g be a n-dimensional grid, subgird or part.

ii. s_k, t_k $1 \le k \le n$ be integer valued expressions such that $s_k \le t_k$

Then the grid section $g[s_1..t_1, s_2..t_2, ..., s_n..t_n]$, a grid itself, refers to those elements of g for which valid$(g, i_1, i_2, ..., i_n)$ $s_k \le i_k \le t_k$ $1 \le k \le n$ is true.

If any $s_k..t_k$ is replaced by a '*' then default values of $s_k =$ ℓbound$(g, p_1, p_2, ..., p_{k-1}, *, p_{k+1}, ..., p_n)$ and $t_k =$ ubound$(g, p_1, p_2, ..., p_{k-1}, *, p_{k+1}, ..., p_n)$ are assumed, where the p_i's are equal to '*' 's, $1 \le i \le k-1$, $k+1 \le i \le n$.

The grid sections may be used in the grid aggregate assignment statements.

Examples

We will again use the duct of the steady state temperature problem to illustrate the use of grid sections.

1. t_new[1,*]:=t_out
   This assigns the elements of the upper part of the outer boundary the value t_out.

2. t_new.outer_boundary[1,*]:=t_out

and
t_new[1,1..100]:=t_out
are identical to example 1.

3. t_old[*,*] is the same as t_old[1..100,1..100]. Both refer to the whole grid t_old.

4. t_old[50,*] refers to the elements t_old[50,j] where 1≤j≤26, 75≤j≤100

5. t_new.inner_boundary[50,*] refers to t_new[50,26] and t_new[50,75].

6. t_new[*,1]:=t_old[*,1] assigns the left hand boundary elements of t_new the values of the corresponding elements of t_old.

__Note__ Even though t_new[1,*] and t_new.outer_boundary[1,*] refer to the same grid elements, the use of t_new.outer_boundary is more indicative of the indended meaning.

The grid iteration statement can be used to iterate over a grid section.

__Examples__

1. __for__ i,j __in__ __row__ __domain__(g3[3,*,*] __do__ visit(g3[3,i,j]);
2. __for__ i __in__ __domain__(t_old[50,*]) __do__ visit(t_old[50,i]);
3. __for__ i,j __in__ __domain__(t_old[1..4,1..4]) __do__ S

Instead of a single section we can visit a union $m \geq 1$ section minus the union of $n \geq 0$ sections.

For example

```
for i in col domain((t_new,minus t_new,outer_boundary,
          minus t_new.inner_boundary)(50,*)) do
                              visit(t_new[50,i]);
```

8. Built-In Functions

Three built-in functions are provided for grids. They are

8a. Valid

Let i. g be a n-dimensional grid, subgrid or part name.

ii. $i_1, i_2, \ldots, i_n$ be integer valued expressions

Then valid$(g, i_1, i_2, \ldots, i_n)$ returns <u>true</u> if $g[i_1, i_2, \ldots, i_n]$ is a legal element of g (i.e. $\langle i_1, i_2, \ldots, i_n \rangle$ belongs to <u>domain</u>(g)) otherwise <u>false</u>.

<u>Example</u>

Suppose we have a 2-dimensional grid named wall with subgrids window and door. Then valid could be used as follows:

```
if valid(wall.door,i,j)
   then process door point i,j;
   else if valid(wall.window,i,j)
          then process window point i,j;
          else process wall point i,j;
```

8b. <u>ℓbound</u>

The function ℓbound introduced before applies to grids also. Let

i) g be an n-dimensional grid or grid expression
ii) i be an integer expression
iii) j_k, $1 \le k \le n$, be expressions such that the type of j_k matches the index type of the kth dimension of g.

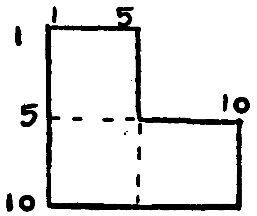
Then

1) ℓbound(g,i) returns the lower limit of the i^{th} dimension of the grid

2) ℓbound(g) returns a record with n components comp1,comp2,..,compn such that ℓbound(g).compi is equal to ℓbound(g,i)

Since all sections of a grid do not have the same lower limits as the grid itself, ℓbound(g[j_1,j_2,...,j_{m-1},*,j_{m+1},...; i_n]) may not be equal to ℓbound(g,m).

<u>Example</u>

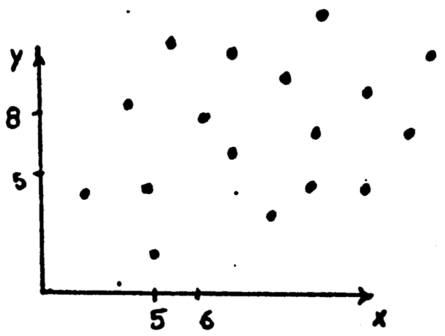


<u>var</u> a:<u>grid</u>[1..10,1..5; 5..10,6..10] <u>of</u> integer;
ℓbound(g,1) = 1 and ℓbound(g[*,10]) = 5

8c. ubound

Same as ℓbound except that the upper limit is returned.

9. Some Examples



city of Zend
Fire station locations

The above is a sketch of the fire companies located in the city of Zend. Write a program to find the fire company nearest to a fir at point (k,m) and print the number of engines available there.

```
var fire:grid[5..5,5..5; 6..6,8..8;......] of integer;
                    k,m,i,j,mini,minj:integer;

initialize the grid fire to the number of engines
  available at each company;
initialize k,m;
 {find the nearest company}
  mini:=0; minj:=0;
  for i,j in domain(fire) do
     if (k-mini)**2+(m-minj)**2>(k-i)**2+(m-j)**2
          then begin mini:=i; minj:=j;  end
  write (mini,minj,fire[mini,minj]);
```

b. Inventory control example

Information about the quantities of motorcycle, can and truck part components is to be stored.

	part numbers	number of components of each part
motorcycle	100-115	10
car	200-235	25
truck	680-730	32

The information will be stored in the following grid:

```
var vehicles:grid[motor_cycle[100..115,1..10];
                             car[200..235,1..25];
                             truck[680..730,1..32] of
                                                  integer;
```

Write program segments to

a. print the quantities of the components of part x that are in stock.

b. Update all the components of the truck parts.

c. Print all parts with more than 10 components.

The program segments can easily be written using the general for statement as follows:

```
a. for i in row domain(vehicles[x,*]) do
                              write(vehicles[x,i])

b. for i,j in row domain(vehicles.truck) do
                              update(vehicles[i,j]);

c. for i in row domain(vehicles[*,11]) do write(i)
```

9c. Data processing example

The names and addresses of telephone subscribers in Hudson county, N.J. are to be stored so as to facilitate easy access.

first 3 digits	last 4 digits
659	0..9999
792	0..9999
798	0..9999

Write program segments to

1. Print the names and addresses of persons whose telephones start with the number 792
2. Print the names of people whose telephone number does not begin with 792

The telephone numbers will be stored in a string grid declared as follows:

```
type string = packed array[1..40] of char;
var tel:grid[a[659,0..9999]; b[792,0..999];
             c[798,0..9999]] of string;
```

The above requires a lot of storage and using it would probably require paging.

We assume that unused telephone numbers have been assigned a string of blanks.

```
1. for i in row domain(tel.a[792,*] do
       if tel[792,i] ≠ bs then write(tel[792,i]);
2. for i,j in row domain(tel,minus tel.a) do
       if tel[i,j] ≠ bs then write(tel[i,j]);
```

where bs is a variable of type string that has been assigned a string of blanks.

9d. Automating road map information retrieval

If the map of Ithaca (figure IV.9d) is represented as the grid

```
var Ithaca:grid[                              ]
          with subgrids r13[     ],
                        r79[     ],
                        r89[     ],
                        r366[    ],
                        r392[    ],
                        r34[     ],
                        r96[     ],
                        r966[    ],
                        state_street[  ],
                        business_district[  ] of integer;
```

then write program segments to

i) Find how close is a house located at coordinates i,j from road r13.

ii) Print Ithaca's map with '*' 's for points on State street and '.' for points on other roads.

iii) Find out where the roads r13 and r34 intersect

iv) change electronic speed signs on r392 by calling procedure change_sign. A point i,j has an electronic sign if ithaca[i,j] has the value -99.

Program segments to do the above are:

```
i)   c:=maxint;
     for i,j in domain(Ithaca,r13) do
                 begin d:=sqrt((m-i)**2+(n-j)**2);
                       if c>d then c:=d
                       end
ii)  for i,j in row domain(Ithaca) do
           if valid(ithaca.State_street,i,j)
                 then write('*')
                 else write('.')
iii) for i,j in domain(Ithaca,r13) do
           if valid(Ithaca,r13,i,j)
                 then begin write('they intersect at',i,j);
                           goto finish
                           end

     finish:
iv)  for i,j in domain(Ithaca.r392) do
           if Ithaca[i,j]= -99 then change_sign(i,j)
```

Comments

The declaration of the grid will be quite cumbersome but this is due to the complexity of Ithaca's road system. The size of the grid (and therefore the storage required) will depend upon how 'finely' (i.e. the amount of detail) the road system of Ithaca is represented.

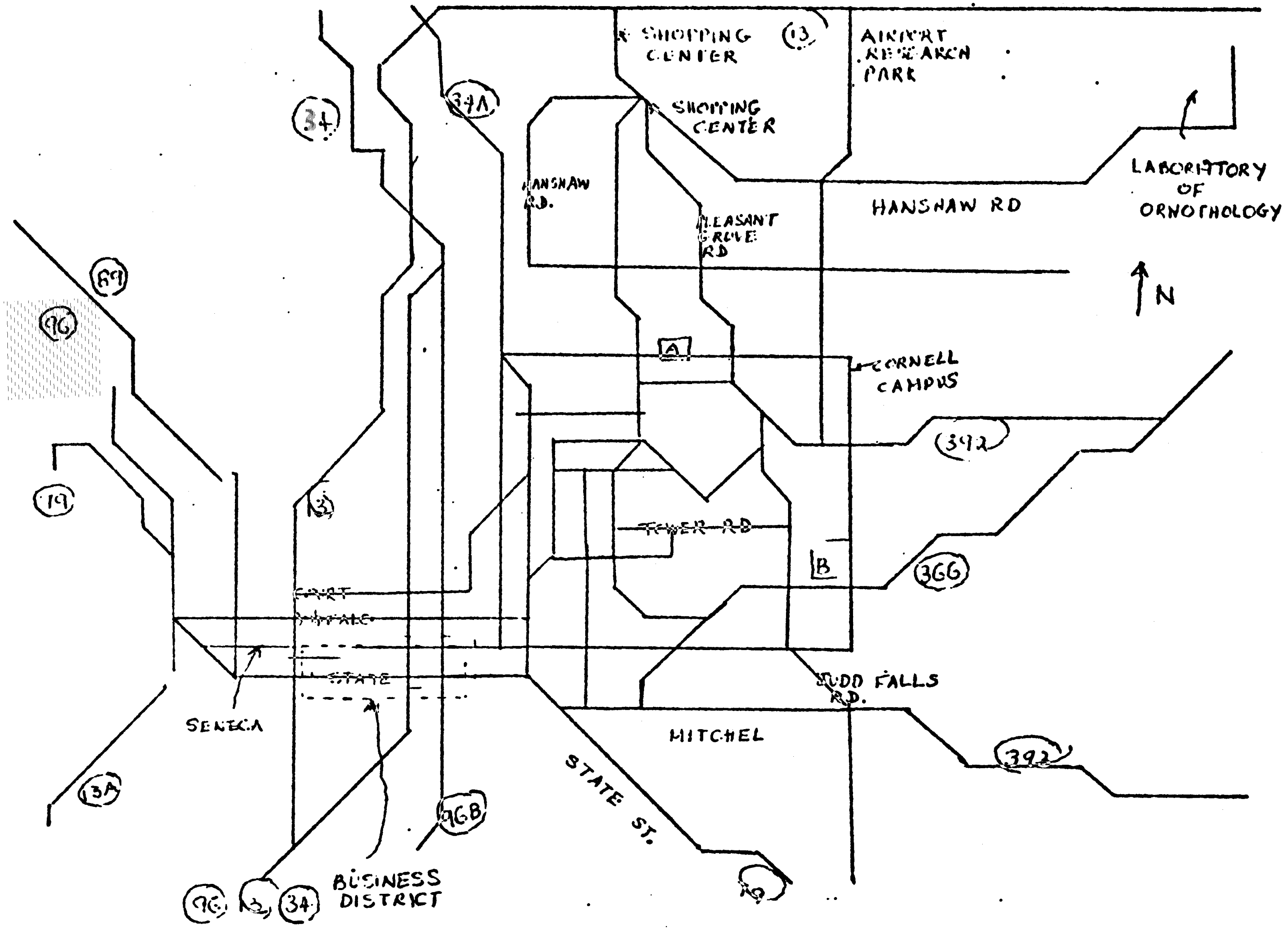


Fig [illegible].9.d Map of Ithaca

10. <u>Sparse grids</u>

We would like to indicate to the language processor that a positive grid part is sparse so that it may find an efficient representation for it. A grid is considered to be sparse if at least one of its parts has been specified to be sparse. Extension of the grid declaration syntax to allow positive parts to be specified sparse blends in naturally with the concept of grids.

Ordinary grid parts are non sparse i.e. dense. A positive grid part may be declared to be sparse with portions of it being specified as dense. A part should be declared sparse only if most of its elements are to have the same value (called sparse value). Otherwise, the program will run slowly.

The sparse value is assigned by the use of the psuedo function <u>sparse</u> in an executable statement of the form

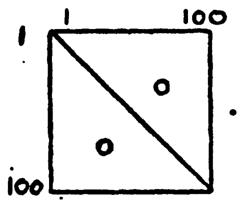
sparse(n):= e

where n is the name of a sparse part or a sparse grid and e is an expression whose type is the same as the elements of n. If n is the name of a sparse part then all the elements of the sparse part minus its dense elements are assigned the value e. If n is the name of a sparse grid then the above statement assigns the value e to all the sparse parts.

Declaring a grid to be sparse in general causes a reduction in the amount of storage to be used and an increase in execution time. Apart from this and the psuedo function sparse, sparse grids behave exactly as ordinary grids as far as the user is concerned. The domain of a dense part is the same as that of a subgrid declared with the same bounds (for the same grid, of course).

Examples

a.



diagonal matrix

```
var dm:grid[sparse[1..100,1..100] with dense[
                                  diag[i in 1..100,i..i]]] of real;
```

The above declares dm to be a grid with one unnamed part, and has one "dense" part named diag.

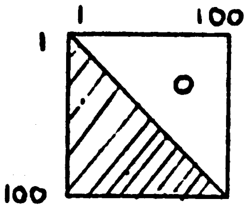
Some examples of assigning values to a sparse grid are:

i) dm:=8.0
   all elements of the grid dm are set to 8.0

ii) sparse(dm):=0.0
   all non diagonal elements of dm are set to 0.0

iii) diag:=1.0
   the diagonal elements are set to 1.0

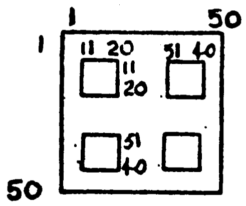
b.



Lower triangular matrix

```
var ltm:grid[sparse[1..100,1..100] with dense[lower
                                        [i in 1..100,1..i]]] of real;
```

c.

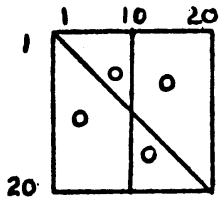


stiffness matrix for a beam
(in structural analysis)

```
var sm:grid[sparse[1..50,1..50] with dense[11..20,11..20;
                                           11..20,31..40;
                                           31..40,11..20;
                                           31..40,31..40]
                                              ] of real;
```

In this case the sparse part has four dense portions.

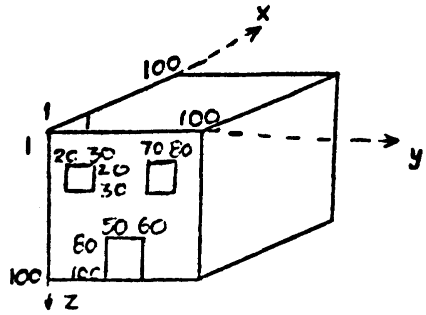
d.



left hand factor matrix L^j
In this case j=10

```
var hfm:grid[sparse[1..20,1..20] with dense[i in 1..20,i..i;
                                            1..9,10..10;
                                            11..20,10..10]
                                              ] of integer;
```

e.




```
var house:grid[sparse[1..100,1..100,2..99; minus
                     [2..99,2..99,2..99];
               sparse front_wall[1..100,1..100,1..1]
                     with dense
                     windows[20..30,20..30,1..1;
                             20..30,70..80,1..1];
                     door[80..100,50..60,1..1]
               with subgrid openings[window,door] of
                                                  real;
```

This example illustrates the case when a grid has more than one sparse part. A grid can have both sparse and non-sparse parts.

Notes

1. The dense portions must be contained in the corresponding part.
2. Subgrids do not have sparse components.
3. The sparse parts and their dense portions may or may not be named.
4. The sparse elements (the elements of a sparse part minus the elements of its dense portions) may be assigned values just like an ordinary grid element.
5. Elements of the dense portions behave like ordinary grid elements.

Advantages of sparse grids

1. The programmer is relieved from worrying about how to represent a sparse grid in the most efficient way.
2. Changing the shape of a sparse grid or its 'sparseness' may not require the modification of any statement except the grid declaration.
3. Sparse grids may be used to represent buildings in architectural programming applications.

11. Grids As Parameters

A grid may be passed to a procedure or function as an actual parameter. The corresponding formal parameter must be a grid too. The type of the formal parameter must be one of the following:

1. The same as that of the corresponding actual parameter.
2. The same as that of the corresponding actual parameter with some or all of the bounds replaced by '*' indicating that the bounds are to be received from the procedure or the function call (as in the case of PL/I array parameters).
3. If we are not interested in the parts of the grid, we may then just replace all the parts by $i_1, i_2, \ldots, i_n$ where

   i. n is the grid dimension

   ii. i_k='*' $1 \leq k \leq n$

   Similarly, the subgrid bounds may be replaced by $i_1, i_2, \ldots, i_n$.

The number of subgrids in the formal parameter should be less than or equal to the number of subgrids in the actual parameter, otherwise there is an error. Left to right matching of the subgrids of the formal and actual parameter grids occurs.

Examples

```
i)   var t_old:grid[1..100,1..100; minus 27..74,27..74]
               with subgrids inner_boundary[26..75,26..75;
                                   minus 27..74,27..74] .
                             outer_boundary[1..100,1..100;
                                   minus 2..99,2..99] .
                             inside_region[2..99,2..99;
                                   minus 26..75,26..75]
                                     of real
```

If t_old is used as an actual parameter then the type of the corresponding formal parameter could be

```
     i.   grid[*,*] of real
     ii.  grid[*,*; minus *,*] of real
     iii. grid[*,*] with subgrids ib[*,*],ob[*,*],ir[*,*] of
                                                        real
     iv.  grid[*,*] with subgrid a[*,*] of real
```

```
ii)  var a:grid[sparse[x[1..10,1..10] with dense[4..5,4..5;
                                  y[7..8,7..8] ]] of
                                                  integer
```

The type of the corresponding formal parameter could be

```
     i.  grid[*,*] of integer
     ii. grid[sparse a[*,*] with dense[*,*; b[*,*]] ] of integer
```

12. Data Security

If a portion of a grid is taken out (deleted or masked), then the resulting data structure is still a grid (this is not so in case of arrays). This fact is used in providing data security. When a grid is passed as a parameter, the calling procedure may not want the called procedure to be able to

1. write on portions of the grid
2. read or write on portions of the grid

This is termed data security. Data security is provided by the introduction of the readonly and mask options that may be appended after the actual parameter as shown below.

$$g \ \{\text{readonly}(x_1, x_2, \ldots, x_p)\} \ \{\text{mask}(y_1, y_2, \ldots, y_p)\}$$

where

1. g is the n-dimensional grid actual parameter
2. $x_i \ \ 1 \le i \le p$, $y_i \ \ 1 \le i \le q$ are subgrid or part names or $s_1..t_1, s_2..t_2, \ldots, s_n..t_n$. The s_i, t_i, $1 \le i \le n$ are expressions.
3. { } are meta brackets. {a} means a may or may not be included

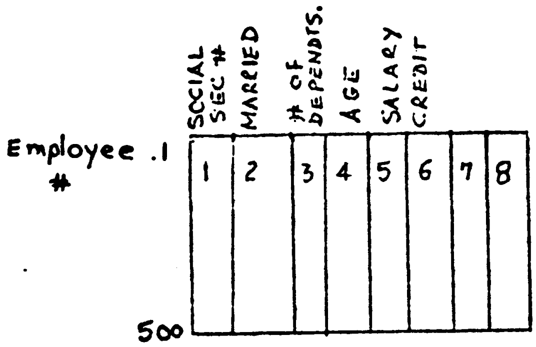
If in the called procedure, an attempt is made to read or write on a element of the grid formal parameter that corresponds to one of the y_i' elements, a security violation occurs causing an error. Similarily writing on a element that corresponds to one of the x_i's elements causes an error.

Notes

1. Grids may be passed by reference or value.
2. The domain of a grid, part or subgrid does not include elements that have been masked out.
3. Valid$(g, i_1, i_2, .., i_n)$ returns <u>false</u> if the element $i_1, i_2, .., i_n$ has been masked out.
4. The built-in functions ℓbound and ubound do not consider the masked out elements.

Examples

1. Consider the following grid named history which contains information about a campanies employees.



<u>var</u> history:<u>grid</u>[1..500,1..8] <u>of</u> integer

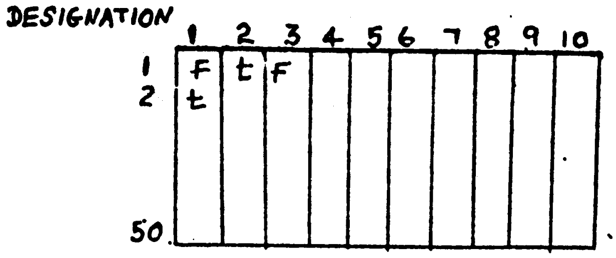
a) The grid history is passed to a procedure z with its salary and credit rating columns masked out to avoid divulging confidential information as follows:

z(history <u>mask</u>(1..500,5..6))

b) In case we want to allow the procedure z to look at the credit rating but not be able to modify it we call z as follows:

z(history <u>readonly</u>(1..500,6..6)<u>mask</u>(1..500,5..5))

2.



f = <u>false</u>
t = <u>true</u>

<u>var</u> access_matrix:<u>grid</u>[1..50,1..10] <u>of</u> boolean

Depending upon a computer center employee's (or user's) designation the access matrix would be passed to his procedures with certain restrictions.

13. <u>Implementation</u>

We shall briefly discuss two possible ways of implementing grids.

13a. <u>The smallest encompasing array (SEA) method -</u>
<u>representation I</u>

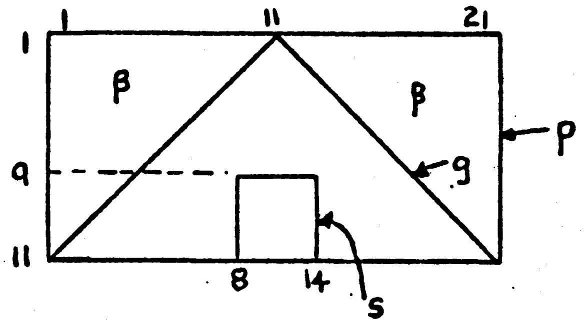
The grid is represented by the SEA. In WATFIV and PL/C (compilers for Fortran IV and PL/I) uninitialized variables are assigned a value α at the beginning of the program execution. Attempts to use variables without first assigning them a value detected by runtime checks for the value α, result in an error. Similarly, in the SEA grid representation, the non grid points will be assigned a value β at the beginning of program execution. Attempts to use non grid points, detected by the insertion of runtime checks to see if the grid element as a value β, will result in a 'subscripts out of bounds' error.

The general _for_ statement that iterates over the set of values _domain_($p_1, p_2, \ldots, p_j$, _minus_ q_1, _minus_ $q_2, \ldots$, _minus_ q_k), where the p_i's and the q_i's are n-dimensional grid, part or subgrid names, is implemented by reducing it to a set of n nested _for_ statements. These iterate over the bounds of the SEA of the 'effective' grid $p_1, p_2, \ldots, p_j$ _minus_ q_1, _minus_ $q_2, \ldots$, _minus_ q_k with the indices being generated in the right order. Conditional statements, to make sure that only the indices of the elements that belong to the 'effective' grid are generated, are inserted inside the nested _for_ statements.

Data security is implemented by storing values of the _readonly_ and _masked_ portions of the grid and passing the SEA as an actual parameter. On return from the called procedure, the values of the SEA are restored. Masked elements are set to β in the SEA before passing it to the called procedure. This method of implementing data security will be very expensive if a large number of elements are to be read only or masked out. An alternate implementation would be to pass three dope vectors, one each for the grid points, the read only points and the masked points. Runtime checks are inserted in the called procedure to ensure no violation of data security.

In general the SEA method could result in a lot of storage being wasted.

<u>Example</u>



<u>var</u> g:<u>grid</u>[i <u>in</u> 1..11; 11-i+1..11+i+1] <u>with</u> <u>subgrid</u>
s[9..11,8..14] <u>of</u> real

The grid g will be represented by a real array p that has the bounds 1..11, 1..21.

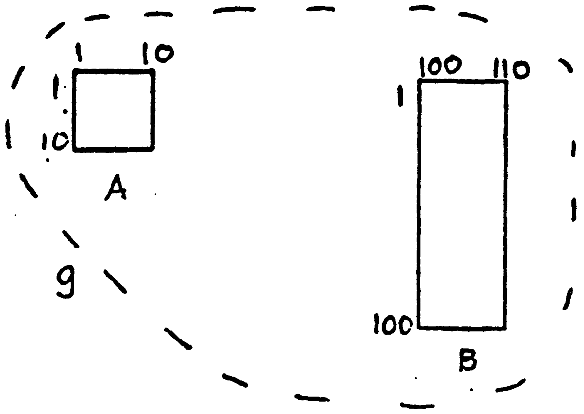
The statement

<u>for</u> i,j <u>in</u> <u>row</u> <u>domain</u>(g,<u>minus</u> g.s) <u>do</u> S

will be translated to

<u>for</u> i:=1 <u>to</u> 11 <u>do</u>
  <u>for</u> j:=1 <u>to</u> 21 <u>do</u>
    <u>if</u> p[i,j] ≠ β <u>and</u> p[i,j] is not an element of
      g.s <u>then</u> <u>do</u> S

An intelligent compiler should be able to do better than the simple translation scheme presented above for the <u>for</u> statement. Suppose we have the grid g shown below:



The statement

<u>for</u> i,j <u>in</u> <u>domain</u>(g) <u>do</u> S

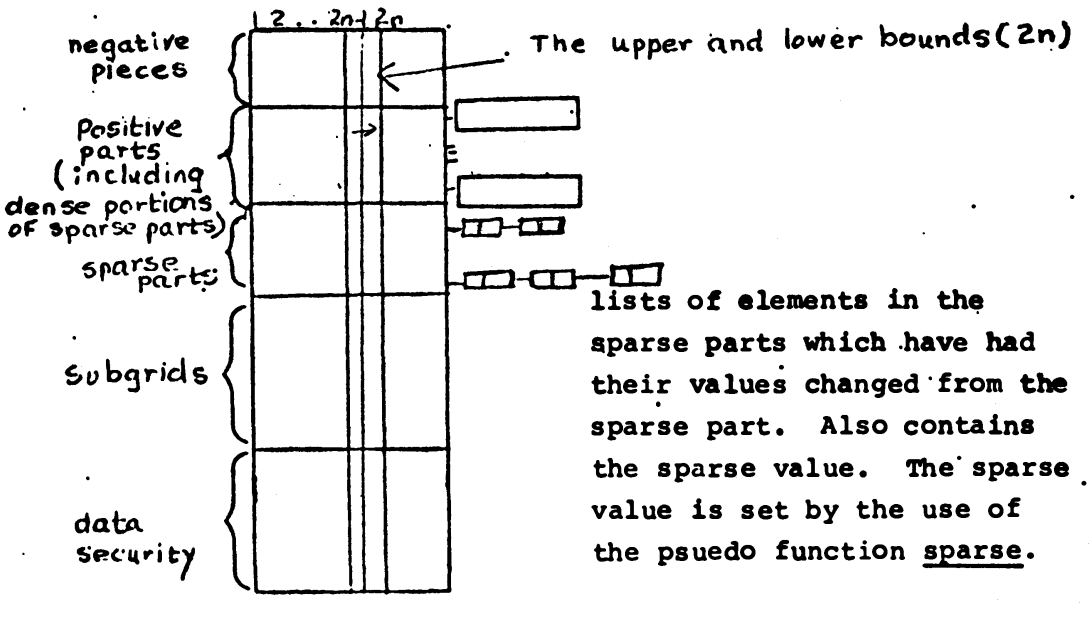
should be translated into a pair of disjoint loops with the same bodies iterating over parts A and B instead of over the SEA of g. The code produced would be

```
for i:=1 to 10 do
  for j:=1 to 10 do S
for i:= 1 to 100 do
  for j:= 100 to 110 do S
```

13b. <u>The descriptor array representation of a grid - representation II</u>

Each positive piece is represented by an array. We have assumed that pieces with functional bounds can be mapped onto arrays (elsewhere in this dissertation we discuss how pieces having certain types of functional bounds that can be represented by linear arrays). In the description array we store information (like bounds, address of where a particular part has been stored etc.) about the negative pieces and positive parts (including dense components of sparse parts), the sparse parts, the subgrids and the read only and masked portions of the grid. All grid references involve the use of the descriptor array.

<u>THE DESCRIPTOR ARRAY OF n-DIMENSIONAL GRID</u>



The general <u>for</u> statement is again reduced to a set of nested <u>for</u> statements with conditional statements inside to ensure that only the indices of the elements of the 'effective grid' are generated. The limits of the for statements are the bounds of the SEA of the 'effective grid'.

13c. Elimination of negative pieces

When using representation II, it would be more efficient to split up a positive piece that has a 'large' portion of it minused out into smaller pieces with the negative piece being elminated.

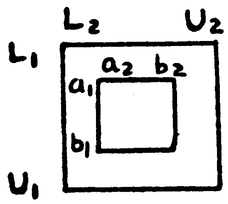
Suppose we have a declaration of the form $L_1..U_1, L_2..U_2,..,L_n..U_n$; minus $a_1..b_1, a_2..b_2,..,a_n..b_n$ then the negative piece can be eliminated by reducing the above two pieces to a union of positive pieces as shown below.

We assume for the moment that the minus part is encolsed by the positive part i.e. $L_i \le a_i$ and $b_i \le U_i$ for $i=1, n$.

1. The one-dimensional case

$L_1..U_1$; minus $a_1..b_1$ reduces to $L_1..a_1-1$; $b_1+1..U_1$

2. The 2-dimensional case

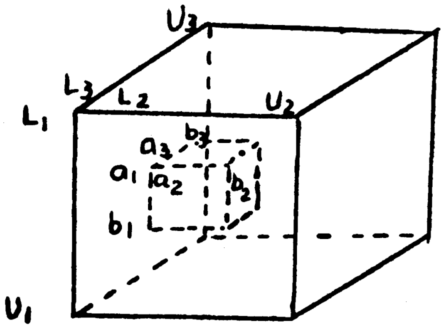


$L_1..U_1, L_2..U_2$; minus $a_1..b_1, a_2..b_2$

reduces to

$L_1..U_1, L_2..a_2-1$; $L_1..U_1, b_2+1..U_2$;
$L_I..a_1-1, a_2..b_2$; $b_1+1..U_1, a_2..b_2$

3. The 3-dimensional case



$L_1..U_1, L_2..U_2, L_3..U_3;$ <u>minus</u> $a_1..b_1,$
$a_3..b_3$

reduces to

$L_1..U_1, L_2..U_2, L_3..a_3-1;$
$L_1..U_1, L_2..U_2, b_3+1..U_3;$
$L_1..U_1, L_2..a_2-1, a_3..b_3;$
$L_1..U_1, b_2+1..U_2, a_3..b_3;$
$L_1..a_1-1, a_2..b_2, a_3..b_3;$
$b_1+1..a_1, a_2..b_2, a_3..b_3$

4. The n-dimensional case

$L_1..U_1, L_2..U_2, ..., L_n..U_n$ <u>minus</u> $a_1..b_1, a_2..b_2, ..., a_n..b_n$ } (A)

reduces to

$L_1..U_1, L_2..U_2, ..., L_{n-1}..U_{n-1}, L_n..a_n-1;$ ------ 2n
$L_1..U_1, L_2..U_2, ..., L_{n-1}..U_{n-1}, b_n+1..U_n;$ ------ 2n-1
$L_1..U_1, ..., L_{n-2}..U_{n-2}, L_{n-1}..a_{n-1}-1, a_n..b_n;$ -- 2n-2
$L_1..U_1, ..., L_{n-2}..U_{n-2}, b_{n-1}+1..U_{n-1}, a_n..b_n;$ -- 2n-3
.
.
$L_1..a_1-1, a_2..b_2, ..., a_n..b_n;$ ---------------- 2
$b_1+1..U_1, a_2..b_2, ..., a_3..b_3;$ ---------------- 1

} (B)

<u>Theorem</u> (A) is equivalent to (B) (i.e. all elements in (A) are in (B) and vice versa).

<u>Proof</u>

1. We show that all elements in (A) are also in (B).

   Assume that the element $i_1, i_2, .., i_n$ is in (A).
   This implies that

   i. $L_k \le i_k \le U_k$ for $k=1,n$
   ii. There exists r such that $\underline{not}(a_r \le i_r \le b_r)$

   Let $r_{max} = \max[r \mid \underline{not}(a_r \le i_r \le b_r)]$

   Parts $2*r_{max}$ and $2*_{max}-1$ of (B) are of the form

$$\begin{cases} L_1..U_1, .., L_{r_{max}-1}..U_{r_{max}-1}, L_{r_{max}}..a_{r_{max}}-1, \\ a_{r_{max}+1}..b_{r_{max}+1}, .., a_n..b_n \\ \\ L_1..U_1, .., L_{r_{max}-1}..U_{r_{max}-1}, b_{r_{max}}+1..U_{r_{max}}, \\ a_{r_{max}+1}..b_{r_{max}+1}, .., a_n..b_n \end{cases}$$

   Clearly the element $i_1, i_2, .., i_n$ is in part $2*r_{max}$ if $i_{r_{max}} \le a_{r_{max}}-1$ and in part $2*r_{max}-1$ if $i_{r_{max}} \ge b_{r_{max}}+1$

2. We now show that all elements in (B) are also in (A).

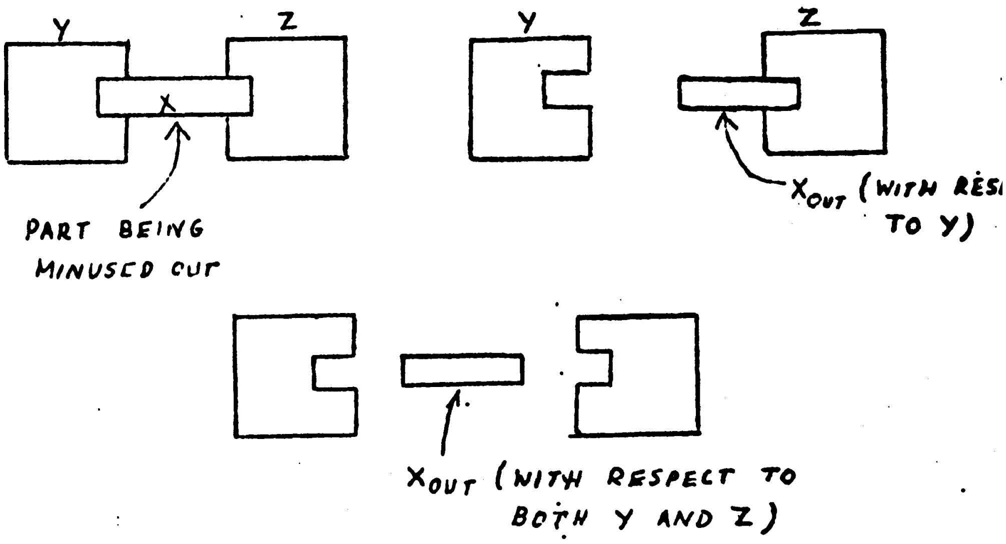
Assume that element $i_1, i_2, \ldots, i_n$ is in (B). This implies that

i. $L_k \le i_k \le U_k$ for k=1,n

ii. it belongs to one of the parts of (B) say k. Then <u>not</u>($a_r \le i_r \le b_r$) where r=truncate((k+1)/2) implying condition (ii, of part I). Thus $i_1, i_2, \ldots, i_n$ belongs to (A) also.

Q.E.D.

Till now we have restricted ourselves to the case when the <u>minus</u> part is completely enclosed by the part it is being subtracted from. We now remove this restriction and show how to eliminate <u>minus</u> parts that project out of the parts they are <u>minused</u> from. First a 2-dimensional pictorial view.



Part X of size $a_1..b_1, a_2..b_2, ..., a_n..b_n$ is <u>minused</u> from part Y of size $L_1..U_1, L_2..U_2, ..., L_n..U_n$

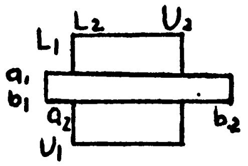
i.e. $\underbrace{L_1..U_1, L_2..U_2, ..., L_n..U_n}_{Y}$; <u>minus</u> $\underbrace{a_1..b_1, a_2..b_2, ..., a_n..b_n}_{X}$

For X to project out of Y the following conditions must be satisfied:

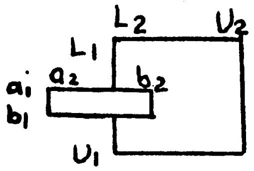
i. There exists r such that $L_r \leq a_r$, $b_r \leq U_r$

ii. There exists s such that one of the following is satisfied:

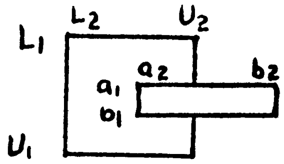
a) $a_s < L_s$ and $b_s > U_s$



b) $a_s < L_s$ and $L_s \leq b_s \leq U_s$



c) $L_s \leq a_s \leq U_s$ and $b_s > U_s$



These are the only 2 types of relations between a_i, b_i and L_i, U_i.

The following algorithm eliminates the *minus* part X.

```
while X projects out of Y do
    begin split X into X_IN and X_OUT with respect to
                                                    Y;
          set bounds of part X equal to bounds of X_IN
          if X_OUT projects out of another part Z in
          the grid declaration then add it to the list
          of minus parts else throw it away;
          end
    delete X from the list of minus parts;
```

In the above algorithm

1. X projects out of Y if conditions i. and ii. are satisfied.
2. split X into X_{IN} and X_{OUT} with respect to Y is defined as follows:

```
{as X projects out of Y there exists an i such
 that lbound(X,i)<lbound(Y,i) or ubound(X,i)>
 ubound(Y,i)}

set bounds of X_IN and X_OUT equal to bounds of X;
if lbound(X,i)<lbound(Y,i)
    then begin set lbound(X_IN,i) = lbound(Y,i);
               set ubound(X_IN,i) = ubound(X,i);
               set lbound(X_OUT,i) = lbound(X,i);
               set ubound(X_OUT,i) = lbound(Y,i)-1;
               end
```

```
else if ubound(X,i)>ubound(Y,i)
  then begin set lbound(X_IN,i)=lbound(X,i);
             set ubound(X_IN,i)=ubound(Y,i);
             set lbound(X_OUT,i)=ubound(Y,i)+1;
             set ubound(X_OUT,i)=ubound(X,i);
             end
```

13d. A representation selection algorithm

If procedures are to be compiled in conjunction with their calls, then both these representations may be used by the compiler or preprocessor that implements grids. The following decision algorithm helps in deciding what representation to use.

Let

1. S(g) be the minimum number of words of storage required to implement grid g.
2. S(SEA) be the minimum amount of storage required for g's SEA.
3. Let a and b be system defined constants which the user can alter.

```
if g does not have a sparse part and (S(SEA)<b or
                                      S(G)/S(SEA)>a)
     then use representation I for g
     else use representation II for g
```

Another parameter that may be considered in deciding what representation to be used is the total number of grid components i.e. the complexity of the grid. If the grid complexity is large the use of representation II would slow down program execution.

REFERENCES

1. Constable, R.L., To be published.

2. Dahl, O.J., B. Myhrhang and K. Nygaard. The Simula 67 Common Base Language. Norwegian Computing Center, Oslo 1968.

3. Fischer, Alice E. et. al. Mode modules as representations of domains. ACM Symposium on Principles of Programming Languages Oct 73, 139-143.

4. Jensen, K. and N. Wirth. Pascal user manual and report. Springer Verlag Lecture notes in Computer Science 18, 1975 (2nd edition).

5. Hoare, C.A.R. and N. Wirth, An axiomatic definition of the programming language Pascal. Eidgenossische Technische Hochschule Zurich, Berichte Der Fuchgar, Computer-Wissenschaften, nr. 6 Nov 1972.

6. Hoare, C.A.R. Record Handling in Programming Languages edited by Genuys, F. Academic Press, New York 1968.

7. Morris, J.H. Types are not sets. ACM Symposium on Principles of Programming Languages Oct 73, Boston.

8. Organick, E.I. A Computer Primer for the MAD Language. Univ. of Mich., Ann Arbor, Mich. 1961 and Computer Data Proc. Center, Univ. of Houston, Houston, Texas 1961.

9. Perlis, A.J. The synthesis of algorithmic systems. JACM v14, number 1, 1-9.

10. Remorenko, M. Dept. of Mechanical Engineering, Cornell University, Ithaca, N.Y. 14853

11. Spiegel, M.R. Mathematical Handbook of formulas and tables. Schaums outline series - McGraw-Hill Book Company, New York.

12. Standish, T.A. A data definition facility for programming languages. Ph.D. thesis May 1967, Carngie Institute of Technology, Pittsburg, Pa.

13. Wegbreit, B. The treatment of data types in EL1.. CACM 17,5 May 74, 251-264.

14. Wirth, N. The programming language PASCAL (revised report). Eidgenossische Technische Hochschule Zurich. Berichte Der Fachgar, Computer - Wissenschaften, nr. 5, Nov 1972.

15. Wulf, W.A. Alphard: Towards a language to support structured programs, Dept. of Computer Science, Carneige Mellon University, Pittsburgh, Pa (April 74)

APPENDICES

APPENDIX A

Efficient representations for grids with functional bounds

Some special cases of grids with functional bounds have been considered and an efficient representation derived for them.

Case 1.

var g:grid[j_1 in a..b,j_2 in $f_2(j_1)..g_2(j_1),\ldots,j_n$ in $f_n(j_1)..$
$g_n(j_1)$] of s;

where

1. $n \geq 2$
2. $a \leq b$ for the grid declaration to be non-vacuous and are both integers
3. $f_i(\ )$ and $g_i(\ )$ $2 \leq i \leq n$ are linear functions with integer range.

   i.e.

   $f_i(p) = h_i(p) + c_i \quad 2 \leq i \leq n$

   $g_i(p) = k_i(p) + d_i \quad 2 \leq i \leq n$

   The h_i's, k_i's, c_i's and d_i's are constants.
4. $j_1 \in a..b\ (f_i(j_1) \leq g_i(j_1))$ $\quad 2 \leq i \leq n$

   for the grid declaration to be non-vacuous. The above is true if $j_1 \quad a..b\ (g_i(j_1) - f_i(j_1) \geq 0)$

   $2 \leq i \leq n$ or if

$$g_i(a) - f_i(a) \geq 0 \quad 2 \leq i \leq n$$
$$g_i(b) - f_i(b) \geq 0 \quad 2 \leq i \leq n$$

We want to find

i. The minimum total storage required to represent g i.e. $m_g(a,b)$.

ii. a representation for g.

iii. How to access $g[i_1,i_2,..,i_n]$ in its representation.

There are 3 cases of the above problem:

I. $a \geq 0$ ($\Rightarrow b \geq 0$ because $a \leq b$)

II. $b \leq 0$ ($\Rightarrow a \leq 0$ because $a \leq b$)

III. $a \leq 0$ and $b \geq 0$

I.

$$m_g(a,b) = \sum_{i=a}^{b} \prod_{r=2}^{n} (g_r(i) - f_r(i) + 1)$$

$$\sum_{i=a}^{b} \prod_{r=2}^{n} (k_r - h_r)i + (d_r - c_r + 1)$$

$$\left.\begin{aligned} \text{Let } x_r &= k_r - h_r \\ y_r &= d_r - c_r + 1 \end{aligned}\right\} \quad 2 \leq r \leq n$$

$$m_g(a,b) = \sum_{i=a}^{b} \prod_{r=2}^{n} (x_r i + y_r)$$

$$= \sum_{i=a}^{b} (x_2 i + y_2)(x_3 i + y_3)..(x_n i + y_n)$$

$$= \sum_{i=a}^{b} \{t_{n-1}\, i^{n-1} + t_{n-2}\, i^{n-2} + t_{n-3}\, i^{n-3} + \ldots$$

$$+ t_2\, i^2 + t_1\, i + t_0\}$$

The t_i's are constants and have the following values:

$$t_{n-1} = x_2 x_3 x_4 x_5 \ldots x_n = \prod_{r=2}^{n} x_r$$

$$t_{n-2} = \sum_{k=2}^{n} (x_3 x_4 \ldots x_{k-1} y_k x_{k+1} \ldots x_n)$$

$$t_{n-3} = y_2 \sum_{k=3}^{n} (x_3 x_4 \ldots x_{k-1} y_k x_{k+1} \ldots x_n)$$

$$+ x_2 y_3 \sum_{k=4}^{n} (x_4 x_5 \ldots x_{k-1} y_k x_{k+1} \ldots x_n)$$

$$\vdots$$

$$+ x_2 x_3 \ldots x_{n-2} y_{n-1} \sum_{k=n}^{n} y_k$$

$$= \sum_{q=2}^{n-1} \{(x_2 x_3 \ldots x_{q-1} y_q) \sum_{k=q+1}^{n} (x_{q+1} x_{q+2} \ldots x_{k-1} y_k x_{k+1} \ldots x_n)\}$$

$$t_{n-4} = \sum_{p=2}^{n-2} \{(x_2 \ldots x_{p-1} y_p) \sum_{q=p+1}^{n-1} \{(x_{p+1} \ldots x_{q-1} y_q)$$

$$\vdots$$

$$\sum_{k=q+1}^{n} (x_{q+1} \ldots x_{k-1} y_k x_{k+1} \ldots x_n)\}\}$$

$$t_0 = \sum_{r=2}^{2} y_r \sum_{m=r+1}^{3} y_m \ldots \sum_{q=}^{n-1} y_q \sum_{k=q+1}^{n} y_k = y_2 y_3 y_4 \ldots y_n = \prod_{i=2}^{n} y_i$$

$$m_g(a,b) = \sum_{i=a}^{b} (t_{n-1}\, i^{n-1} + t_{n-2}\, i^{n-2} + \ldots + t_1\, i + t_0)$$

$$= t_{n-1} \sum_{i=a}^{b} (i^{n-1}) + t_{n-2} \sum_{i=a}^{b} (i^{n-2}) + \ldots + t_1 \sum_{i=a}^{b} i + t_0 \sum_{i=a}^{b} 1$$

1. $\sum_{i=a}^{b} 1 = (b-a+1)$

2. Because $0 \leq a \leq b$

$$\sum_{i=a}^{b} (i^p) = \sum_{i=0}^{b} (i^p) - \sum_{i=0}^{a-1} (i^p) \qquad \forall p$$

$$\begin{aligned} m_g(a,b) = {} & t_{n-1} \left\{ \sum_{i=0}^{b} i^{n-1} - \sum_{i=0}^{a-1} i^{n-1} \right\} \\ & + t_{n-2} \left\{ \sum_{i=0}^{b} i^{n-2} - \sum_{i=0}^{a-1} i^{n-2} \right\} \\ & \vdots \\ & + t_1 \left\{ \sum_{i=0}^{b} i - \sum_{i=0}^{a-1} i \right\} \\ & + t_0 (b-a+1) \end{aligned}$$

Let $m(x) = t_{n-1} \sum_{i=0}^{x} i^{n-1} + t_{n-2} \sum_{i=0}^{x} i^{n-2} + \ldots + t_1 \sum_{i=0}^{x} i + t_0 (x+1)$

Then

$$m_g(a,b) = m(b) - m(a-1)$$

The following formula is taken from Spiegel [11] .

$$\sum_{i=0}^{b} i^n = \frac{b^{n+1}}{n+1} + \frac{b^n}{2} + \frac{B_1}{2!} \; n\, b^{n-1} - \frac{B_2}{4!} n(n-1)(n-2)b^{n-3} + \ldots.$$

The series ends at the b^2 term if n is odd and at the b term if n is even. B_i's are the Bernoulli numbers. Let $\div$ represent integer division. Then the above formula can be rewritten as

$$\sum_{i=0}^{b} i^n = \frac{b^{n+1}}{n+1} + \frac{b^n}{2} + \sum_{j=1}^{n \div 2} \frac{(-1)^{j+1}}{(2*j)!} B_j \left[\prod_{k=0}^{2*j-2} (n-k) \right] b^{n-2*j+1}$$

The Bernoulli numbers are given in Spigel [11] and the first few of them are:

B_1 = 1/6

B_2 = 1/30

B_3 = 1/42

B_4 = 1/30

B_5 = 5/66

B_6 = 691/2730

B_7 = 7/6 etc

We now expand the terms of m(x) so as to be able to collect the coefficients of the x^i's.

$$t_{n-1} \sum_{i=0}^{x} i^{n-1} = t_{n-1} \left[\frac{x^n}{n} + \frac{x^{n-1}}{2} + \sum_{j=0}^{(n-1)\div 2} \frac{(-1)^{j+1} B_j}{(2*j)!} \right.$$

$$\left\{ \prod_{k=0}^{2*j-2} (n-1-k) \right\} x^{n-1-2*j+1}$$

$$t_{n-2} \sum_{i=0}^{x} i^{n-1} = t_{n-2} \left[\frac{x^{n-1}}{n-1} + \frac{x^{n-2}}{2} + \sum_{j=1}^{(n-2)\div 2} \frac{(-1)^{j+1} B_j}{(2*j)!} \right.$$

$$\left. \left\{ \prod_{k=0}^{2*j-2} (n-2-k) \right\} x^{n-2-2*j+1} \right]$$

$$t_{n-3} \sum_{i=0}^{x} i^{n-3} = t_{n-3} \left[\frac{x^{n-2}}{n-2} + \frac{x^{n-3}}{2} + \sum_{j=1}^{(n-3)\div 2} \frac{(-1)^{j+1} B_j}{(2*j)!} \right.$$

$$\left. \left\{ \prod_{k=0}^{2*j-2} (n-3-k) \right\} x^{n-3-2*j+1} \right]$$

$$\vdots$$

$$t_2 \sum_{i=0}^{x} i^2 = t_2 \left[\frac{x^3}{3} + \frac{x^2}{2} + \frac{B_1 (2x)}{2!} \right]$$

$$t_1 \sum_{i=0}^{x} i = t_1 \left[\frac{x^2}{2} + \frac{x}{2} \right]$$

$$t_0 x = t_0 x$$

Let e_i be the coefficients of x^i in the expansion of $m(x)$. They have the values

$$e_n = \frac{t_{n-1}}{n}$$

$$e_{n-1} = \frac{t_{n-1}}{2} + \frac{t_{n-2}}{n-1}$$

$$e_{n-2} = \frac{t_{n-1}\ B_1\ (n-1)}{2!} + \frac{t_{n-2}}{2} + \frac{t_{n-3}}{n-2}$$

$$e_{n-3} = \frac{t_{n-2}\ B_1\ (n-2)}{2!} + \frac{t_{n-3}}{2} + \frac{t_{n-4}}{n-3}$$

$$e_{n-4} = \frac{-t_{n-1}\ B_2}{4!} \prod_{r=1}^{3} (n-r) + \frac{t_{n-3}\ B_1\ (n-3)}{2!} + \frac{t_{n-4}}{2} + \frac{t_{n-5}}{n-4}$$

$$e_{n-5} = \frac{-t_{n-2}\ B_2}{4!} \prod_{r=1}^{4} (n-r) + \frac{t_{n-3}\ B_1\ (n-4)}{2!} + \frac{t_{n-5}}{2} + \frac{t_{n-6}}{n-5}$$

$$e_{n-6} = \frac{t_{n-1}\ B_3}{6!} \prod_{r=1}^{5} (n-r) - \frac{t_{n-3}\ B_2}{\text{[illegible]}!} \prod_{r=3}^{5} (n-r) + \frac{B_1 t_{n-5} (n-5)}{2!}$$

$$+ \frac{t_{n-6}}{2} + \frac{t_{n-7}}{n-6}$$

.

.

.

e_1

The general formulas for these coefficients are given by

i. $e_n = \dfrac{t_{n-1}}{n}$

The rest of the coefficients $e_{n-1}, e_{n-2}, \ldots, e_1$ are calculated using either ii or iii.

for $1 \le i \le n \div 2$

ii. $$e_{n-2*i} = \frac{t_{n-2*i-1}}{n-2*i} + \frac{t_{n-2*i}}{2} + \sum_{j=1}^{i} \left\{ \frac{t_{n+2(j-i)-1} B_j}{(2*j)!} (-1)^{j+1} \prod_{k=1}^{2*j} (n-2*i+k) \right\}$$

iii. $$e_{n-2*i+1} = \frac{t_{n-2*i-2}}{n-2*i-1} + \frac{t_{n-2*i-1}}{2} \sum_{j=1}^{i} \left\{ \frac{t_{n+2(j-1)-2} B_j}{(2*j)!} (-1)^{j+1} \prod_{k=1}^{2*j-1} (n-2*i+k-1) \right\}$$

The minimum amount of storage required to represent the grid g can now be expressed as

$$m_g(a,b) = [e_n b^n + e_{n-1} b^{n-1} + .. + e_1 b] - [e_n (a-1)^n + .. + e_1 (a-1)] = \sum_{i=1}^{n} e_i \; (b^i - (a-1)^i)$$

ii. The grid g can be represented by a linear array having the bounds

$$0 \; .. \; m_g(a,b)-1$$

with elements of type s.

iii. We now calculate the index of $g[j_1, j_2, \ldots, j_n]$ in the linear array. For $\mathrm{valid}(g, j_1, j_2, \ldots, j_n)$ to be true

$$a \leq j_1 \leq b \text{ and } f_k(j_1) \leq j_k \leq g_k(j_1) \quad 2 \leq k \leq n$$

The index I in the linear array is given by

$$I = m_g(a,j_1-1) + \sum_{p=2}^{n} [j_p - f_p(j_1)] \prod_{q=p+1}^{n} (g_q(j_1) - f_q(j_1) + 1).$$

$$= m_g(a,j_1-1) + \sum_{p=2}^{n} [j_p - h_p j_1 - c_p] \prod_{q=p+1}^{n} [(k_q - h_q) j_1 + (d_q - c_q + 1)]$$

$$= m_g(a,j_1-1) + \sum_{p=2}^{n} [j_p - h_p j - c_p] \prod_{q=p+1}^{n} [x_q j_1 + y_q]$$

where the x_q's and the y_q's have been defined before.

$$I = m_g(a,j_1-1) + \sum_{p=0}^{n} [j_p - c_p] \prod_{q=p+1}^{n} (x_q j_1 + y_q) -$$

$$\underbrace{\sum_{p=2}^{n} h_p j_1 \prod_{q=p+1}^{n} (x_q j_1 + y_q)}_{(*)}$$

We now expand (*) so as to put it in the form

$$E_{n-1} j_1^{n-1} + E_{n-2} j_1^{n-2} + \ldots + E_1 j_1$$

$$-E_{n-1} = h_2 \prod_{q=3}^{n} x_q$$

$$-E_{n-2} = h_2 \sum_{q=3}^{n} (x_3 \ldots y_q \ldots x_n) + h_3 \prod_{q=4}^{n} x_q$$

$$-E_{n-3} = h_2 \sum_{q=3}^{n-1} [x_3 \ldots x_{q-1} y_q \sum_{p=q+1}^{n} (x_{q+1} \ldots x_{p-1}\ y_p\ x_{p+1} \ldots x_n)]$$

$$+ h_3 \sum_{q=4}^{n} (x_4 \ldots x_{q-1}\ y_q\ x_{q+1} \ldots x_n) + h_4 \prod_{q=5}^{n} x_q$$

$$-E_{n-4} = h_2 \sum_{r=3}^{n-2} [x_3 \ldots x_{r-1} y_r \sum_{q=r+1}^{n-1} (x_{r+1} \ldots x_{q-1} y_q \sum_{p=q+1}^{n} (x_{q+1} \ldots y_p \ldots$$

$$x_n)] + h_3 \sum_{q=4}^{n-1} [x_4 \cdot\cdot x_{q-1} y_q \sum_{p=q+1}^{n}$$

$$(x_{q+1} \cdot\cdot y_p \cdot\cdot x_n)] + h_4 \sum_{q=5}^{n} (x_5 \cdot\cdot y_q \cdot\cdot$$

$$x_n) + h_5 \sum_{q=6}^{n} x_q$$

$$\vdots$$

$$-E_1 = h_2 \prod_{q=3}^{n} y_q + h_3 \sum_{q=4}^{n} y_q + \cdot\cdot + h_n \sum_{q=n}^{n} y_q$$

$$I = m_g(a, j_1 - 1) + \sum_{p=1}^{n-1} E_p j_1^p + \sum_{p=2}^{n} [j_p - c_p] \prod_{q=p+1}^{n} (x_q j_1 + y_q)$$

Now $m_g(a, j_1-1) = m(j_1-1) - m(a)$

We let $m'(j_1-1) = m(j_1-1) + \sum_{p=1}^{n-1} E_p\, j_1^p$

Since $m()$ is a polynomial of degree n, $m'()$ is also a polynomial of degree n.

$$I = m'(j_1-1) - m(a) + \sum_{p=2}^{n} [j_p - c_p] \prod_{q=p+1}^{n} (x_q j_1 + y_q]$$

We would like to know the cost of evaluating I. m(a) is constant for a grid and we just need to evaluate it once. This requires n *(multiplications) as m(a) is a polynomial of degree n. $m'(j_1-1)$ is also a polynomial of degree n and hence it requires n *. m'() must be evaluated for every grid element (unless if we want to calculate it for all

j_1's and store the resulting values).

To calculate the third term $\sum_{p=2}^{n} [j_p - c_p] \prod_{q=p+1}^{n} (x_q j_1 + y_q)]$

of I we adopt the following scheme.

$$\sum_{p=2}^{n} (j_p - c_p) \prod_{q=p+1}^{n} (x_q j_1 + y_q) = (j_n - c_n) + \left[\sum_{p=2}^{n-1} (j_p - c_p) \prod_{q=p+1}^{n-1} (x_q j_1 + y_q) \right] * (x_n j_1 + y_n) = (j_n - c_n) + \left[(j_{n-1} - c_{n-1}) + \left\{ \sum_{p=2}^{n-2} (j_p - c_p) \prod_{q=p+1}^{n-2} (x_q j_1 + y_q) \right\} (x_{n-1} j_1 + y_{n-1}) \right] (x_n j_1 + y_n)$$

Note: $\prod_{\ell=n+1}^{n} () = 1$

Let M(n) be the number of multiplications required to calculate the above term. Inspection of the recursive expansion shows that

$M(n) = M(n-1) + 2$

$M(2) = 0$

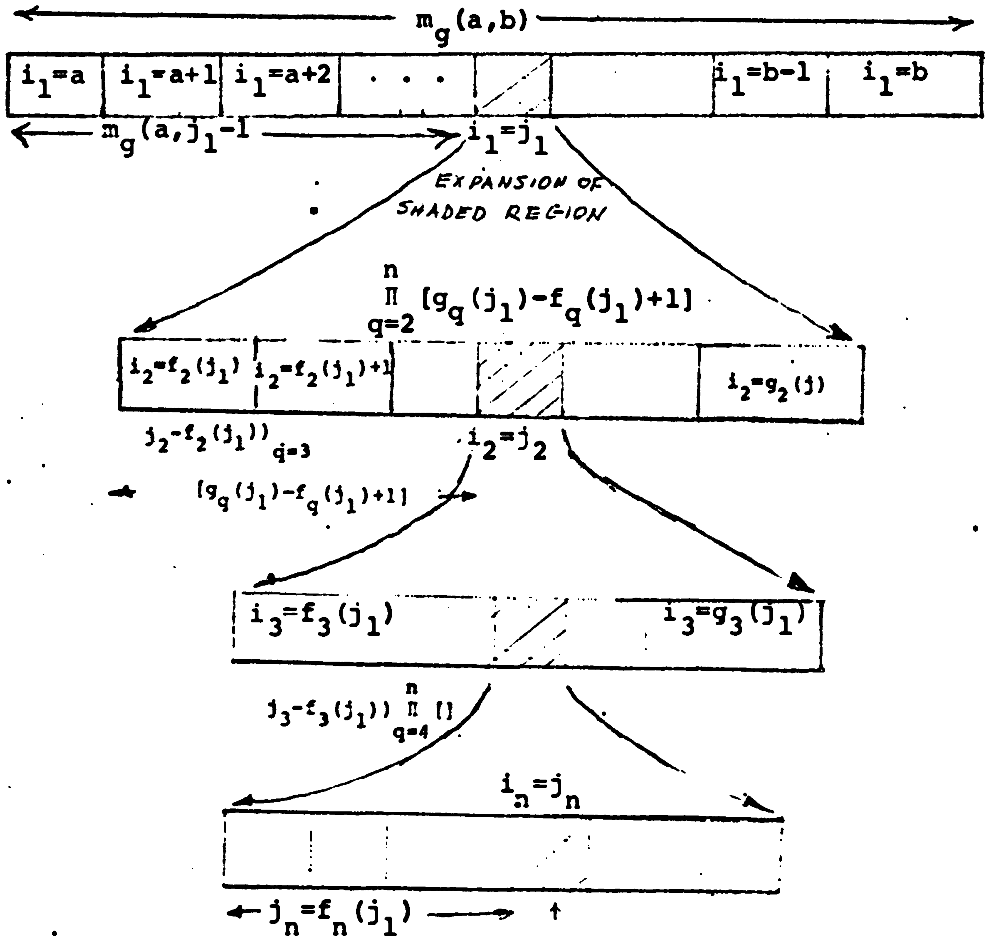
$\Longrightarrow$ M(n) has the solution 2(n-2).

Total number of multiplications required to evaluate the index = 3n-2.

A PICTORIAL VIEW OF ACCESSING A GRID ELEMENT $g[j_1,j_2,\ldots,j_n]$

IN ITS LINEAR REPRESENTATION.

$$I = m_g(a,j_1-1) + \sum_{p=2}^{n} (j_p-f_p(j_1)) \prod_{q=p+1}^{n} [g_q(j_1)-f_q(j_1)+1]$$

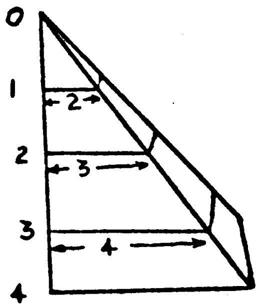


The element in the linear array representing $g[j_1,j_2,\ldots,j_n]$

Example

var g:**grid**[i **in** 0..4, j **in** 0..i, 0..i] **of** a

We calculate the storage required for this grid by inspection on the right hand side and using the formulas previously derived down below.



NUMBER OF ELEMENTS REQUIRED = 1+4+9+16 +25 = 55

n=3

$k_2=0,\ h_2=0,\ c_2=0,\ d_2=0$

$k_3=0,\ h_3=0,\ c_2=0,\ d_3=0$

$x_2=1,\ y_2=1$

$x_3=1,\ y_3=1$

$t_2=x_2x_3=1$

$t_1=y_2x_3+x_2y_3=2$

$t_0=y_2y_3=1$

$c_3=t_2/3 = 1/3$

$c_2=t_2/2 + t_1/2 = 2/2 +1/2 = 3/2$

$c_1=t_2/6 + t_1/2 + t_o = 13/6$

$$m_g(a,b) = \sum_{i=1}^{3} (b^i-(a-1)^i) = 65/3 + 45/2 + 65/6 = 55$$

The grid can be represented by the array

<u>array</u>[0..54] <u>of</u> s

The index I is given by

$$I = m_g(a,j_1-1) + \sum_{p=1}^{n-1} e_p j_1^p + \sum_{p=2}^{n-1} [j_p - c_p] \prod_{q=p+1}^{n} (x_q j_1 + y_q)$$

$$m_g(a,j_1-1) = j_1^3/3 + j_1^2/2 + j_1/6$$

(we used the coefficients calculated before)

$e_2 = 0$

$e_1 = 0$

$$I = j_1^3/3 + j_1^2/2 + j_1/6 + j_2 + j_3 + j_1 j_2$$

g[0,0,0] has the index 0 in the linear array

g[4,4,4] has index I = 64/3 + 16/2 + 4/6 + 4 + 4 + 16 = 54

<u>Cases b and c</u>

Because the identity

$$\sum_{i=a}^{b} i^p = \sum_{i=0}^{b} i^p - \sum_{i=0}^{a-1} i^p \qquad \text{for all } p$$

holds for all values of a and b our results for the first case (i.e. $a \geq 0$ and $b \geq 0$) also hold for these cases.

Case 2.

<u>var</u> g:<u>grid</u>[i <u>in</u> a..b, f(i)..g(i)] <u>of</u> s;

where

1. a and b are integers and $a \leq b$ for the declaration to be non-vacuous.
2. $i \in a..b$ $[f(i) \leq g(i)]$ for the declaration to be non-vacuous.
3. f() and g() are integer valued polynomials of degree n.
   i.e.
   $$f(i) = h_n i^n + h_{n-1} i^{n-1} + .. + h_1 i + c$$
   $$g(i) = k_n i^n + k_{n-1} i^{n-1} + .. + k_1 i + d$$

Condition (2) can be verified by showing i a..b $[g(i)-f(i) \geq 0]$. If f() and g() are linear, i.e. $n=1$, then we just have to show that $g(i)-f(i) \geq 0$ at $i=a$ and $i=b$. If $n \geq 2$ then we must show that $g(i)-f(i) \geq 0$ at the minimas of the polynomial $g(i)-f(i)$ that lie in the range a..b. If no minimas lie in the range a..b, then we just need to check at the end points.

As before we want to find

i. The minimum total storage $m_g(a,b)$ required to represent g.
ii. A representation for g.
iii. How to access $g[i_1,i_2]$

We shall compute our results assuming a and b are both ≥ 0. The results for the other cases will be identical to the one computed becuase of the reason mentioned before.

$$m_g(a,g) = \sum_{i=a}^{b} (g(i)-f(i)+1)$$

$$= \sum_{i=a}^{b} [(k_n-h_n)i^n+(k_{n-1}-h_{n-1})i^{n-1}+..+(k_1-h_1)i+(d-c+1)]$$

$$= \sum_{i=a}^{b} [x_n i^n+x_{n-1}i^{n-1}+..+x_1 i+x_0]$$

where $x_0 = d - c + 1$

$x_i = k_i - h_i \qquad 1 \le i \le n$

Let $N(p) = \sum_{i=0}^{p} [x_n i^n+x_{n-1}i^{n-1}+..+x_1 i+x_0]$

Then $m_g(a,b) = N(b) - N(a-1)$ as

$$m_g(a,b) = \sum_{i=0}^{b} [x_n i^n+x_{n-1}i^{n-1}+..+x_1 i+x_0] - \sum_{i=0}^{a-1} [x_n i^n+x_{n-1}i^{n-1}+..+x_1 i+x_0]$$

We want to rewrite $N(p)$ in the form $N(p) = \sum_{i=0}^{n+1} y_i p^i$

We first expand $N(p)$ using the formula

$$\sum_{i=0}^{p} i^n = \frac{p^{n+1}}{n+1} + \frac{p^n}{2} + \sum_{j=1}^{n\div 2} \frac{(-1)^{j+1}B_j}{(2*j)!} \prod_{k=0}^{2*j-2} (n-k)\ p^{n-2*j+1}$$

and collect the coefficients of the p^i's.

Note B_i's are the Bernoulli constants.

$$N(p) = x_n \left(\frac{p^{n+1}}{n+1} + \frac{p^n}{2} + \sum_{j=1}^{n\div 2} \frac{(-1)^{j+1} B_j}{(2*j)!} \prod_{k=0}^{2*j-2} (n-k) p^{n-2*j+1} \right)$$

$$+ x_{n-1} \left(\frac{p^n}{n} + \frac{p^{n-1}}{2} + \sum_{j=1}^{(n-1)\div 2} \frac{(-1)^{j+1} B_j}{(2*j)!} \prod_{k=0}^{2*j-2} (n-1-k) p^{n-1-2*j+1} \right)$$

$$+ x_{n-2} \left(\frac{p^{n-1}}{n-1} \frac{p^{n-2}}{2} + \sum_{j=1} \frac{(-1)^{j+1} B_j}{(2*j)!} \prod_{k=0} (n-2-k) p^{n-2-2*j+1} \right)$$

$$+ \vdots$$

$$+ x_1 \left(\frac{p^2}{2} + \frac{p}{2} \right)$$

$$+ x_0 (p+1)$$

Collecting the foefficients of the p^i's we get

$$y_{n+1} = \frac{x_n}{n+1}$$

$$y_n = \frac{x_{n-1}}{n} + \frac{x_n}{2}$$

$$y_{n-1} = \frac{x_{n-2}}{n-1} + \frac{x_{n-1}}{2} + \frac{B_1 (n)}{2!} x_n$$

$$y_{n-2} = \frac{x_{n-3}}{n-2} + \frac{x_{n-2}}{2} + \frac{B_1 (n-1)}{2!} x_{n-1}$$

$$y_{n-3} = \frac{x_{n-4}}{n-3} + \frac{x_{n-3}}{n-2} + \frac{B_1 (n-2)}{2!} x_{n-2} - \frac{B_2}{4!} x_n (n)(n-1)(n-2)$$

$$y_{n-4} = \frac{x_{n-5}}{n-4} + \frac{x_{n-4}}{n-3} + \frac{B_1 (n-3)}{2!} x_{n-3} - \frac{B_2}{4!} x_{n-1} (n-1)(n-2)(n-3)$$

$$y_1 = p$$

$$y_0 = 1$$

The above formulas are generalised as follows;

i. $y_{n+1} = \frac{x_n}{n+1}$

ii. $y_0 = 1$

iii.a. for $0 \leq i \leq n \div 2$

$$y_{n-2*i} = \frac{x_{n-2*i-1}}{n-2*i} + \frac{x_{n-2*i}}{2} + \sum_{j=1}^{i} \left[\frac{(-1)^{j+1} \; B_j}{(2*j)!} \; x_{n+2*(j-1)-1} \prod_{k=1}^{2*j-1} (n-2*i+k) \right]$$

$$y_{n-2*i+1} = \frac{x_{n-2*i}}{n-2*i+1} + \frac{x_{n-2*i+1}}{2} + \sum_{j=1}^{i} \left[\frac{(-1)^{j+1} \; B_j}{(2*j)!} \; x_{n+2*(j-1)} \prod_{k=1}^{2*j-1} (n-2*i+k+1) \right]$$

The minimum total storage required for g is now given by

$$m_g(a,b) = \sum_{i=0}^{n+1} y_i (b^i - (a-1)^i)$$

ii. The grid g can be represented by a linear array having the bounds

$$0..m_g(a,b)$$

with elements of type s..

iii. We calculate the index of g[i,j] in the linear array as follows:

For valid(g,i,j) to be true $a \leq i \leq b$ and $f(i) \leq j \leq g(i)$.

The index I in the linear array is given by

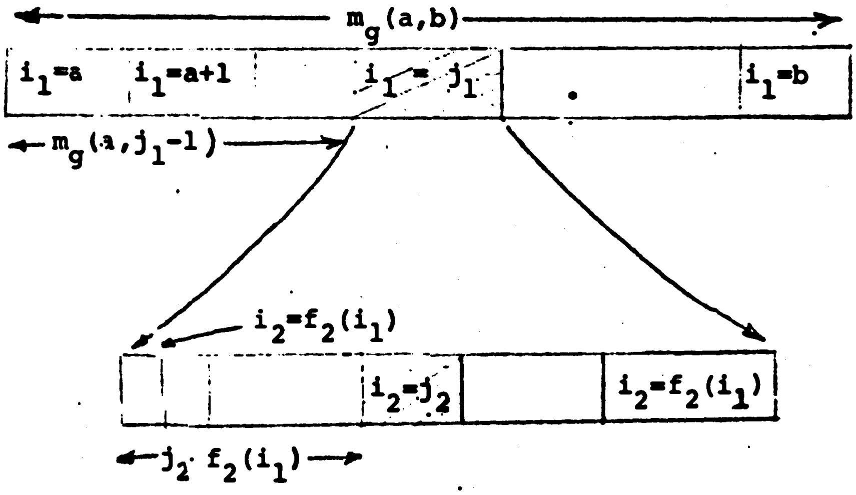
$$
\begin{aligned}
I &= m_g(a,i-1) + j - f(i) \\
&= N(i-1) - N(a-1) + j - (k_n i^n + k_{n-1} i^{n-1} + .. + k_1 i + c) \\
&= N'(i) - N(a-1) + j.
\end{aligned}
$$

$N'(i)$ is a polynomial of degree $n+1$. The cost of evaluation I is therefore $n+1$ multiplications since $N(a-1)$ is a constant.

If $n=0$ then our grid is equivalent to a 2-dimensional array. The cost of evaluating an index then is one multiplication.

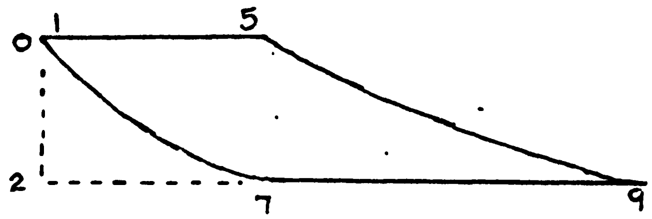
PICTORIAL VIEW OF ACCESSING A GRID ELEMENT $g[j_1,j_2]$

IN ITS LINEAR REPRESENTATION.



Example

var g:grid[i in 0..2, 1+i+i**2..5+2*i+2*i**2] of integer



Case 3.

var g:grid[i_1 in a..b, i_2 in $f_2(i_1)..g_2(i_1)$, $f_3(i_1,i_2)..$
$g_3(i_1,i_2)$] of s;

where

1. a,b are integer and $a \leq b$ for the declaration to be non-vacuous.
2. i. $f_2(i_1) \leq g_2(i_1)$ $\quad a \leq i_1 \leq b$

   ii. $f_3(i_1,i_2) \leq g_3(i_1,i_2)$ $\quad a \leq i_1 \leq b$ and $f_2(i_1) \leq i_2 \leq g_2(i_1)$
3. $f_2(x)$ and $g_2(x)$ are integer valued functions linear in x and $f_3(x,y)$ and $g_3(x,y)$ are integer valued functions linear in x and y. They have the form

   $f_2(x) = h_{21}x+c_2$
   $g_2(x) = k_{21}x+d_2$
   $f_3(x,y) = h_{31}x+h_{32}y+c_3$
   $g_3(x,y) = k_{31}x+k_{32}y+d_3$

Condition 2.i is true if $g_2(i_1)-f_2(i_1) \geq 0 \quad i_1=a,b$ i.e.

$$(k_{21}-h_{21})a+(d_2-c_2) \geq 0$$

$$(k_{21}-h_{21})b+(d_2-c_2) \geq 0$$

Condition 2.ii is true if $g_2(i_1,i_2)-f_2(i_1,i_2) \geq 0 \quad i_1=a,b$
$i_2=f_2(i_1),g_2(i_1)$

i.e. $(k_{31}-h_{31})i_1+(k_{32}-h_{32})i_2+(d_3-c_3) \geq 0 \quad i_1=a,b$
$i_2=f_2(i_1),g_2(i_1)$

i.e.
$$\left.\begin{array}{l}(k_{31}-h_{31})i_1+(k_{32}-h_{32})f_2(i_1)+(d_3-c_3) \geq 0 \\ (k_{31}-h_{31})i_1+(k_{32}-h_{32})g_2(i_1)+(d_3-c_3) \geq 0\end{array}\right. \quad i_1=a,b$$

i.e.
$$(k_{31}-h_{31})a+(k_{32}-h_{32})f_2(a)+(d_3-c_3) \geq 0$$
$$(k_{31}-h_{31})b+(k_{32}-h_{32})f_2(b)+(d_3-c_3) \geq 0$$
$$(k_{31}-h_{31})a+(k_{32}-h_{32})g_2(a)+(d_3-c_3) \geq 0$$
$$(k_{31}-h_{31})b+(k_{32}-h_{32})g_2(b)+(d_3-c_3) \geq 0$$

In all we have to make 6 tests to make sure that conditions 2.i and 2.ii are true.

Again we want to find

i. The minimum total storage $m_g(a,b)$ required to represent g.

ii. A representation for g

iii. How to access $g[i_1,i_2,i_3]$

For the reason given before we will limit our calculations to the case when a and b are both ≥ 0.

Let $x_{ij}=k_{ij}-h_{ij}$ $\quad i=2,3$

$y_i=d_i-c_i+1$ $\quad j=1,2$

$$m_g(a,b) = \sum_{i_1=a}^{b} \sum_{i_2=f_2(i_1)}^{g_2(i_1)} [g_3(i_1,i_2)-f_3(i_1,i_2)+1]$$

$$= \sum_{i_1=a}^{b} \sum_{i_2=h_{21}i_1+c_2}^{k_{21}i_1+c_2} [x_{31}i_1+x_{32}i_2+y_3]$$

$$= \sum_{i_1=a}^{b} (x_{31}i_1+y_3)(x_{21}i_1+y_2)+ \sum_{i_2=h_{21}i_1+c_2}^{k_{21}i_1+d_2} [x_{32}+i_2]$$

Now $$\sum_{i_2=h_{21}i_1+c_2}^{k_{21}i_1+d_2} [x_{32}i_2] = x_{32} \sum_{i_2=0}^{k_{21}i_1+d_2} i_2 - \sum_{i_2=0}^{k_{21}i_1+c_1-1}$$

$$= \frac{x_{32}}{2} [(k_{21}i_1+d_2)(k_{21}i_1+d_2+1)-(h_{21}i_1+c_2)(h_{21}i_1+c_2-1)]$$

$$= \frac{x_{32}}{2} [(k_{21}^2-h_{21}^2)i_1^2+k_{21}(d_2+1)+k_{21}d_2-h_{21}(c_2-1)-h_{21}c_2i_1 + d_2(d_2+1)-c_2(c_2-1)] .$$

We collect the coefficients of i_1^k in $m_g(a,b)$'s right hand side to get

$$m_g(a,b) = \sum_{i_2=h_{21}i_1+c_2}^{k_{21}i_1+d_2} \Big\{[\frac{x_{32}}{3} (k_{21}^2-h_{21}^2)+x_{31}x_{21}]i_1^2$$

$$+ [\frac{x_{32}}{2} k_{21}(d_2+1)+k_{21}d_2-h_{21}(c_2-1)-h_{21}c_2+ y_3x_{21}+y_2x_{31}]i_1$$

$$+ [\{d_2(d_2+1)-c_2(c_2-1)\}\frac{x_{32}}{2} + y_2y_3]\Big\}$$

Let $s_2 = \frac{x_{32}}{2}(k_{21}^2-h_{21}^2)+x_{21}x_{31}$

$s_1 = \frac{x_{32}}{2}[2k_{21}d_2+k_{21}-2h_{21}c_2-h_{21}]+y_3x_{21}+y_2x_{31}$

$s_0 = \frac{x_{32}}{2}[d_2(d_2+1)-c_2(c_2-1)]+y_2y_3$.

$$\rightarrow \quad m_g(a,b) = \sum_{i_1=a}^{b}[s_2i_1^2+s_1i_1+s_0]$$

$$= s_2[\sum_{i_1=0}^{b} i_1^2 - \sum_{i_1=0}^{a-1} i_1^2] + s_1[\sum_{i_1=0}^{b} i_1 - \sum_{i_1=0}^{a-1} i_1]$$

$$+ s_0(b-a+1)$$

Let $M(x) = s_2\sum_{y=0}^{x} i_1^2 + s_1\sum_{i_1=0}^{x} i_1 + s_0x$

$$\rightarrow \quad m_g(a,b) = M(b) - M(a-1)$$

Note $\quad M(x) = s_2[\frac{x(x+1)(2x+1)}{6}] + s_1[\frac{x(x+1)}{2}] + s_0x$

ii. The grid g is represented by the array

<u>array</u> $[0..m_g(a,b)-1]$ <u>of</u> s

iii. $g[j_1,j_2,j_3]$ is a valid grid element if the following are true.

1. $a \le j_1 \le b$
2. $f_2(j_1) \le j_2 \le g_2(j_1)$
3. $f_3(j_1,j_2) \le j_3 \le g_3(j_1,j_2)$

The index I of $g[j_1,j_2,j_3]$ in the linear array is given by

$$I = m_g(a,j_1-1) + \sum_{i_2=f_2(j_1)}^{j_2-1} [g_3(j_1,i_2)-f_3(j_1,i_2]+1]+j_3-f_3(j_1,j_2)$$

Now $\sum_{i_2=f_2(j_1)}^{j_2-1} [g_3(j_1,i_2)-f_3(j_1,i_2)+1] = \sum_{i_2=f_2(j_1)}^{j_2-1} [x_{31}j_1+x_{32}i_2+y_3]$

$$= (x_{31}j_1+y_3)(j_2-h_{21}j_1-c_2) + \sum_{i_2=h_{21}j_1+c_2}^{j_2-1} x_{32}i_2$$

$$= (x_{31}j_1+y_3)(j_2-h_{21}j_1-c_2)+\frac{x_{32}}{2}[j_2(j_2-1)-(h_{21}j_1+c_2)(h_{21}j_1+c_2-1)]$$

$$= j_1^2(-x_{31}h_{21})+j_1(-c_2x_{31}-y_3h_{21})+j_2(x_{31}j_1+y_3)-c_2y_3 + \frac{x_{32}}{2}[j_2^2-j_2-h_{21}^2j_1^2+j_1h_{21}(2c_2-1)+c_2(c_2-1)]$$

$$= j_1^2[-x_{31}h_{21}-\frac{x_{32}}{2} h^2]+j_1[-c_2x_{31}-y_3h_{21}+\frac{x_{32}}{2} h_{21}(2c_2-1)] + \frac{x_{32}}{2}j_2^2+j_2[x_{31}j_1+y_3-\frac{x_{32}}{2}]-[c_2y_3+\frac{x_{32}}{2} c_2(c_2-1)]$$

Using the fact that $f_3(j_1,j_2) = h_{31}j_1+h_{32}j_2+c_3$ we get

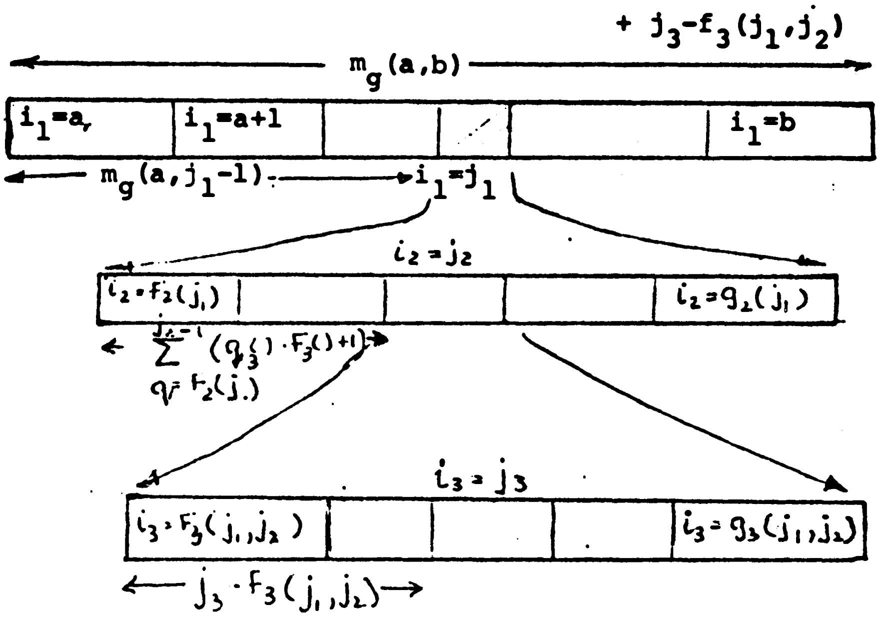
$$I = m_g(a,j_1-1) + j_1^2[-x_{31}h_{21} - \frac{x_{32}}{2} h_{21}^2]$$
$$+j_1[-c_2x_{31} - y_3h_{21} - h_{31} + \frac{x_{32}}{2} h_{21}(2c_2-1)]$$
$$+\frac{x_{32}}{2}j_2^2 + j_2[x_{31}j_1 + y_3 - \frac{x_{32}}{2} - h_{32}]$$
$$-[c_2y_3 + \frac{x_{32}}{2} c_2(c_2-1) + c_3] + j_3$$

To calculate the index I we require 6 multiplications.

A PICTORIAL VIEW OF ACCESSING THE

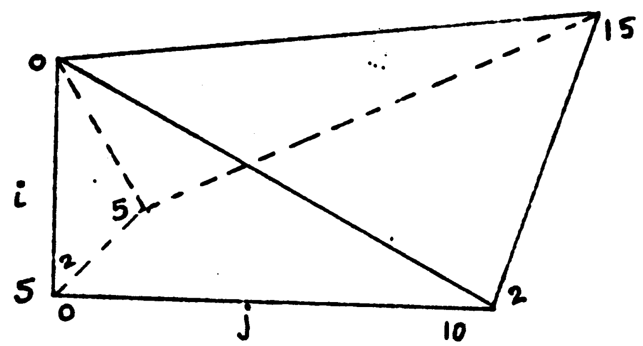
GRID ELEMENT $g[j_1,j_2,j_3]$

$$I = m_g(a,j_1-1) + \sum_{q=f_2(j_1)}^{j_2-1} [g_3(j_1,q) - f_3(j_1,q) + 1]$$



Example

var g:**grid**[i **in** 0..5, j **in** 0..2*i, 2..i+j] **of** real



APPENDIX B

Modifications to the Pascal Syntax to include grids

The productions of Pascal that are modified are preceded by a '*' while those that have been added are preceded by a '+'.

Note {a} means a may be repeated as many times as disued

{a} means a may be used at most once.

$\begin{bmatrix} a_1 \\ \cdot \\ \cdot \\ a_n \end{bmatrix}$ means use only one of $a_1,\ldots,a_n$

* <unpacked structured type>::= <array type>|<record type>|
<set type>|<file type>|<grid type>

\+ <grid type>::= grid[<sparse or regular parts>]
{<subgrid modifier>}^{1} of <component type>

\+ <sparse or regular parts>::=<sparse or regular part>
{; <sparse or regular part>}

\+ <sparse or regular part>::=<sparse part>|<regular part>

\+ <subgrid modifier> := with $\begin{bmatrix} \text{subgrids} \\ \text{subgrids} \end{bmatrix}$ <sbgrid>{;<sbgrid>}

\+ <sbgrid>::=<identifier>[<piece or name>
{; {minus}^{1} <piece or name>}]

\+ <piece or name>::=<piece>|<identifier>

```
+    <piece>::=<bounds> {,<bounds>}

+    <bounds>::={<identifier> in}^1 <simple bounds>

+    <simple bounds>::=<expression>..<expression>|*

+    <sparse part>::= sparse
                <positive part> with dense[<positive part>;
                                  s;<positive part>}]

+    <positive part>::={<identifier>}^1 [<piece> {;{minus}^1
                                       <piece>}]

+    <regular part>::=<positive part>|{minus}^1 <piece>
```

A restricted actual parameter may be used instead of an actual parameter.

```
     <restricted actual parameter>::=<identifier> {readonly
             [<piece or name> {;<piece or name>}]}^1
            { mask[<piece or name>{;<piece or name>}]}'

*    <repetitive statement>::=<while statement>|<repeat statement>
                |<for statement>|<general for statement>

+    <general for statement>::= for <identifier>
                {,<identifier>} in <set of values>
                                          do <statement>
```

To be able to iterate over a grid, the set of values is supplied in the form:

```
+    <set of values>::={{rev}^1 [row] }^1 domain(
                                [col]
                <grid expression>{,{minus}^1<grid expression>})
```

```
+    <grid expression>::=<identifier>{.<identifier>}1
                                      {(<sub list>)}'

+    <sub list>::=<sub> {,<sub>}

+    <sub>::=<expression>{..<expression>}' | *
```

APPENDIX C

The Fortran Grid Syntax

Self explanatory and common non-terminals have been left undefined. The meta-symbols [ ], { } and { }^{1} have the same meaning as defined in appendix B.

```
<grid declaration>::=<type> GRID [ <identifier>        ]
                                 [ (<identifier list>) ]
          (<sparse or regular parts>) <subgrid modifier>

<identifier list>::=<identifier> {,<identifier>}

<type>::= REAL|INTEGER|DOUBLE PRECISION|COMPLEX|...

<sparse or regular parts>::=<sparse or regular part>
                            {,<sparse or regular part>}

<subgrid modifier>::= WITH [SUBGRID ] <sbgrid> {;<sbgrid>}
                           [SUBGRIDS]

<sbgrid>::=<identifier> (<piece or name> {;{MINUS}
                                   <piece or name>})

<piece or name>::=<piece>|<identifier>

<piece>::=<bounds> {,<bounds>}

<bounds>::={<identifier> in }1 <simple bounds>

<simple bounds>::=<integer expression>:<integer expression>

<sparse part>::= SPARSE <positive part> WITH DENSE(<positive
                            part>{;<positive part>})
```

<positive part>::={<identifier>}^{1} (<piece>{;{MINUS} <piece>})

<regular part>::=<positive part>|{MINUS} <piece>

A restricted argument may be used for an ordinary one in a function or procedure call.

<restricted argument>::=<identifier> {RDONLY(<piece or name> {;<piece or name>})}^{1} {MASK(<piece or name>{;<piece or name> })}

The DO statement that generates the grid indices i.e. grid iteration statement, has the following syntax:

<grid iteration statement>::= DO <statement#><index variable list>={ {DEC}^{1} $\begin{bmatrix}\text{ROW}\\ \text{COL}\end{bmatrix}$ }^{1} DOMAIN(<grid expression> {,{MINUS}^{1} <grid expression>})

<index variable list>::=<integer variable> {,<integer variable>}

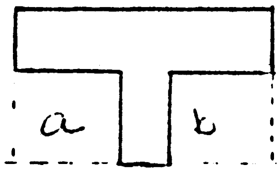
In case the key words ROW or COL are left out, then ROW is assumed. We have used DEC instead of REV in the Pascal syntax.

APPENDIX D

Implementation of Grids in Fortran

A restricted version of grids has been implemented in Fortran by the aid of a preprocessor written in PL/C, a Cornell implemented subset of PL/I. The preprocessor takes a Fortran program with grids and translates it inot an equivalent Fortran program without grids. Functional bounds, sparse parts and data security were not implemented. All grids were represented by the SEA.

For comparison of Fortran/g and Fortran itself a program written by M. Remorenko [10], (referred to as F from now on) to solve the fluid flow over an urban street canyon was taken and modified to include grids (the modified program is referred to as FG from here on). The grid required in the program F had the following shape:

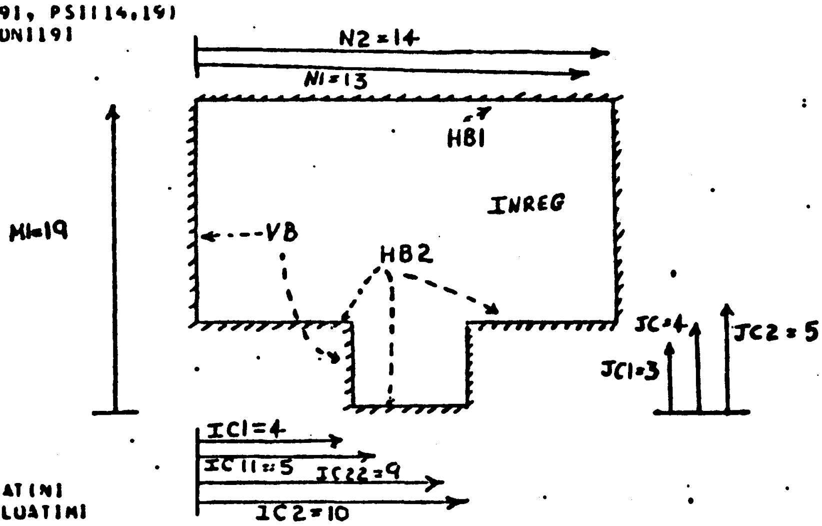


Remorenko used the SEA to represent the grid. When iterating over the SEA he used conditionals (8 tests in all) to make sure he avoided the regions a and b. While modifying F it was noticed that at one place the conditional were not written properly and regions a and b were not excluded from the computation. Usually computation on the boundaries included the end points but occasionally the end points were missed erroneously. It was fortunate that these errors had no bearing on the results.

The translation of FG produced by the preprocessor took up 15% more core during execution but it ran in 5% less time. This was because only one test (with the special value β) was required to stay out of the regions a and b while in F eight tests were required. (Had Remorenko used our way of representing grids, then his program would have executed in less time). There is much room for optimization of the code produced by the preprocessor.

Remorenko's program F is listed below along with the modifications to produce FG on the right.

```
C  FLOW OVER THE URBAN STREET CANYON
C
C  THE CALCULATION SCHEME USES THE METHOD OF FROMM; CENTRAL TIME DIFFERE
C  CENTRAL DIFFERENCES FOR THE CONVECTION TERMS ( BASED ON THE PRODUCT O
C  AVERAGES BETWEEN GRID POINTS ), AND THE DUFORT-FRANKEL PATTERN FOR TH
C  DIFFUSION TERMS.
C
      DIMENSION Z1(14,19), Z2(14,19), Z3(14,19), U(14,19), V(14,19),
     1 UM(14,19), VM(14,19), PSI(14,19)
      DIMENSION P(19), DADN(19)
C  SET UP PARAMETERS
      PI = 3.141593
      RE = 100.
      UTOP = 1.
      RHL = 0.5
      RHHL = 2.
      RWL = 1.
      M = 18
      N = 12
      NTAU = 0
      TAUZER = 0.
      TAUNOM = 0.
      TAUMAX = 1.5
      FPRINT =1.5
      SAFE = 1.0
      JDATA = 0
      ITMAX = 10
      FPUNCH=TAUMAX
C
C  COMPUTE CONSTANTS
      DX = (1. + RWL)/FLOAT(N)
      DY = (RHL + RHHL)/FLOAT(M)
      DXSQ = DX*DX
      DYSQ = DY*DY
      IC1 = ((0.5*RWL)/(1.+RWL))*FLOAT(N) + 1.1
      IC2 = ((1.+0.5*RWL)/(1.+RWL))*FLOAT(N) + 1.1
      JC =  (RHL/(RHL+RHHL))*FLOAT(M) + 1.1
      JC1 = JC-1
      JC2 = JC+1
```



ARRAYS DECLARED BY STATEMENT MARKED 1 ARE REPLACED BY GRIDS DECLARED AS

```
REAL GRID(Z1,Z2,Z3,U,V,UM,VM,PSI)(A(1:14,4:19); B(4:10,1:3))
     WITH SUBGRIDS HB1(1:4,19:19); HB2(2:4,4:4; 10:13,4:4; 5:9,1:1);
                   VB(14:14,4:19;10:10,1:4);INREG(2:13,5:18;5:9,2:4)
```

```
      IC11 = IC1+1
      IC22 = IC2 - 1
      M1 = M + 1
      N1 = N + 1
      N2 = N + 2
      A1 = 1./DY
      A2 = 1./DX
      A3 = 1./DYSQ
      A4 = 1./DXSQ
      A5 = 2./DYSQ
      A6 = 2./DXSQ
      A3Z = A3/RE
      A4Z = A4/RE
      A5Z = A5/RE
      A6Z = A6/RE
      A7 = (A5Z + A6Z)
      A8 = 1./(A5+A6)
      A9 = 1./FLOAT(N)
      A11 = 0.5/DY
      A12 = 0.5/DX
      DTAU = SAFE/A7
      DTAU = 0.15
C
C  CALCULATE OPTIMUM RELAXATION FACTOR
      XMU=(A5*COS( PI/FLOAT(M-1)) + A6*COS( PI/FLOAT(N-1)))/(A5+A6)
      RELAX= 2./(1.+SQRT(1.-XMU**2))
      RELAX1 = 1. - RELAX
      RW= RELAX/(A5 + A6)
C
C  PRINT SOME PARAMETERS
      WRITE(6,1) XMU,RELAX,RELAX1,DX,DY
    1 FORMAT('1  MU =',F12.6,'   RELAX =',F12.6,'   RELAX1 =',F12.6,
     *  '      DX =', F10.6, '      DY =', F10.5,///)
C
C  INITIALIZE VELOCITIES, VORTICITY, AND STREAM FUNCTION
      DO 20 I=1,N2
      DO 20 J=JC,M1
      Z1(I,J) = -0.5
      Z2(I,J) = -0.5
```

replace by Z1.A = -0.5
Z2.A = -0.5
DO 20 I,J = DOMAIN (PSI.A)

```
      Y = DY*FLOAT(J-JC)
   20 PSI(I,J) = 0.25*Y*Y
      DO 21 I=IC11,IC22         | replace by   Z1.INREG(*,JC)= -0.25
      Z1(I,JC) = -0.25          |               Z2.INREG(*,JC)= -0.25
   21 Z2(I,JC) = -0.25          |
      DO 22 I=1,N2          |
      DO 22 J=1,JC1         |
      Z1(I,J) = 0.          |
      Z2(I,J) = 0.          |                 Z1.B = 0.
      U(I,J) = 0.           |                 Z2.B = 0..
      V(I,J) = 0.           |  replace by      U.B = 0.
   22 PSI(I,J) = 0.         |                  V.B = 0.
      ZWALL1 = -0.25
      ZWALL2 = -0.25
      IF(JDATA) 58,59,58
   58 READ(5,35) ((I,J,Z1(I,J),Z2(I,J),PSI(I,J),K=1,M1),L=1,M2)
   35 FORMAT(2I2,3E14.6)
   59 DO 40 I=1,N2              |
      DO 40 J=1,N1              |  replace by
   40 Z3(I,J) = Z2(I,J)         |
   41 TAU = TAUZER                          Z3 = Z2
      GPRINT = FPRINT
      GPUNCH=0.
      ITE = ITMAX
      KTRVEL = 40
      IF (JDATA.EQ.0)  GO TO 51
      GPRINT = TAU - TAUNOM + FPRINT
      GPUNCH = TAU - TAUNOM
   51 GO TO 501
C
C  START COMPUTATION LOOP
   52 TAU = TAU + DTAU
      KTRVEL = KTRVEL + 1
      GPRINT = GPRINT + DTAU
      GPUNCH = GPUNCH + DTAU
C
      NTAU = NTAU + 1
C  COMPUTE NEW VORTICITY FIELD
      WZ1 = (1. + DTAU*A7)
```

```
      WZ3 = (1. - DTAU*A7)
      W1 = WZ3/WZ1
      W1DTAU = DTAU/WZ1
      ZW1 = Z3(IC1,JC)
      ZW2 = Z3(IC2,JC)
      DO 80 JJ=2,M
      J = M-JJ+2
      DO 80 I=2,N1
      IF(I.LE.IC1.AND.J.LE.JC)  GO TO 80
      IF(I.GE.IC2.AND.J.LE.JC)  GO TO 80
      W2 = -(UM(I,J) - UM(I-1,J)) - (VM(I,J) - VM(I,J-1))
      W4 = A62 - UM(I,J)
      W5 = A62 + UM(I-1,J)
      W6 = A52 - VM(I,J)
      W7 = A52 + VM(I,J-1)
      IF(I.EQ.IC11.AND.J.EQ.JC)  GO TO 81
      IF(I.EQ.IC22.AND.J.EQ.JC)  GO TO 82
      GO TO 88
   81 Z2(IC1,JC) = Z3(1,1)
      GO TO 88
   82 Z2(IC2,JC) = Z3(N2,1)
   88 Z3(I,J) = W1*Z1(I,J) + W1DTAU*( W2*Z2(I,J) + W4*Z2(I+1,J)
     1 + W5*Z2(I-1,J) + W6*Z2(I,J+1) + W7*Z2(I,J-1))
   80 CONTINUE
      Z3(IC1,JC) = ZW1
      Z3(IC2,JC) = ZW2
C
C  UPDATE PSI ON UPPER BOUNDARY
      SUMZ = 0.
      DO 90 I=2,IC1
   90 SUMZ = SUMZ + Z3(I,JC)
      DO 91 I=IC11,IC22
   91 SUMZ = SUMZ + Z3(I,1)
      DO 92 I=IC2,N1
   92 SUMZ = SUMZ + Z3(I,JC)
      SUMZ = SUMZ - 0.5*(Z3(IC1,JC) + Z3(IC2,JC))
      ZBAR = SUMZ*A9
      SUMZ = 0.
      DO 93 I=1,N2
```

POINTS Z3(1,1) and Z3(N2,1) were used For non grid purposes. We shall use ZQ1 and ZQN instead.

replace by

```
      DO 92 I,J= DOMAIN(Z3.HB2)
      SUM  = SUMZ + Z3(I,J)
   92 CONTINUE
```

The limits 2,N1 had been used by Remerenko in

Replace by DO 93 I,J = DOMAIN(Z3.HB1)

```
   93 SUMZ = SUMZ + Z3(I,M1)     |   replace by         SUMZ = SUMZ + Z3(I,J)
      ZABAR = SUMZ*A9                                   93 CONTINUE
      DS = 0.5*DY/(DX*RE)
      DSU = 0.5*DX/(DY*RE)
      DWDN(1) = DS*(Z3(IC1,JC)-Z3(1,1))
      DO 94 II=2,JC                         we use ZQ1
      I = JC-II+2
   94 DWDN(I) = DS*(Z3(IC1,II) - Z3(IC1,I))
      DWDN(JC) = 0.
      DO 95 JJ=IC11,IC22
      I = JC+JJ-IC1
   95 DWDN(I) = DSU*(Z3(JJ,2) - Z3(JJ,1))
      K=JC+IC2-IC1
      DWDN(K) = 0.
      DO 96 JJ=2,JC1
      I = K+JJ-1
   96 DWDN(I) = DS*(Z3(IC22,JJ)- Z3(IC2,JJ))
      L = K+JC1
      DWDN(L) = DS*(Z3(IC22,JC)-Z3(N2,1))
      P(1) = 0.                             we use ZQN
      DO 97 I=2,L
   97 P(I) = P(I-1) - (DWDN(I) + DWDN(I-1))
      PINT = -0.5*DY*( (P(L)-P(1)) + (P(K)-P(JC)) )
      DO 98 I=1,JC
      II = L-I+1
   98 PINT = PINT + DY*(P(II) - P(I))
      PSITOP = PSI(2,M1) - DTAU*((PINT/(1.+RL)) + ((ZABAR-ZOBAR)/RE))
      DO 99 I=1,N2
   99 PSI(I,M1) = PSITOP | replace by   PSI.MB1 = PSITOP
C
C     CALCULATE NEW STREAM FUNCTION FIELD USING SUCCESSIVE OVER-RELAXATION
      DO 160 ITER = 1,ITMAX
      REFPSI = 0.
      DELPSI = 0.
      RELAX2 = RELAX1
      RW1 = RW
      IF(ITER.EQ.1)  RELAX2 = 0.
      IF(ITER.EQ.1)  RW1 = A8
      DO 155 J=2,M
```

```
      DO 155 I=2,N1                                  replace by
      IF(I.LE.IC1.AND.J.LE.JC)  GO TO 155              DO 155 I,J = DOMAIN(PSI.IN
      IF(I.GE.IC2.AND.J.LE.JC)  GO TO 155
      PP = PSI(I,J)
      PSI(I,J) = RELAX2*PSI(I,J) + RW1*( Z3(I,J) + A4*(PSI(I+1,J) +
     1 PSI(I-1,J)) +A3*(PSI(I,J+1) +    PSI(I,J-1)))
      PSIABS = ABS(PSI(I,J))
      IF(PSIABS  .GT.REFPSI)  REFPSI = PSIABS
      DEL = ABS(PSI(I,J) -PP)
      IF(DEL.GT.DELPSI) DELPSI=DEL
  155 CONTINUE
  152 ITE = ITER
C
C  SATISFY PERIODIC BOUNDARY CONDITION
      DO 159 J=1,M1                                  PSI (1,*) = PSI(N,*)
      PSI(1,J) = PSI(N1,J)
  159 PSI(N2,J) = PSI(2,J)           replace by      PSI (N2,*) = PSI(2,*)
      IF((DELPSI/REFPSI).LT.0.0001)  GO TO 500
  160 CONTINUE
  500 WRITE(6,605) TAU,DELPSI,ITER
  605 FORMAT(5X,'CONVERGENCE CHECK -- TIME =',1PE14.5,5X,
     1 'MAXIMUM DELTA PSI =',E14.6,5X,'NO. OF ITERATIONS =',I3)
      WRITE(6,606) NTAU,PSITOP,Z3(1,1),Z3(N2,1),
     1 PSI(IC11,JC),PSI(IC22,JC)
  606 FORMAT(1X,'NTAU =',I3,1X,'PSITOP =',1PE12.5,1X,'Z3(1,1) =',E12.5,
     1 1X,'Z3(N2,1) =',E12.5,1X,'PSI(IC11,JC) =',E12.5,1X,
     2'PSI(IC22,JC) =',E12.5,//)
C
C  PROGRAM TO COMPUTE VELOCITIES U AND V AT NODE POINTS
  501 DO 502 J=2,M
      DO 502 I=2,N1                      replace by  DO 502 I,J = DOMAIN(V.INI
      IF(I.LE.IC1.AND.J.LE.JC)  GO TO 502
      IF(I.GE.IC2.AND.J.LE.JC)  GO TO 502
      U(I,J) = A11*(PSI(I,J+1) - PSI(I,J-1))
  502 V(I,J) =-A12*(PSI(I+1,J) - PSI(I-1,J))
      DO 503 I=1,N2
      U(I,M1) = UTOP
      V(I,M1) = 0.           replace by  U.HB1= UTOP
      KK = JC                            V.HB1= 0
```

```
      IF(I.GT.IC1.AND.I.LT.IC2)  KK=1
      U(I,KK) = 0.
  503 V(I,KK) = 0.
      DO 504 J=JC2,M
      V(1,J) = -A2*(PSI(2,J) - PSI(1,J))
      U(1,J) = A11*(PSI(1,J+1) - PSI(1,J-1))
      U(M2,J) = A11*(PSI(M2,J+1) - PSI(M2,J-1))
  504 V(M2,J) = -A2*(PSI(M2,J) - PSI(M1,J))
C
C  PROGRAM TO COMPUTE INTER-NODE TRANSPORT VELOCITIES UM AND VM.
      DO 530 I=1,M2
      DO 530 J=1,M1
      IF(I.LT.IC1.AND.J.LT.JC)  GO TO 530
      IF(I .GT. IC2 .AND.  J .LT. JC) GO TO 530
      IF(J.EQ.M1)  GO TO 520
      VM(I,J) = (V(I,J) + V(I,J+1))*A11
  520 IF(I.EQ.M2)  GO TO 530
      IF(I.EQ.IC1 .AND.J .LE. 4) GO TO 530
      UM(I,J) = (U(I,J) + U(I+1,J))*A12
  530 CONTINUE
 1300 CONTINUE
C
C  COMPUTE VORTICITY ON UPPER AND LOWER BOUNDARIES
      DO 511 I=1,M2
      Z3(I,N1) = -A5*(PSI(I,N) - PSI(I,N1)) - 2.*A1*UTOP
      IF(I.GT.IC1.AND.I.LT.IC2)  GO TO 512
      Z3(I,JC) = -A5*(PSI(I,JC2) - PSI(I,JC))
      GO TO 511
  512 Z3(I,1) = -A5*(PSI(I,2) - PSI(I,1))
  511 CONTINUE
C
C  COMPUTE VORTICITY ON VERTICAL BOUNDARIES OF THE CAVITY
      DO 513 J=2,JC1
      Z3(IC1,J) = -A6*(PSI(IC11,J) - PSI(IC1,J))
  513 Z3(IC2,J) = -A6*(PSI(IC22,J) - PSI(IC2,J))
      Z3(1,1) = -A6*(PSI(IC11,JC) - PSI(IC1,JC))
      Z3(M2,1) = -A6*(PSI(IC22,JC) - PSI(IC2,JC))
C
C  SATISFY PERIODIC BOUNDARY CONDITION
```

U.HB2 = O.
V.HB2 = O.

replace by

```
   DO 520 I,J = DOMAIN(VM, MINUS VM.HB1)
      VM(I,J) = (V(I,J) + V(I,J+1))*A11
520   CONTINUE

   DO 530 I,J = DOMAIN(UM, MINUS UM.HB2)
      UM(I,J) = (U(I,J) + U(I+1,J))*A12
530   CONTINUE
```

```
      DO 514 J=JC2,M
      Z3(1,J) = Z3(N1,J)
  514 Z3(N2,J) = Z3(2,J)
C
C  EQUATE Z1(I,J) TO Z2(I,J) AND Z3(I,J) TO Z2(I,J) TO REINITIATE CALCUL
      DO 130 I=1,N2
      DO 130 J=1,M1
      Z1(I,J) = Z2(I,J)
  130 Z2(I,J) = Z3(I,J)
C
C
C  LOOP COMPLETE - - CHECK FOR OUTPUT
  237 IF(GPRINT.LT.FPRINT) GO TO 1000
      WRITE(6,600) TAU, DTAU, ITE
  600 FORMAT(///,5X,'FIELDS OF PSI, U, V, AND VORTICITY AT TIME =',
     * 1PE14.7,'  NEXT TIME STEP = ', E14.7, '  ITERATIONS =', I4)
      WRITE(6,601)
  601 FORMAT(//,5X,'PSI FIELD')
      CALL BLOCK(N2,M1,PSI)
      WRITE(6,602)
  602 FORMAT(//,5X,'U - VELOCITY FIELD')
      CALL BLOCK(N2,M1,U)
      WRITE(6,603)
  603 FORMAT(//,5X,'V - VELOCITY FIELD')
      CALL BLOCK(N2,M1,V)
      WRITE(6,604)
  604 FORMAT(//,5X,'VORTICITY FIELD')
      CALL BLOCK(N2,M1,Z2)
 1900 GPRINT = GPRINT - FPRINT
 1000 IF(GPUNCH.LT.FPUNCH) GO TO 1100
      GPUNCH = GPUNCH - FPUNCH
C
C  CHECK FOR RETURN TO START OF LOOP
 1100 IF(TAU.LT.TAUMAX)  GO TO 52
 1200 CONTINUE
  250 STOP
      END
      SUBROUTINE BLOCK ( N, M, DUMMY)
C
```

replace by Z3(1,*) = Z3(N1,*)
Z3(N2,*) = Z3(2,*)

replace by Z1 = Z2
Z2 = Z3

```
C  PRUGRAM TU PRINT BLOCKS OF OUTPUT DATA.
C
      DIMENSION DUMMY(4,19)
      NUMBER = (N-1)/7 +1
    2 FORMAT(2X,1PE13.4,6E16.4)
      DO 700 K = 1,NUMBER
      IND1 = 1 +7*(K-1)
      IND2 = 7*K
      WRITE(6,650)
  650 FORMAT (1H )
      DO 700 J = 1, M
      JJ = M+1-J
      IF(K.EQ.NUMBER) GO TO 690
                          WRITE(6,2)          (DUMMY(I,JJ),I=IND1,IND2)
      GO TO 700
  690                     WRITE(6,2)          (DUMMY(I,JJ),I=IND1,N )
  700 CONTINUE
      RETURN
      END
*DATA
```