

INTRODUCTION TO OREGANO

Daniel M. Berry†

General Electric Research and Development

Schenectady, New York

Abstract	172
Introduction	172
Need for Model	172
Some Features of the Model	173
Modes	173
Blocks, Declarations, Assignments, and Other Basics	174
Retention	177
Multiple Values	178
Labels	181
Procedures	182
Coroutines and Tasks	184
Coroutines	184
Tasks	186
Practicality	188
Conclusion	189
Acknowledgments	189
Bibliography	189

†Present address: Division of Applied Mathematics, Brown University,
Providence, Rhode Island 02912

a Coercive, Recursive,
Block Structured Language
with Deference to Reference,
Attention to Retention,*
and Unification of Information
and Control Structures,
and Which Has an Implementation Orientation
in Its Definition and Specification

Abstract

This paper introduces Oregano, a practical generalization of ALGOL 60. The semantic definition of the language is in terms of an information structure model for its implementation, the contour model. Some of the major features are emphasized, including that of retention (non-deallocation of still accessible cells). The contour model is briefly described as a cell-based, fixed program component model with a retentive deallocation scheme. Modes (data types) are described as cell templates. Then, blocks, declarations, assignments, and pointer handling are illustrated in terms of sequences of pictorial snapshots in the model. A wide variety of heterogeneous and homogeneous multiple values are described using the data structure models of their implementation. Labels and procedures, which can be called recursively, are generalized to the full status of values. Coroutines and tasks are introduced as simple extensions of procedure calls, and various synchronization devices such as locking and events are illustrated. Finally, the practicality and ease of use of the language are demonstrated.

*Pun effects here were developed in collaboration with Professor Peter Wegner.

Introduction

Oregano is a highly practical generalization of ALGOL 60 [22] which makes use of certain features of BASEL and ALGOL 68 [30, 2]. The design and semantic specification are based on an information (data) structure model [34, 35, 37, 33], which specifies semantics of linguistic features in terms of their implementations. Oregano is a block structure language with recursive procedures. It provides an infinity of modes (data types) including those of pointers, labels, and procedures. There is a uniform treatment of values and identifiers of any mode. Oregano has the usual declaration and assignment features as well as pointer handling operators. The language provides a wide variety of heterogeneous and homogeneous multiple values, differentiated by complexity of structure. In addition, there is unification of control structures such as labels, procedures, coroutines, and tasks (parallel routines). There is one aspect of Oregano which differentiates it from most other languages, which makes some of the above unification possible, and which follows from the information structure model; namely, the concept of retention: a cell is not deallocated until there is no access path left to the cell. We first discuss the need for and the basic details of the model. Then the features of Oregano will be presented.

Need for Model

The current state of the art of semantic definitions is best exemplified by the ALGOL 68 Report [30]

and the ULD PL/1 Definition [19]. These specifications are difficult to read, provide little intuitive basis for understanding the language, and provide almost no basis for feasible implementations. Neither the user, implementer, nor designer thinks in terms of copy rules, renaming, text modification, unique name generators, [38, 39] etc. The language Oregano is an attempt to halt this unfortunate trend by presenting a definition of the semantics of the language in terms of a cell-based, fixed program component, information structure model.

Such a model is certainly helpful, if not necessary, to the designer to assure consistency and the very possibility of ever implementing the language being designed. It is clear that certain restrictions of ALGOL 68 and of PL/1 stem from their probably intended stack implementations [7, 4], even though the reasons for the restrictions do not logically follow from the formal definitions. Quite obviously, an implementation model of this sort is necessary for the implementer, for he is faced with the actual machine and its data cells. The user can and does benefit greatly from the knowledge of such a model. He is able to see precisely what he is causing to happen. Once the model is clear, what must be said to give the computer the necessary information becomes intuitively clear, so that the mastering of the syntax is made easier by the presence of the model. Thus it is this author's opinion that this sort of definition can be used as a vehicle for understanding design of programming languages, can form the basis of a feasible efficient implementation, and has intuitive appeal to the user.

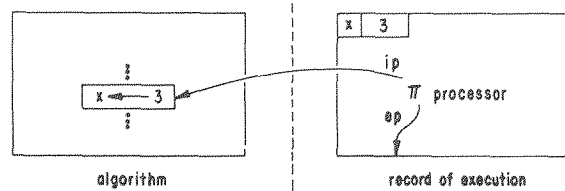
Some Features of the Model

The model used for the semantic definition of Oregano is J. B. Johnston's contour model [11, 12, 13, 6], a cell-based model which uses a fixed program component and a retentive scheme for deallocation.

The model and language specification recognize that the language is for specification of computations on a computer which stores its data in memory cells. Hence, the language specification will use the concept of a cell. Identifiers will designate the locations of cells. Assignments are made by changing the content of a designated cell. Pointers are values which are the locations of other cells. The implementation independent notions of possession, reference [30, 36], etc., used to describe the relation between identifiers and their values will be discarded as being unilluminating and unintuitive.

The model used consists of two components: the algorithm and the record of execution. The algorithm is a fixed re-entrant pure procedure copy of the program. The record of execution contains the variable data cells. It also contains several processors which are cells that control execution of the algorithm. Each processor has an instruction pointer, ip, pointing to an instruction in the algo-

rithm. Also each processor has an environment pointer, ep, which points to an accessing environment containing data cells in the record of execution. Execution of the algorithm is accomplished by having the instruction pointer scan the instructions in the algorithm step by step. When an identifier



(location designator) is encountered in the algorithm, the processor uses the environment pointer to develop the designator into the address of a cell in the record in which the value assigned to the identifier is stored [13, 36].

Once a cell is allocated in the record of execution it remains in existence so long as there exists a chain of pointers from some awake processor to the cell. This is known as the retentive deallocation scheme. (In an actual implementation this would be handled by reference count management [32] and/or garbage collection [20].) With this scheme for deallocation in the model, it becomes possible to specify retention in the language.

In the rest of this paper, we illustrate some of the features of Oregano using several example programs written in the language.* Each example is accompanied by a verbal and pictorial description of the execution of the program using the contour model. The invariant algorithm component is represented by one copy of the source language program with lines numbered for reference purposes. There is also a sequence of schematic diagrams of the state of the record of execution at various strategic points in the execution (after each source language statement, say). This sequence is called a sequence of snapshots of the record of execution and it corresponds to a sequence of states of an information structure as a result of transformations applied to the structure.

Modes

Oregano has an infinity of modes which are constructed both in the syntax and in the implementation


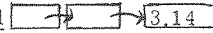
*The syntax for Oregano has not been fixed yet, but the intent is to have an easily used syntax which resembles that of well-known languages such as ALGOL 60, ALGOL 68, EULER, and PL/1. The syntax used in the examples should be clear either from the discussions or from familiarity with other languages.

model, from five basic modes and a variety of constructors. A mode can be thought of as a template which indicates the cell structure required to store values of the given mode. Below is a brief listing and description of some of the modes available in Oregano. In most cases, the cell structure and use of the mode will be discussed in detail in the appropriate section of this paper. In some simple cases, the cell structure is shown now so that we have some value types to work with in the examples. The cases marked with "*" are modes whose descriptions are outside the scope of this paper, but which nevertheless demonstrate the practical power of the language. The "... " after some of the mode names below indicates that the word itself is not a mode, and that the word represents a general class of modes, all of whose names begin with the word.

The five basic modes describe classes of values which may be stored in single cells:

Mode	Description	
<u>int</u>	integer numbers	-5975
<u>real</u>	real number	3.14E0
<u>bool</u>	boolean values, <u>true</u> or <u>false</u>	true
<u>char</u>	single character	'a'
<u>label</u>	statement label	

Others are:

- * compl complex numbers (pair of reals)
- ptr... a value of mode ptr amode (amode an arbitrary mode) is a pointer to a cell containing a value of mode amode
 e.g., ptr int 
ptr ptr real 
- proc... procedure
heterogeneous multiple values:
- tuple... fixed length--accessible only as a whole
- struct... fixed length--elements are selectable
homogenous multiple values:
- array... fixed length (at declaration time)
- seq... fixed length (at allocation time--distinct from declaration time)
- flex... flexible length
- * sparse... sparse matrix (à la Knuth [15])

- routine... name of a particular invocation (call) of a procedure
- event... event for communication between parallel routines
- * interrupt programmer defined interrupt and action upon interrupt
- * union... value which can be any of specified modes
- * pointer pointer to value of any mode, i.e., ptr union all...
- * basic value of one of five basic modes or of mode pointer (can be stored in one cell)

Oregano also provides for defining new modes in terms of existing modes by use of the word represent. For example, list rep struct {union [seq char, ptr list] car, ptr list cdr} defines the mode list. Given a value of mode list, the first element, selected by car, is either a sequence of characters or a pointer to another list, and the second element, selected by cdr, is a pointer to yet another list. Such a defined mode becomes a template just like any other mode.

In Oregano, modes are associated with identifiers in declarations. Execution of a declaration has a threefold effect:

1. A cell of the proper size for storing a value of the given mode is allocated.
2. The identifier designates this allocated cell.
3. Assignments to the identifier are restricted so that only values of the given mode are stored in the cell designated by the identifier.

Finally, it will be emphasized that, in Oregano, all values of any mode are treated uniformly with respect to assignment. Just as there is no restriction on assignment of an integer because of scope rules, there is no restriction to assignment of pointers, labels, and procedures because of scope rules. This freedom, which is not available in MUTANT 0.5, ALGOL 68, BASEL, PL/1, and EULER, [26, 30, 2, 19, 38] is a direct result of the retention assumption in Oregano.

Blocks, Declarations, Assignments, and Other Basics

We will use an example program as the vehicle for explaining some of the main features of Oregano: block entry and exit, declarations, scope, assignment, allocation, indirection and references, coercion, and retention.

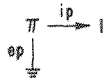
```

1  (int x, ptr int y;
2      x ← 1;
3  (int y ← 2;
4      x ← x + y;
5  )
6      y ← alloc 0;
7  (int x;
8      x ← 2;
9      y* ← x;
10     y ← @ x;
11 )
12     x ← y;
13 )

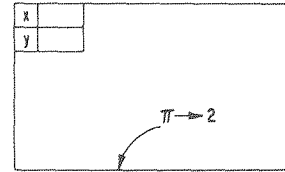
```

This program consists of an outer block (lines 1 through 13) declaring an integer x and a pointer to an integer y . (A block is delimited by a pair of parentheses.) The outer block contains two inner blocks, the first (lines 3 through 5) declaring an integer y , and the second (lines 7 through 11) declaring an integer x . The statement parts of each of these blocks contain several assignment statements, each using the assignment operator \leftarrow . Blocks serve to govern allocation of storage for identifiers and to define the scope in which an identifier is known. For example, the x declared in line 1 is known in the outer block minus the second inner block. It will be emphasized that exit from a block does not govern deallocation of cells for identifiers.

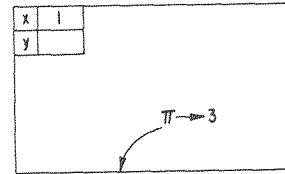
To start with, we have just a processor* with a null environment pointer, ep, and an instruction pointer, ip, pointing to line 1.



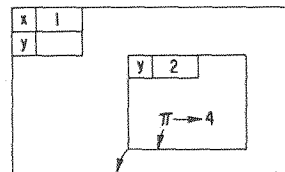
As a result of scanning line 1, the outer block is entered. A contour[†] is allocated with a declaration array containing cells for the declared identifiers, x and y . The ep of the processor is made to point to this contour so that the processor's environment consists of this contour. Conceptually we say that the processor is nested inside this contour. Also, the ip is moved to the next instruction in line 2. This sequencing to the next instruction occurs after each instruction except a goto and will not be dwelled upon any more.



In line 2, the assignment, $x \leftarrow 1$, causes a 1 to be stored into the cell designated by the identifier x .



In line 3, a new block is entered resulting in a new contour with a cell for a new y . The declaration int y ← 2 is an initializing declaration, so the cell for y is initialized to contain 2. The new contour is nested inside the old one by having the new contour's static link point to the old contour. The processor is then moved inside the new contour.

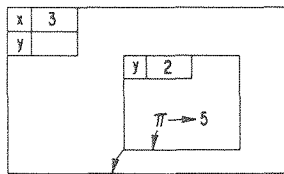


[†]Both a processor and a contour are cells. The processor is a cell whose subcells are copies of the processing units' registers. A contour is a cell, some of whose subcells are the so-called cells of the declaration array. We will refer to these subcells of the declaration array as cells while remembering that the contour and declaration array are all one cell.

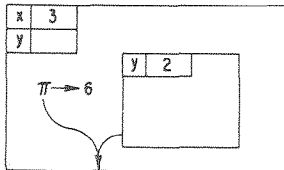
*See next column's footnote.

The assignment in line 4, $x \leftarrow x+y$, illustrates the use of local and nonlocal identifiers. The cell that is used is the innermost cell designated by the used identifier as one starts at the processor and works outward on the enclosing contours. The search path can be described as follows. Follow the ep of the processor to a contour and search its declaration array. If no cell for the desired identifier is found, follow the static link to the next contour and repeat the search until a cell is found or until the search fails.*

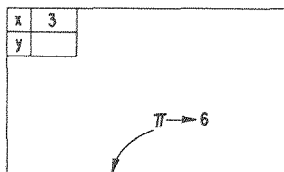
So in this example the y of the inner contour and the x of the outer contour are used to compute 3 which is stored in the cell for the outer contour's x . Observe that the usual scoping rules have been thus implemented and that the inner y shields the processor from using the outer y .



Then the first inner block is exited. The explicit effect of a block exit is to remove the processor from inside the current contour to inside the next outer contour. This is accomplished by resetting the ep of the processor to be the static link of the current contour. Of course the ip of the processor is sequenced to the next instruction.

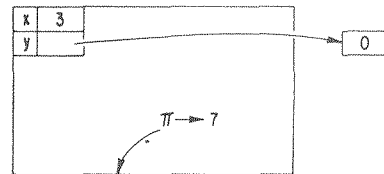


Observe that there is no access path from the executing processor to the inner contour. Hence, the processor can never regain the contour. This contour can thus be deallocated.

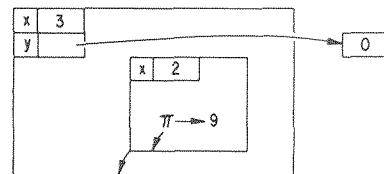


Note that deallocation has not taken place explicitly because the block was exited. Deallocation has taken place because the contour was no longer accessible by any path of pointers from the processor. In this case there were no paths left after block exit, but we will see cases where a path has been left and the contour is not deallocated at block exit.

Continuing with the example in line 6, the assignment $y \leftarrow \text{alloc } 0$ causes allocation of a cell initialized to the integer 0. A pointer pointing to this newly allocated cell, i.e., the address of the new cell, is returned for assignment to the cell for y .

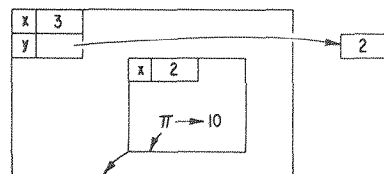


In line 7, a new block is entered resulting in creation of a new contour with a cell for a new declaration of x . Then in line 8, $x \leftarrow 2$ is executed and 2 is stored in this new cell for x .



When an assignment is to be performed, the target cell must be explicitly designated. The indirection operator, *, is used to designate cells pointed to by pointer values.

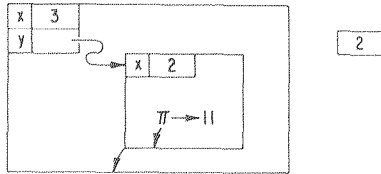
In line 9, $y^* \leftarrow x$, the value stored in the cell designated by x , that is 2, is stored in the cell designated by y^* .



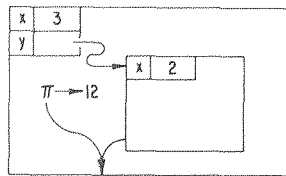
If the programmer wishes to point to an already existing cell, he uses the @ operator which returns the address of the cell designated by its operand.

*A syntactically correct program is guaranteed successful searches by the scope conventions.

The assignment $y \leftarrow @x$ in line 10 assigns to the cell for y a pointer pointing to the cell for x in the inner contour. Observe that the cell containing 2 which used to be pointed to by the contents of the cell for y is no longer accessible, so it is deallocated.

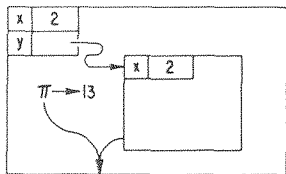


In line 11, the second inner block is exited, resulting in removing the processor from inside the inner contour.



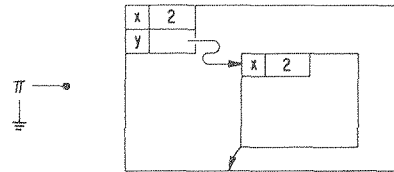
The inner contour has not been deallocated since the contour is still accessible via the pointer in the cell for y in the outer contour. The cell for x in the inner contour is no longer accessible by use of an identifier, but rather through indirection on y .

In line 12, the assignment $x \leftarrow y$ is executed by obtaining the integer value 2, pointed to by y and assigning it to the cell for x in the outer contour.



The value referred to by y was obtained by a coercion or automatic mode conversion called dereferencing. In Oregano, the only coercions allowed are those that obtain an already existing value. Conversions which compute new values such as integer to real conversion must be done explicitly. (Generic functions are used to allow mixing modes in expressions rather than conversion.) Furthermore, dereferencing and other coercions are performed only on the right-hand side of an assignment, where the needed mode can be figured out. Dereferencing does not occur on the left-hand side where the assignee must be explicitly designated.

Finally, in line 13 the outer block is exited and the processor is moved out into a null environment since the static link of the outer contour is null. The entire contour structure can be deallocated since there is no access path to the structure from the processor.



The processor terminates itself because there are no more instructions.



In the future, we will not always show the ep of the processor and the static links of contours since the topological nesting adequately symbolizes their use. However, the presence of these pointers is not to be forgotten, particularly in view of retention. Occasionally, for emphasis, these pointers will be explicitly shown.

Retention

In the previous example, we saw retention in operation. Consider this example which serves to isolate the effect of retention:

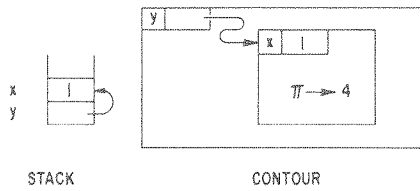
```

1  (ptr int y;
2  [ (int x = 1;
3  [   y ← @x;
4  [ ]
5  [   print (y*)
6  [ ] )

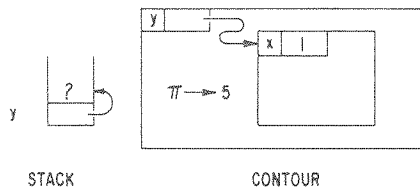
```

In most languages which have pointers as data types, e.g., MUTANT 0.5, ALGOL 68, PL/1, BASEL and EULER [26, 30, 14, 2, 4, 38, 6, 7], the equivalent of this program is illegal. These languages apparently are meant to be implemented in a stack model [34, 4, 14]. Thus, a cell for an identifier is allocated at entry to the block in which the identifier is declared, and this cell is released at exit from this block. Let

us compare the stack model and the contour model snapshots just before exit from the inner block:



After block exit, the top stack item is deleted leaving a pointer pointing to an nonexistent cell. The indirection on y in the print statement will fail. However, the contour model does not deallocate its corresponding cell (for x).



Retention is not present for declared identifiers in MUTANT 0.5, ALGOL 68, BASEL, PL/1, and EULER; retention is present in the heaps of BASEL and ALGOL 68 [14, 30, 20] as it is for cells in SLIP and LISP [21, 31, 32]. GEDANKEN has retention for all cells including those for declared identifiers, but the author expressed reservations [24] perhaps stemming from the lack of a good model. Oregano has retention for all cells and the retention follows quite logically from the model the specification is based on.

Multiple Values

Oregano has a rather comprehensive set of modes describing multiple values. There are two groups of these modes: one of multiples whose elements are of heterogeneous modes and the other of multiples of elements of homogeneous modes.

The two heterogeneous element multiples are tuples and structures [2, 14, 30]. They are both fixed length sequences of elements of possibly mixed modes. Tuples are accessible and assignable only as a whole, while structures have programmer defined selector names to allow working with individual components. The following illustrates the storage and manipulation of tuples and structures. The notation should be clear.

```

1  (tuple [real, int] t1, t2,
2      struct[real realpart, int intpart]s1, s2;
3      t1 ← [1., 2];
4      t2 ← t1;
5      s1 ← t2;
6      realpart of s2 ← 2.;
7      intpart of s2 ← 3 * intpart of s1;
8      -----

```

The following snapshot shows the configuration of storage after line 7 has been executed.

t1	1.	2
t2	1.	2
s1	1.	2
s2	2.	6

π → 8

Both structures and tuples are stored in precisely the same manner. A selector can be thought of as a constant index, so that application of a selector to a struct identifier can be completely resolved into a location designator at compile time.

The other group, that of collections of homogeneous values consists of three mode sets: arrays, sequences, and flexible lists. They are distinguished by the time at which the size (bounds) of the multiple is set and the flexibility of the length at run time. Arrays have bounds set at compile time. Sequences and flexible lists have their bounds set at allocation time. Arrays and sequences are fixed length after allocation, but flexible lists can be stretched, shrunk or inserted into. Storage for arrays is allocated at declaration time, but storage for sequences and flexible lists is allocated by execution of an explicit allocation instruction that returns a pointer pointing to the allocated cells.

Also, the mode of the elements may be any mode even other arrays, sequences, or flexes. The following illustrates some of the features of array, seq, and flex modes.

```

1  (array [1:5] int a,
2      array [1:5, 2:3] int aa
3      seq int s,
4      seq seq int ss,

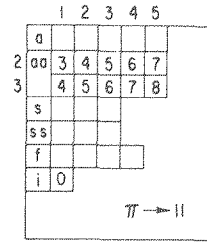
```



```

5      flex int f,
6      int i;
7      i ← 0;
8      for j ← 1 to 5 do
9          for k ← 2 to 3 do
10             aa [j, k] ← j+k;
11         a ← aa [ , 3];
12         s ← allocseq a;
13         ss ← allocseq
            (3 times allocseq(i ← i+1 times i));
14         ss [2] [1] ← 1;
15         ss [2] ← s;
16         f ← allocflex(5 times i ← i+1);
17         insert(allocflex [2]) after f [2];
18         conc (allocflex [1, 2]) to f;
19         -----

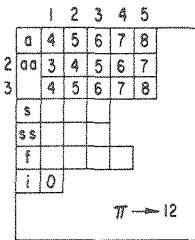
```



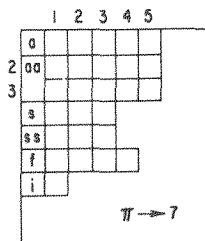
In the for statement, variable indexes have been used. The number of indexes, that is the dimensionality, can be checked for correctness against the mode at compile time. But, in general, there must be a run time check for validity of variable indexes. For arrays the value of the particular index is checked against the bound in the declaration. For sequences and flex's the bound stored in the descriptor is used for comparison against a possible index. In certain cases such as in the for loop where a variable is bound by the looping bounds a compiler might be able to forgo a run time check.

Line 11, a ← aa [, 3] causes copying of a sub-array of aa into the cells for array a, specifically, the 3-row of aa whose length is equal to that of a, and whose element mode is the same.

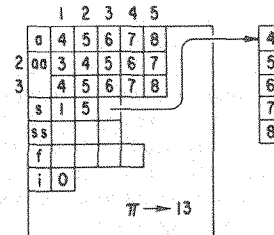
In lines 1 through 6 there are declarations of all of these types. An array can be of arbitrary dimension but its bounds must be integer constants. This allows the size of the space for the array in the declaration array to be computed at compile time. At declaration time only a descriptor cell for seq's and flex's is allocated in the declaration array. A descriptor's size may be computed at compile time because it consists only of spaces for the upper and lower bounds and pointers to the actual sequence or flexible list.



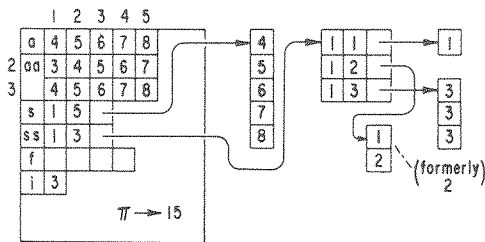
In line 12, s ← allocseq a allocates a sequence by creating cells initialized to a copy of a and returning to the cell for s both the pointer pointing to the newly created cells and the bounds of the sequence. Any array valued expression can be used as the argument of allocseq.



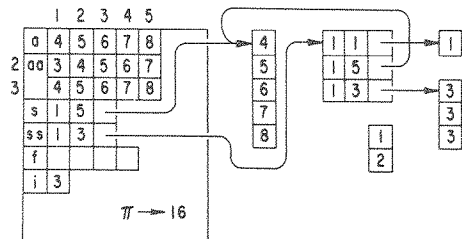
The for statement in lines 8, 9, and 10 assigns to each element of aa the sum of its indexes.



The identifier ss is of mode seq seq int; it can be assigned any sequence whose elements are all sequences of integers. It can be assigned a square matrix in tree form, but it is not restricted to a square form. Each of the elements of ss may be a different size sequence of integers. The assignment in line 13 allocates a triangular matrix to ss. (The expression n times x returns an array of n elements each of whose value is x.) The statement results in first allocating three sequences with successively increasing lengths and then assigning pointers pointing to these three sequences to three sequence descriptors, which are in turn allocated as a sequence of sequence descriptors. The pointer pointing to the sequence of sequence descriptors is returned to ss. Line 14 has an assignment to the first element of the second element of ss in ss[2][1] ← 1.



Furthermore, sequences need not be allocated all at once and in a nice pattern. A jagged array is easy to construct. Line 15 shows assignment of an arbitrary seq int as an element of a seq seq int: ss[2] ← s.

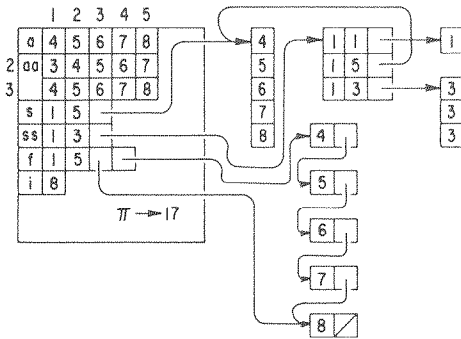


Observe that assignment results in copying that which is in the cell designated by the right-hand side. Since only the descriptor is in the declaration array only the descriptor is designated by s. The result of assignment is sharing of the array pointed to by the descriptor. When an array identifier is assigned to another array identifier as in line 11, there is copying. The entire array is in the declaration array so the entire array is designated. To copy a sequence one might write

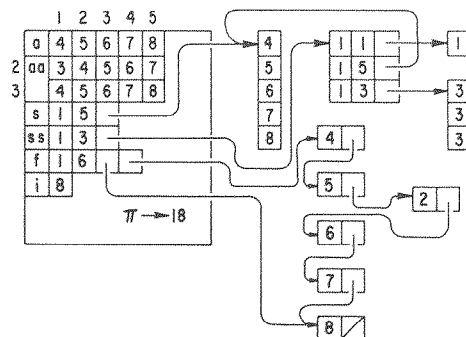
ss[2] ← allocseq s or ss[2] ← copy(s)

Notice also that passing a sequence as a parameter results in passing only the descriptor. Effectively the sequence is passed by reference. To simulate true by-value passing one has to pass a copy of the sequence.

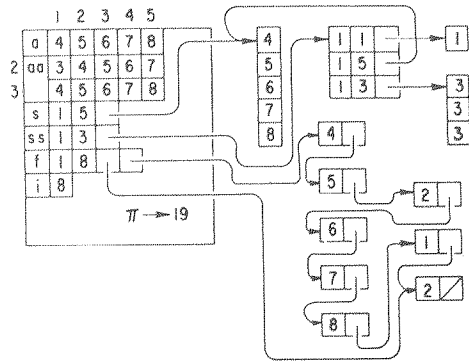
Flexible lists are allocated, assigned, and shared in the same manner as sequences are. Line 16 has the statement, f ← allocflex(5 times i ← i+1). A singly linked list of the elements of the array is created. The bounds and pointers to both the first element and the last element are returned to the cell for f.



The linked list arrangement allows stretching, shrinking, and insertion without reallocation of the entire array. Line 17 results in insertion of a newly allocated element initialized to 2 after the current second element. The upper bound is changed and the indexes of f[3] through f[5] are shifted one number up.



The concatenation of [1, 2] to f in line 18 makes use of the last element pointer to quickly locate the end of the flex where two new elements are added.



This has been a mere sampling of the multiple structures allowed in Oregano. The philosophy of design has been to design structures of increasing difficulty of implementation, to give different names to each structure, to tell the programmer what happens, and to allow him to pick the structure explicitly for his needs. Contrast this with ALGOL 68's scheme of classifying all of the homogeneous multiples under one mode, rows, with fixed and flexible bound options. The programmer has no idea of the expense he is causing when he uses flex.

Labels

In Oregano, labels are treated as any other value [16]. They can be assigned and can be passed as parameters; identifiers can be declared to be of mode label; and label identifiers follow the same scope rules of use as any other identifiers. An identifier which labels a statement is implicitly declared in the innermost block in which the labeled statement occurs; and the identifier is assumed to be initialized and made read-only at declaration time.

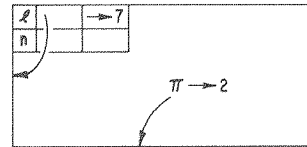
At any time, a processor is at some site of activity which is specified by pointers to each of the record and the algorithm in the form of the ep and the ip. A goto is thought of as moving the processor to a new site of activity, so a label must also consist of an ep and an ip. The following example illustrates label declarations, label values, and goto's.

```

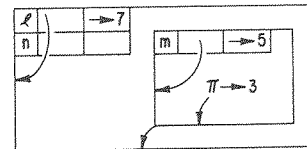
1  (label n;
2  (-----
3      n ← m;
4      goto l;
5      m: -----
6  )
7  l: -----
8  goto n
9  )

```

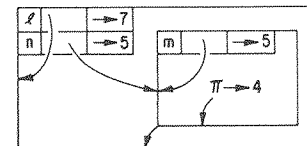
As the block beginning in line 1 is entered, a contour is created. There are cells for n, which is explicitly declared as a label, and for l, which is implicitly declared as a label. The cell for l is initialized to a label value consisting of an (ep, ip) pair. The ep of a statement label identifier is a copy of the ep of the processor at the assumed declaration time. The ip points to the labeled statement in the algorithm (represented by a pointer to the line number).



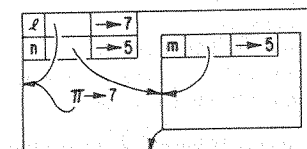
Continuing with the block entry in line 2, a new contour is created. The contour has a cell for the implicitly declared and initialized label m.



Then n ← m in line 3 causes copying the label value stored in the cell for m in the inner contour into the cell for n in the outer contour. The ep of the label value in n now points to the inner contour.

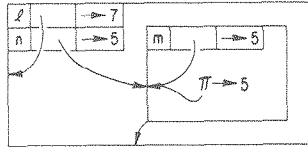


Then goto l is encountered. The label value in the cell for l is accessed. The ep and ip of the processor are made to be copies of the ep and ip of l, respectively. Execution continues in the new accessing environment with the instruction pointed to by the ip of the processor.



Since the ep of the label in n still points to the inner contour, this contour is retained. When a goto is performed, especially with a nonlocal label, there is a potential contour deallocation. However, contour deallocation happens not because the goto has exited the block, but because there is no access path from the processor to the contour.

Continue the execution with goto n at line 7. The processor now finds its site of activity back inside the inner contour and it is ready to execute line 5.



Procedures

Procedures are also treated with the same generality as other values in Oregano. Procedure values may be assigned and passed as parameters. The content of a procedure value is described in detail later.

Procedure variables may be declared. A procedure value has the additional property that when it is accessed, it gives rise to a "block" activation.

Let us consider an example with a rather simple procedure.

```

1      (proc [int] int p, int b;
2          (int a ← 0;
3              p ← (int i;
4                  a ← a + 1;
5                  a + i
6              );
7          (int a ← 2;
8              b ← p(2)
9          )
10     )
11     b ← p(2)
12     )

```

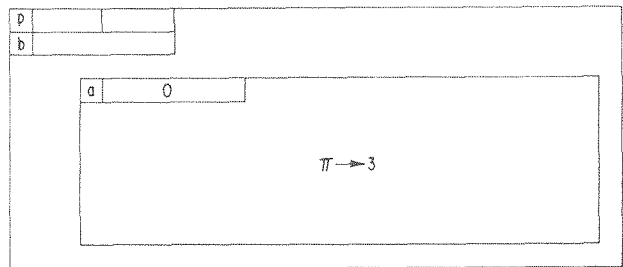
The identifier p is declared to be a procedure identifier which may be assigned the value of any procedure which accepts one integer parameter and returns one integer result. The assignment begin-

ning in line 3 is the assignment of a procedure literal. The literal matches the mode of p; it has one integer parameter i and it returns an integer result a+i. The textual scope rules extend to procedure bodies, so the nonlocal a is a use of the a declared in line 2.

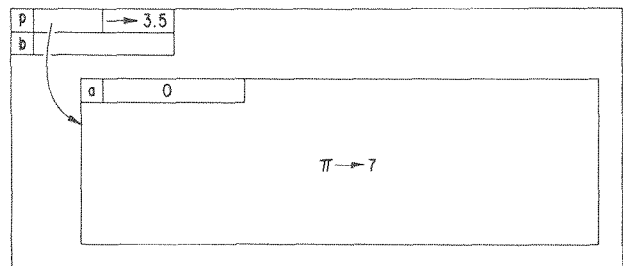
To call a procedure, we need two pieces of information: the text of the procedure body in the algorithm and the environment in the record of execution containing the proper cells for nonlocals. Hence, a procedure value is an (ep, ip) pair.

In Oregano, parameters are passed entirely by value. Other parameter initialization mechanisms, i. e., by reference and by name (procedure), are simulated by explicitly passing reference (pointer) and procedure values by value.

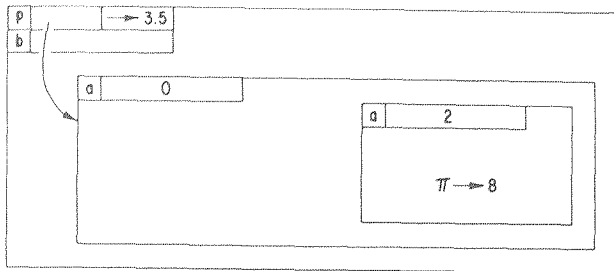
The block beginning in line 1 is entered, creating a contour with cells for p and b. Then another block is entered; a contour is created with a cell for a initialized to 0.



The assignment of the procedure literal to p in lines 3 through 6 causes computation of a procedure value and storage of this value into the cell for p. The ip points to the beginning of the text of the procedure in the algorithm, i. e., to the entry point in the middle of line 3 (represented as "→ 3.5"). The ep is a copy of the ep of the processor at this time, so the ep points to the current contour.

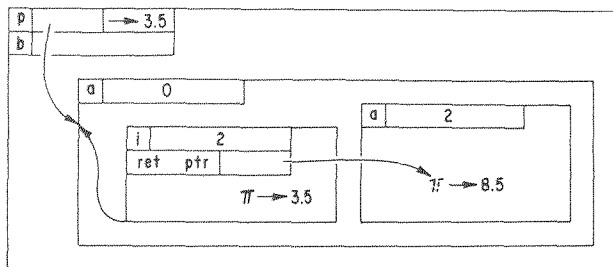


In line 7, a new block, which declares and initializes a to 2, is entered.



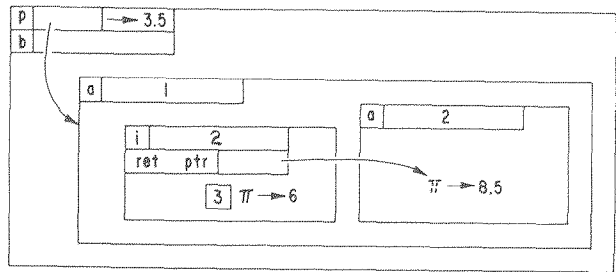
Now in line 8, there is a call of p. A contour is created with cells for both the formal parameter i and a return pointer. The cell for i is initialized to the value of the actual parameter 2. The cell for the return pointer is assigned a pointer pointing to the calling processor. The new contour is nested in the contour pointed to by the ep of the procedure value (the static link of the new contour is a copy of the ep of the procedure value). This places the contour of the call in the environment that has the cells for the nonlocals.

A new processor is created nested inside the new contour, i. e., its ep points to the new contour. The ip of the new processor points to the entry point of the procedure body which is obtained from the ip of the procedure value. Finally, the new processor is awakened and the calling processor is put to sleep. The sleeping calling processor, pointed to by the return pointer, has an ip pointing to the instruction to be resumed with at return time.

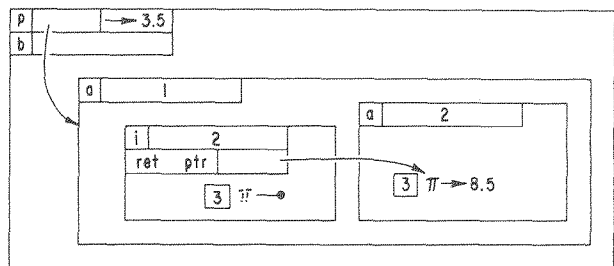


The nesting of the contour for a procedure inside the environment pointed to by the ep of the procedure value insures correct nonlocal linkage merely by the outward search algorithm that has already been used.

Now executing a = a + 1 in the body of the procedure at line 4, the value in a cell for a is incremented by 1. The cell used is the one corresponding to the a declared in line 2, in whose scope the procedure body lies. Then in line 5, the result of a + i, namely 3 is computed and stored in an accumulator accessible to the executing processor (of the call).

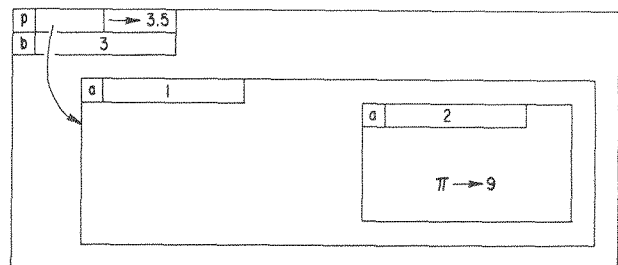


The end of the body is reached in line 6. So a return must be performed. The accumulated result is sent via the return pointer to an accumulator accessible from the sleeping processor at the site of call. The called processor is put to sleep and the processor pointed to by the return pointer is awakened. Now the executing processor is the one at the calling site.

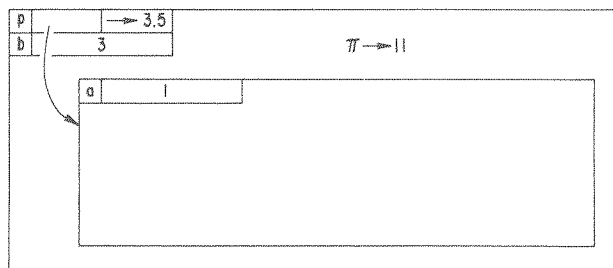


Since the asleep processor and the contour for the call are not accessible from the currently executing processor they may be deallocated.

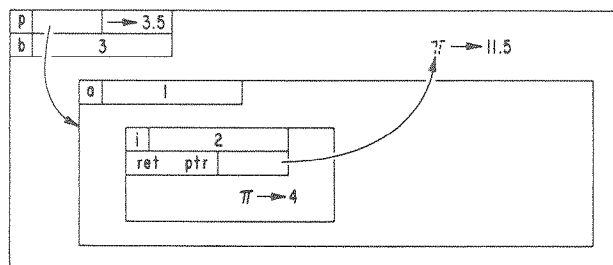
Now at the calling site, line 8, the assignment to b is completed. The result of the call, stored in the accumulator, is assigned to the cell for b.



A block is exited at line 9 moving the processor from inside the innermost contour; this contour is deallocated. Then in line 10, another block is left. The processor moves from inside the second to inside the outermost contour. However, since the second contour is pointed to by the ep of p, the second contour is retained.



Now at line 11, $p(2)$ is called. A contour is created with a cell for i initialized to 2 and a return pointer to the present processor. This contour is placed inside of the retained contour since the ep of p points to this contour. A processor is then created to control execution of the body; the called processor is awakened and the calling processor is put to sleep.



The body of the procedure is computed. The nonlocal a is incremented by 1 to 2. Then the result, 4, is returned, and assigned to the cell for b .

Observe that the cell for a is retained even though the block in which it is declared has been left and the cell for a is going to be accessible only from within an invocation of p . The a can be considered to be private to p (similar to ALGOL own, CPL and BASEL nonlocals [14, 2, 3]).

In closing the section on procedures, we briefly mention that any procedure whose activation environment includes its own identifier can be called recursively from within itself. One example of a recursive procedure is the standard factorial routine:

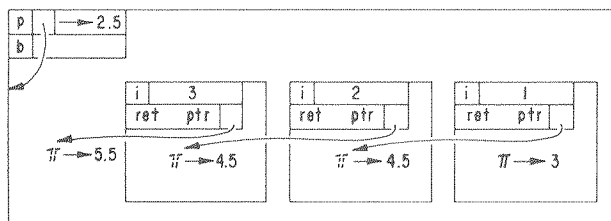
```

1  (proc [int] int p, int b;
2  [
3  [   p ← int i;
4  [   if i ≤ 1 then 1
5  [   else i * p(i-1) );
   b ← p(3) )

```

Since all calls on the same procedure use the ep of the same procedure value, all contours for recursive activations will be immediately nested inside the same contour and not in each other. The reader

should verify that the snapshot at the time of the third recursive call of p is like so:



It is clear that there will be no confusion as to which i should be used in any given activation, because the contours for the activations are disjoint.

Coroutines and Tasks

Oregano has facilities for coroutines and tasks (parallel routines). To help manipulate these control structures, two modes are provided: routine and event. Of course, identifiers having either of these modes follow the same scope rules as any others. Values having any of these modes may be assigned and passed as parameters just as any other value.

Coroutines

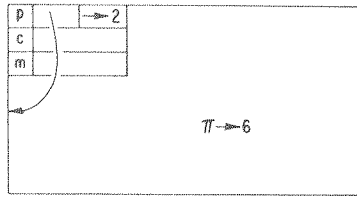
To control coroutines, identifiers and values of mode routine are used to name and retain a particular call of a procedure which is specified to be treated as a coroutine. The retention insures the ability to resume the same instance of call of a coroutined procedure. The techniques and various operators directed at coroutine implementation are illustrated in the following skeleton of a program (assume that the missing lines affect neither the sequencing nor the coroutine).

```

1  (proc p, routine c, m;
2  [   p ← < -----
3  [   resume m;
4  [   -----
5  [   >];
6  [   p coroutine c this m;
7  [   -----
8  [   resume c;
9  [   -----
10 [   quit c, m;
11 [   )

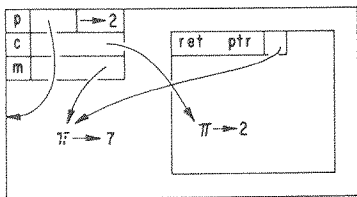
```

In line 1, the block declaring p, c, and m gives rise to a contour creation. Then in line 2, a procedure value is assigned to p. The body has been compiled as any other procedure literal.



In line 6, there is a call of p. However, there is additional information. The call is specified to be a coroutine call, and this particular call or invocation is given the name c. Furthermore, the calling routine, executing in the outer block, is given the name m. These names will be used with resume and in other communication between the two activities.

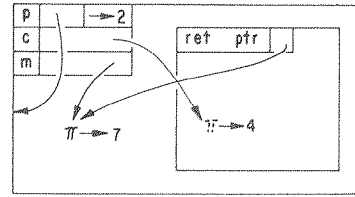
The effect on the record is as follows. A contour is created for the call nested in the contour that the ep of p points to. Parameters, if any, are initialized and a return pointer to the calling processor is set. A new processor to control the execution of the procedure body is nested in the new contour. Coroutine c results in assigning to c a pointer to the called processor. This m results in assigning to m a pointer to the calling processor. Finally, the called processor is awakened while the calling processor is put to sleep.



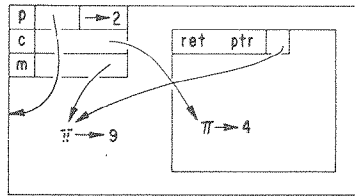
There appears to be very little difference between a coroutine call and a simple procedure call. The only difference is that a pointer to the called processor is assigned to an identifier of mode routine. Because of retention, the call may be left and the called processor and its environment will still be around for resumption.

Now in the body of the procedure, execution proceeds until resume m in line 3 is encountered. The operand of resume must evaluate to a value of mode routine which in turn is a pointer to some processor. Resume m merely causes putting the current (called) processor to sleep while awakening the processor pointed to by the value in m. The processor just put to sleep is left pointing to the

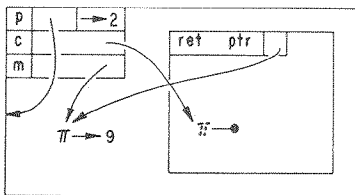
instruction to be continued with. Also, the called processor and the contour of the call are retained.



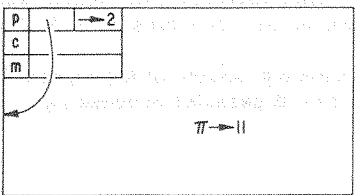
Now back in the main block, the m processor continues with the instruction after the call. In line 8, there is a resume c. The current (calling) processor is put to sleep and the processor pointed to by c is awakened.



Back in the body, the called processor continues with the instruction after the last resume, that is, in line 4. The execution finally arrives at the return bracket, >. The result is that the processor pointed to by the return pointer is awakened while the current (called) processor is put to sleep. It is as though a resume to the calling processor were executed. However, since there is no successor to the return instruction, >, the called processor terminates.



The calling processor finally comes to line 10 where quit c, m merely erases the pointers in c and m. The terminated called processor and the contour for the call can be deallocated.



The assumption of retention in the language, Oregano, makes coroutine handling a trivial extension of normal procedure calls. Note that the sharing of the environment between two coroutines have been quite elegantly taken care of by the normal scope rules combined with environmental nesting. In the literature, two V stacks have been used to implement coroutines [5, 33-p327] and special conventions and tricks have had to be used to obtain sharing.

The routine name serves to name a particular call of a procedure. To regain or otherwise use this call, the routine name is used. A use of the procedure identifier would generate a new independent call of the procedure. Thus, it is possible and quite easy for there to be recursive coroutines, a procedure call within a coroutine of the same procedure, etc., with no confusion as to which instance of a procedure call is which coroutine and as to which instances are plain procedure calls.

This flexibility also extends to tasks as we see that tasks are really another simple (albeit a little less so) extension of procedure calls.

Tasks

A tasked procedure call is quite similar to a plain call or a coroutine call. The main difference is that in the tasked call both the calling and called processors are awake. The order of execution of the statements of the calling and called routines is unpredictable with subsequent unpredictable side effects. A means of synchronizing the tasks is needed.

Procedures may be called with a task option and the resulting called activity optionally given a name of mode routine.

A task may be set at a given priority and a task can wait until an event has occurred [1, 4, 5, 8, 9, 27]. Of course, there must be facilities for locking and unlocking of resources to guarantee that only one processor may manipulate data in potentially simultaneously shared resources at any given time. These facilities are necessary for setting up waits and event notices so that they can be completed without interference. Also, the facilities for locking and unlocking should be made available to the programmer. Oregano will have these features. However, at this time the author has not decided on the implementation (and therefore the expression) of the features. Thus, this paper avoids the topic. (Several possibilities for implementation of locking and unlocking can be found in the literature [5, 8, 9, 27].)

The following outline of a program illustrates these features of parallel processing.

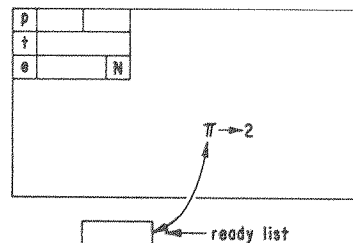
```

1  (proc(int)p, routine t, event e;
2      p ← (int i;
3          ----
4          ----
5          ----
6          ----
7          ----
8          cause e;
9          -----
10     );
11     p(1) task t priority 5;
12     wait e;
13     -----)

```

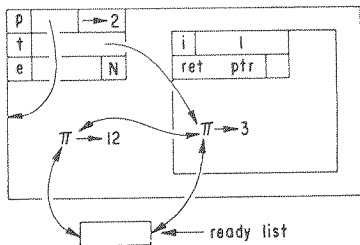
We introduce a new construct to the model which has always been present in the model but has so far been extraneous to the discussion. There is a doubly linked list of the awake processors called the ready list. The header of the ready list is known to the operating system, and the processors are maintained in the list in priority order. The scheduling algorithm will be made explicit only to the extent that processors on the ready list are the only ones which may be executed, and that as many of these as possible are executed given the number of processing units in the machine.

The block beginning at line 1 has declarations for a procedure p, a routine t, and an event e. The format of event cells will be discussed later, but at this time the event is set at "Not happened". The ready list is shown containing the processor which executes this block. To facilitate the linking on the list, each processor has a pair of pointer cells and other registers containing the state and other information. This processor is set at awake.



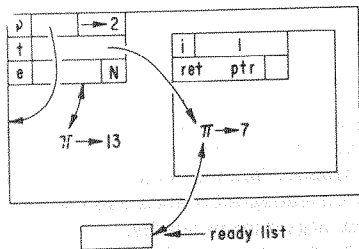
In line 2, the procedure value is assigned in the usual manner. Then in line 11, there is a call of p with a task option, p(1) task t priority 5. A contour

is created and nested in the contour pointed to by the ep of p. The cell for the parameter is initialized to 1. Because of the task option, the return pointer is set to nil. (The procedure body is coded in such a way that if the return pointer is nil, the called processor is to be terminated. One cannot predict at compile time whether or not a procedure literal may be called with the task option during execution. In fact, a given procedure may be used in both a plain call and a task call. So a general mechanism is needed.) A new processor to control the tasked call is created and nested in the new contour. A pointer to this new processor is assigned to the routine identifier \underline{t} . Finally, the new processor, hereinafter referred to as the called processor, is awakened and placed in the ready list according to the priority information given in the call statement. The calling processor is not put to sleep.



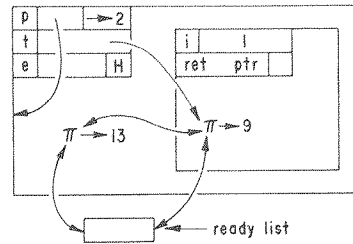
We pick an order of execution which is consistent with the tasking and which illustrates some points.

Assume that the calling processor comes to wait e in line 12. The processor is to wait until the event e has happened. The status of the event is checked; it has not happened, so the calling processor must be put to sleep awaiting the event e. The processor must somehow be linked to the event so that the event can signal the processor when it has happened. The processor is put to sleep and is removed from the ready list. The two pointer cells used for ready list linking are used to link the processor to a doubly linked list of all processors awaiting e. The result is that the calling processor is asleep ready to continue at line 13, and the called processor has been continuing to execute.

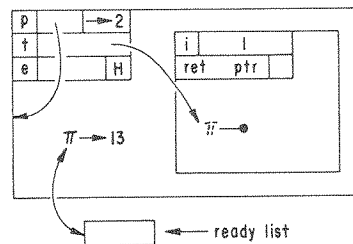


When the called processor scans cause e in line 8, the cell for e is accessed and its status is set to "Happened". The doubly linked list is used to visit

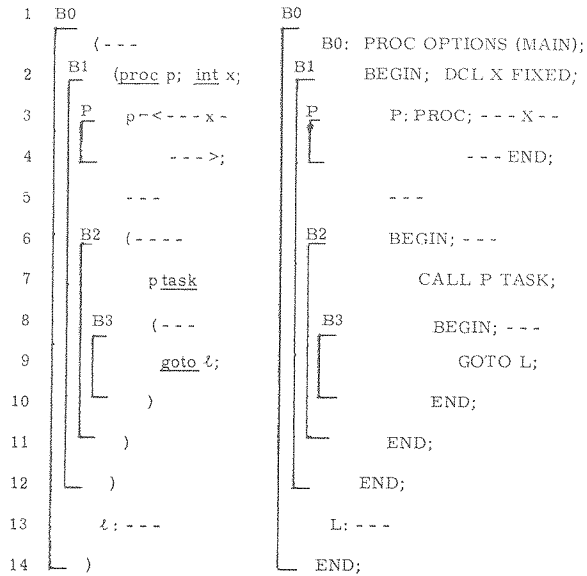
each processor waiting for event e. When the sleeping calling processor on the list is visited, it is awakened and put on the ready list. The freshly wakened processor continues with line 13 while the called processor goes on executing in the body. To allow an event to be reused, the programmer may reset an event to "Not happened" by writing reset e.



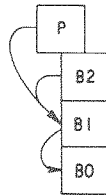
Finally, in line 10, the return bracket is encountered by the called processor. Since the return pointer is nil, the processor knows it should terminate itself. However, because of the pointer in the cell for \underline{t} , the terminated processor and the contour pointed to by its ep are retained. To get rid of these no-longer-useful structures, either a quit t can be executed to get rid of the pointer or else when p(1) was called as a task the routine identifier could have been left off--p(1) task priority 5. The purpose of using a routine name in the task case is to provide a means of communication between the processors in such statements as terminate t or priority (t)-7.



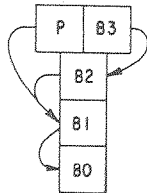
Because of the perspicuity of the model, tasking has been described and implemented as a rather simple extension of procedure calling. Furthermore, because of retention, the ad hoc PL/1 exceptions and abnormal terminations have been eliminated. Consider the Oregono program on the following page and its PL/1 counterpart. Notice the nonlocal x in p; x is declared in B1.



Just after the task is called, the usual PL/1 forked stack implementation [4, 19] looks like (the static links are shown):



Then, when the calling task enters block B3, the stack assumes a forked shape

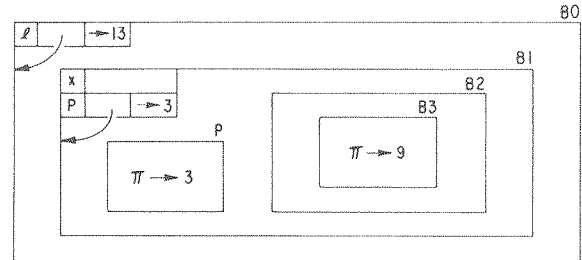


Suppose the goto l in block B3 is reached before execution of the task call of p has terminated. Since the label l is in block B0, transferring to l from in B3 has the effect of exiting blocks B3, B2, and B1. Thus, in normal PL/1 the storage for B3, B2, and B1 is deallocated. This leaves a disastrous gap in the stack; in particular, part of p's accessing environment, namely the x in B1, has disappeared.

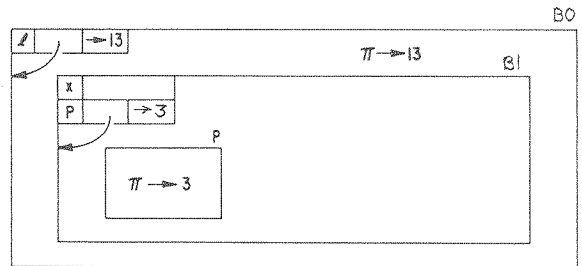


The PL/1 semantics saves the situation by saying that the task call of p is terminated abnormally.

In Oregono, with the contour model and retention, after the task p has been called and the calling processor has entered B3, we have



Now if the right-hand (calling) processor reaches the goto l before the other processor terminates, the right-hand processor merely moves out into the contour for B0



Only the contours for B2 and B3 can be deallocated. The contour for B1, containing the nonlocal x in p's accessing environment has been retained. The other processor can continue until it terminates normally.

Practicality

Oregono has been designed to be compiled into efficient object code rather than to be executed by interpretation. Three features are particularly helpful in this respect: static mode checking, static scope checking, and retention. The mode structure of Oregono is such that all mode compatibility may be checked for at compile time except in cases

involving modes union, pointer, and basic which are provided as escape clauses from static mode checking [14, 2, 30]. Programs that would otherwise terminate because of mode incompatibility are caught at compile time. Furthermore, the need for time consuming run time checks for mode compatibility has been eliminated. The invariance of the nested declaration structure in the algorithm and its preservation in the record of execution allow for compile time checking of scope validity and for generation of $\langle i, j \rangle$ pairs, i. e., \langle nesting height, displacement in declaration array \rangle pairs, as the object representation of identifiers. The $\langle i, j \rangle$ pairs are used in conjunction with a display [23, 11, 13] to provide rapid access of the contour of the proper height and the proper cell in the contour at run time.* The assumption of retention allows pointer, label, and procedure value assignments to be performed without a costly run time check (necessarily at run time [10]) on whether or not the assignment will result in a block exit at a time when a cell in the contour for the block is still pointed to.

Conclusion

This paper has presented the current state[†] of the development of Oregano. The primary vehicle for designing and explaining Oregano has been an information structure model describing its implementation. The model has allowed discovery, inclusion, and explanation of powerful new features such as retention. The model has helped in designing and describing the wide variety of multiple value structures. The model has assisted in unifying and simplifying the various control structures such as labels, procedures, coroutines, and tasks. It is hoped that the model oriented specification makes these features palatable, transparent, and easily applied by showing the user the precise effect of the execution of his programs. The definition used certainly makes Oregano easy to implement, for the definition is but a description of a feasible efficient implementation. Finally, the definition shown serves to help the computer scientist study these features with much more clarity than has been possible with other types of definitions.

Acknowledgments

The author is indebted to many people for their cooperation and assistance. Lynn Gimber, Steve Hobbs, John Hutchison, Philip Lewis, Clem McGowan, and Diane Rice were willing (?) sounding

*The display is not shown in this paper, but its application to the contour model is shown in Ref. 13. This paper has used an interpretative search which is equivalent to the display access method in picking a particular cell for an identifier [39].

†The complete description of Oregano will appear later as the topic of the author's doctoral thesis.

boards to my cries of "Eureka!" William Berry, Dan Rosenkrantz, and Mike Rubens consented to read earlier drafts of this paper; their slashing pencil marks were particularly helpful. Peter Wegner, in addition to doing the above, provided the inspiration to tackle the topic. This summer, John Johnston did all of the above, provided mind-blowing inspiration, hours of discussion, numerous suggestions, and encouragement to use his contour model. Finally, the author wishes to thank Liese Pfeiffer, Donna Rhodes, and Marion White for their perseverance in deciphering my writing to produce this typewritten paper.

BIBLIOGRAPHY

1. Alber, K. et al., Informal Introduction to Abstract Syntax and Interpretation of PL/1, IBM Lab., Vienna, TR 25.099 (June 30, 1969).
2. Applied Data Research Inc., Mass. Computer Assoc., Basel Language Programmer Manual, CA-7005-2011, Wakefield, Mass. (May 20, 1970).
3. Barron, D.W. et al., "The Main Features of CPL," Comp. J., 6, p. 134 (1963).
4. Beech, D., "A Structural View of PL/1," Computing Surveys, 2:1, p.33 (March 1970).
5. Bernstein, A.J. and Johnston, J.B., Implementation of a Parallel Processing Language, General Electric TIS Report 67-C-080 (March 1967).
6. Berry, D.M., Property Grammars, Basel, Contour Model, and Coercion, Brown Univ. (May 1970).
7. Berry, D.M., "The Importance of Implementation Models in Algol 68 or How to Discover the Concept of Necessary Environment," SIGPLAN Notices (Sept. 1970).
8. Cleary, J.G., "Process Handling on Burroughs B6500," Proc. 4th Australian Comp. Conf., p. 231 (1969).
9. Dennis, J.B. and Van Horn, E., "Programming Semantics for Multiprogrammed Computations," CACM, 9:3, p. 143 (March 1966).
10. Goos, Gerhard, "Some Problems in Compiling Algol 68", Rechenzentrum der Technischen Hochschule, Munchen, Germany; paper delivered to ACM SIGPLAN Algol 68 Symp. (June 1970).
11. Johnston, J.B., "The Contour Model of Block Structured Processes," Proc. ACM SIGPLAN Symp. - Data Structures and Programming Languages, Gainesville, Fla. (1971).

12. _____, "Structure of Multiple Activity Algorithms," Proc. Third Ann. Princeton Conf. on Info. Sci. and Systems, p.38 (March 1969).
13. _____, private communication on a model of the process concept (Spring 1970).
14. Jorrand, P. and Wegner, P., Some Aspects of the Structure of Basel, Brown Univ., Providence, R. I. (Jan. 1970).
15. Knuth, D. E., The Art of Computer Programming, Vol. I, Addison-Wesley (1968).
16. Ledgard, H.F., "Ten Mini-Languages in Need of Formal Definition," SIGPLAN Notices (April 1970).
17. Lewis, P.M. and Stearns, R. E., "Syntax Directed Transductions," JACM, 15:3, p. 465 (1968).
18. Lucas, P., Lauer, P. and Stigleitner, H., Method and Notation for the Formal Definition of Programming Languages, IBM Lab., Vienna, Tech. Report TR 25.087 (28 June 1968).
19. Lucas, P. and Walk, K., "On the Formal Description of PL/1"--Annual Review in Automatic Programming, 6:3 pp. 105-182 (1969).
20. Marshall, S., An Algol 68 Garbage Collector, TM0111, Dartmouth College (Dec. 1969).
21. McCarthy, J. et al., The LISP 1.5 Programmers Manual, MIT Press (1963).
22. Naur, P., ed. ; "Revised Report on the Algorithmic Language ALGOL 60," CACM, 6:1, pp. 1-23 (1963).
23. Randell, B. and Russell, L. J., ALGOL 60 Implementation, Academic Press, New York (1964).
24. Reynolds, J.C., "GEDANKEN, A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept," CACM, 13:5, pp. 308-319 (May 1970).
25. Rosenkrantz, D.J. and Stearns, R. E., "Properties of Deterministic Top Down Grammars," p. 180, ACM Symposium on Theory of Computing (May 1969).
26. Satterthwaite, E., Mutant 0.5, An Experimental Programming Language TR CS 120, Stanford Univ. (17 Feb. 1969).
27. Spier, M.J. and Organick, E.I., "The Multics Interprocess Communication Facility," 2nd Symp. on Op. Syst. Principles, p. 83 (1969).
28. Stearns, R.E. and Lewis, P.M. II, "Property Grammars and Table Machines," Info. and Control, 14, p. 524 (1969).
30. Van Wijngaarden, A., Mailloux, B.J., Peck, J.E.L., and Koster, C.H.A., "Report on the Algorithmic Language Algol 68," Num. Math., 14, pp. 79-218 (1969).
31. Weizenbaum, J., "Symmetric List Processor," CACM, 6:9, p. 524 (Sept. 1963).
32. _____, "Recovery of Reentrant List Structures in SLIP," CACM, 12:7, p. 307 (July 1969).
33. Wegner, P., Programming Languages, Information Structures, and Machine Organization, McGraw-Hill, New York (1968).
34. _____, "Three Computer Cultures; Computer Technology, Computer Mathematics and Computer Science," Advances in Computers, Vol. 10 (1970).
35. _____, Theories of Semantics, TR69-10, Center for Comp. and Info. Sciences, Brown Univ. (Sept. 1969).
36. _____, The Variability of Computations, TR-70-22, Center for Comp. and Info. Sciences, Brown Univ. (July 1970).
37. _____, The Vienna Definition Language, TR-70-21-2, Center for Comp. and Info. Sciences, Brown Univ. (May 1970).
38. Wirth, N. and Weber, H., "Euler: A Generalization of Algol and Its Formal Definition, Parts 1 and 2," CACM, 9:1 and 2 (Jan.-Feb. 1966).
39. Henhagl, W. and Jones, C. B., The Block Concept and Some Possible Implementations, with Proofs of Equivalence, IBM Lab., Vienna, TR 25.104 (April 3, 1970).