

# VIRTUAL PROGRAMMING

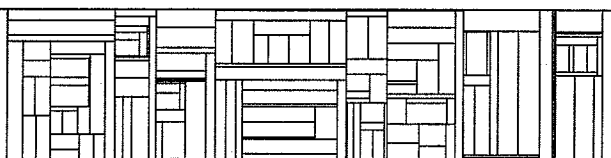
by

Brian H. Mayoh

DAIMI PB-6

August 1974 (2. printing)

Institute of Mathematics University of Aarhus  
DEPARTMENT OF COMPUTER SCIENCE  
Ny Munkegade - 8000 Aarhus C - Denmark  
Phone 06-1283 55



## Virtual Programming

B. H. Mayoh

### 1. Introduction

One of the most important developments in the art of programming is the realization that programs can be written as if there was a virtual machine between the user and the computer. With the advent of problem-oriented languages, the user could imagine that he had available say a FORTRAN machine and write his program accordingly. At first such a virtual machine was no more than a convenience for the user, freeing him from routine book-keeping and giving him a suitable conceptual framework in which to formulate an algorithm. But now we have multiprogramming operating systems that deny the user direct access to the resources of the computer and force him to use a virtual machine. On the other hand many modern computers are, in essence, virtual machines because they are microprogrammable.

Clearly virtual machines must be designed with care, since they provide the bridge between the conceptual framework of the problem-solver and the hardware of an actual computer. In this paper we present a design, that was prompted by recent work on program development (1-28), operating systems (30-37), and microprogramming (40-48). The first half of the paper consists of an analysis of some conceptual mechanisms that are useful when devising an algorithm. Some of these mechanisms are suggested by a flow diagram model of programs (sections 2.1, 2.2, 2.3) some are suggested by an activity model of computations (sections 2.4, 2.5, 2.6), while others are suggested by a model of computation states as collections of data structures (sections 2.7, 2.8, 2.9). The second half of the paper gives an outline of the design of a virtual machine ODIN on which the mechanisms can be implemented simply and efficiently. The emphasis is on the features that distinguish ODIN from more traditional machines: its ring of activity records (section 3.2), and its implementation of data structures and types (section 3.3, 3.5, 3.6). Because of these features ODIN can implement delicate combinations of iteration, parallelism, recursion, subroutines, coroutines and the like, should this be desirable.

### Preface to second printing

In the two years since publication of this report much work has been done in the area of programming languages. Rather than make the appropriate revisions in this report we prefer to wait until the language has been implemented and used. We envisage two implementations:

- using the Aarhus compiler-compiler modified so that it can accept Scott-Strachey definition of the semantics of a programming language
- direct implementation of the ODIN virtual machine on the Aarhus RIKKE-MATHILDE hardware.

## 2.0 Stepwise program development

What is the most natural way of composing an algorithm to solve a problem? A number of recent articles advocate: dividing the original problem into subproblems, supposing that we have algorithms for solving the subproblems, describing how these algorithms can be combined into an algorithm for solving the original problem, and repeating this process for the subproblems. For this approach to be feasible we must have powerful ways of describing an algorithm as a combination of other algorithms. We need a powerful algorithmic language. In the following sections we gradually invent such a language SAGA, not intended as a realistic alternative to other algorithmic languages, but rather as a specification of the kind of mechanism a virtual machine should be able to handle efficiently. As SAGA is to be the starting point for the design of a virtual machine, we can banish its exact definition to an appendix.

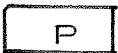

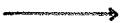







### 2.1 Flowdiagram model

Presumably all programmers know the use of flow-diagrams for describing an algorithm as a combination of more primitive algorithms. Even although ODIN can handle any flow diagram, it is unwise to permit their uncontrolled use and SAGA will have no equivalent of Algol's goto and its attendant labels ( 52 ). However, we shall often use flow-diagrams to illustrate the mechanisms SAGA does have, so we ought to define them precisely. A flow-diagram is a particular kind of directed graph in which all nodes and all edges have titles. To be a flow diagram such a directed graph must satisfy:

- a) each node has at most two leaving edges;
- b) if a node has two leaving edges, one of them bears the title true, while the other bears the title false;
- c) if a node has only one leaving edge, the edge bears the title neutral;
- d) there is precisely one node with no leaving edges, it is the only node bearing the title end;
- e) there is precisely one node with no entering edge, it has one leaving edge and bears the title begin;
- f) any node with title blind has two leaving edges.

Nodes with one leaving edge are called squares, nodes with two leaving edges are called diamonds.

In order to draw a flow-diagram we use the conventions:

	represents a node with title P and just one leaving edge;
	represents a node with title B and two leaving edges;
	represents either an edge with title <u>true</u> or an edge with title <u>neutral</u> ;
	represents an edge with title <u>false</u> ;
	also represents the node with title <u>begin</u> ;
	also represents the node with title <u>end</u> ;
	also represents the node with title <u>success</u> ;
	also represents a node with title <u>failure</u> ;
	also represents a node with title <u>resume</u> ;
	also represents a node with title <u>blind</u> ;

Nodes which have more than one representation are called special nodes, and all of them occur in the flow-diagrams in figure 1. These flow-diagrams specify four algorithms that can be built from algorithms P, Q, A, B, C, D. The last of them is the so-called dummy algorithm.

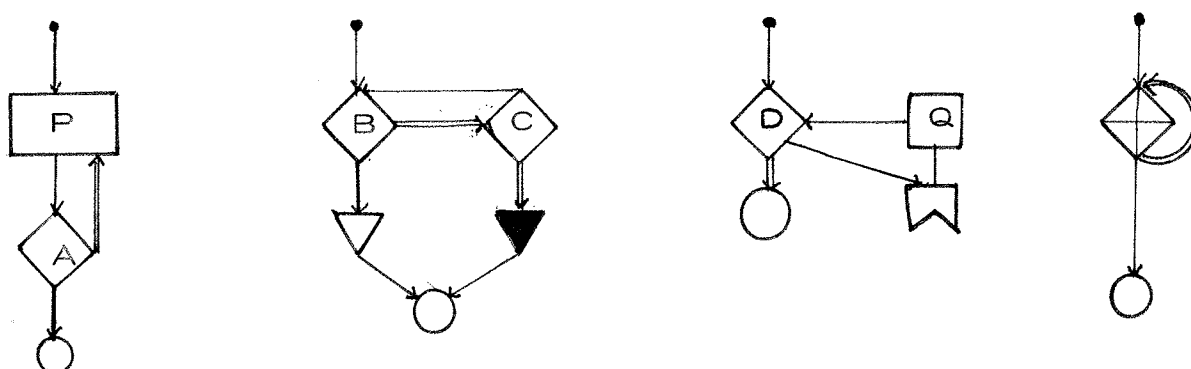


Figure 1: Four flow-diagrams

Our language SAGA, like other higher language, permits nesting of algorithms. To explain this we introduce the idea of replacing a node N by a flow-diagram S. Suppose that N has one leaving edge with destination T. Let L be the destination of the edge leaving the begin node of S. Then the result of replacing S by N is given by:

- 1) Taking L as the destination of edges that originally led to N ;
- 2) Taking T as the destination of edges that originally led to success or end nodes in S ;
- 3) Redrawing S so that each failure node loops (is the destination of its own leaving edge);
- 4) Removing the begin end success nodes of S.

Now suppose that M has a false edge with destination F and a true edge with destination T. Then the result of replacing N by S is given by 1), 2), 4) and:

- 3') If there were failure nodes in S, remove them and take F as the destination of edges leading to such failure nodes, otherwise introduce a dummy node with a false edge leading to F and a looping true edge.

Consider figure 1. The result of replacing nodes P and A by the second flow-diagram are shown in figure 2.

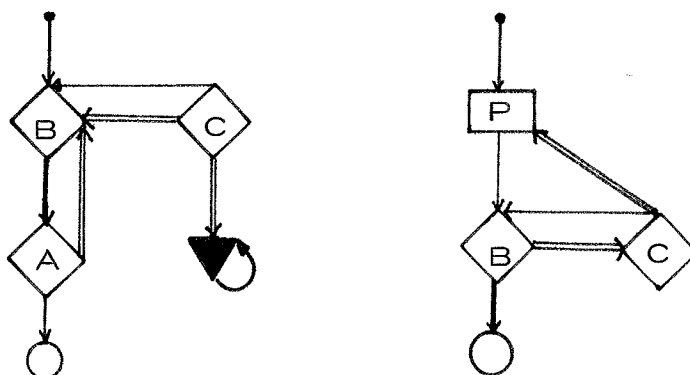


Figure 2: The result of replacing a node by a flow-diagram.

## 2.2 Operator diagrams

The simplest flow diagrams are those in which there are no diamonds and just one path from the begin-node to the end-node. If we write the titles of the nodes along this path separated by semicolons, we get the simplest kind of SAGA code, a sequence like:

begin ; P ; Q ; R ; end

In figure 3 we give other simple flow diagrams with their corresponding SAGA codes.

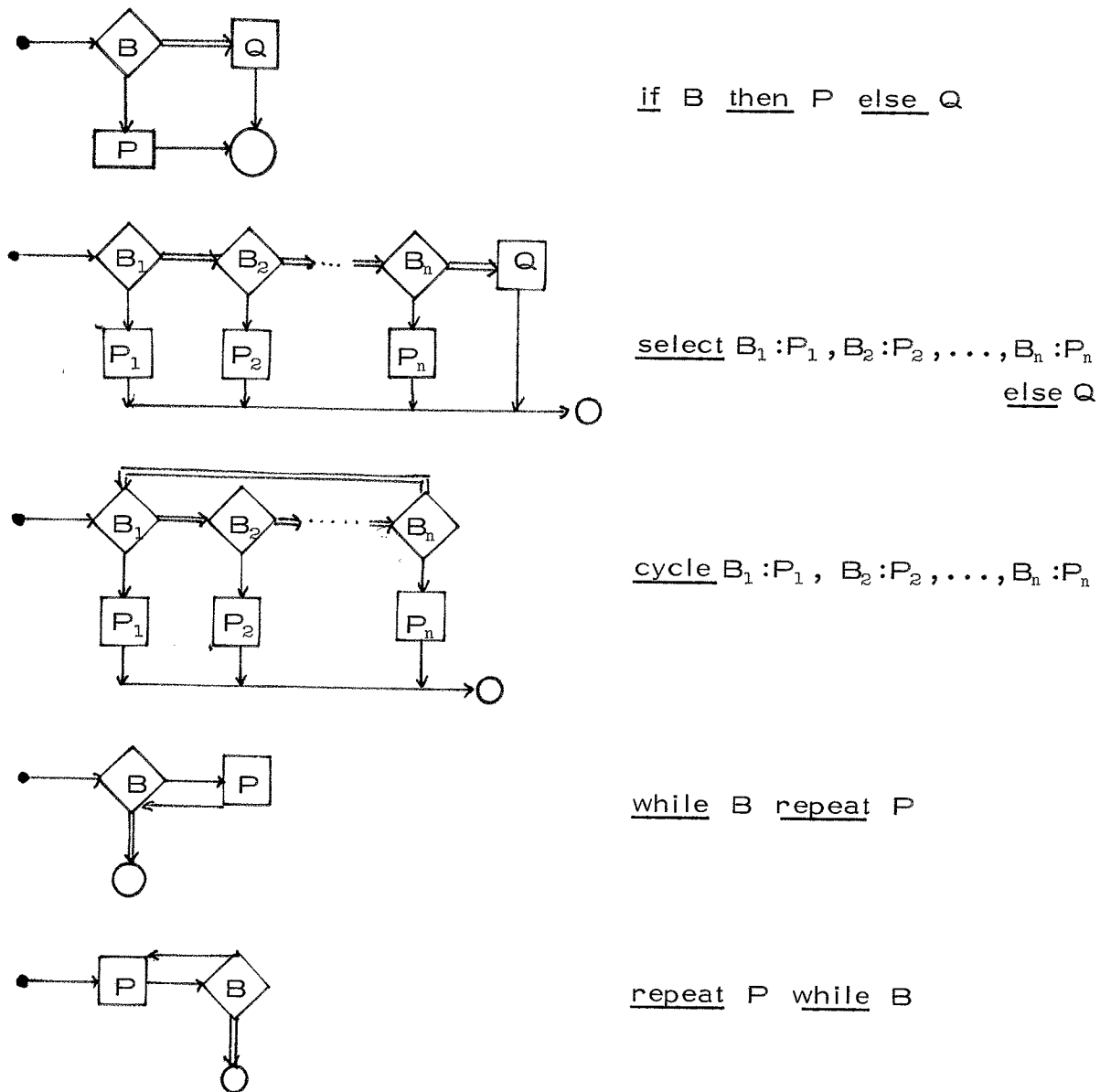


Figure 3: Operator diagrams.

Now we can explain the nesting of SAGA codes by an example. The SAGA code

if B then P else if B2 then P2 else Q2

corresponds to the first of our operator diagrams with – if B2 then P2 else Q2 – in the role of Q. If we replace the Q-node by the flow diagram for – if B2 then P2 else Q2 –, we get another flow diagram for our SAGA

code. As this diagram also corresponds to the code – select B: P, B2: P2 else Q2 – we ought to give an example of the use of nesting to produce something new. We can take the third of the flow diagrams in figure 1; it corresponds to the code: while D repeat begin; resume; Q; end.

For reasons that will become apparent later, we have introduced a name, operator diagram, for those flow diagrams that do not contain a failure node. However we replace a square in such a diagram, we get an operator diagram. If we replace a diamond by an operator diagram, we also get an operator diagram. On the other hand figure 2 shows that the result of replacing a square is not necessarily an operator diagram.

### 2.3 Condition diagrams

In the same way that the SAGA codes of the last section are related to squares in a flow diagram, the SAGA codes in figure 4 are related to diamonds.

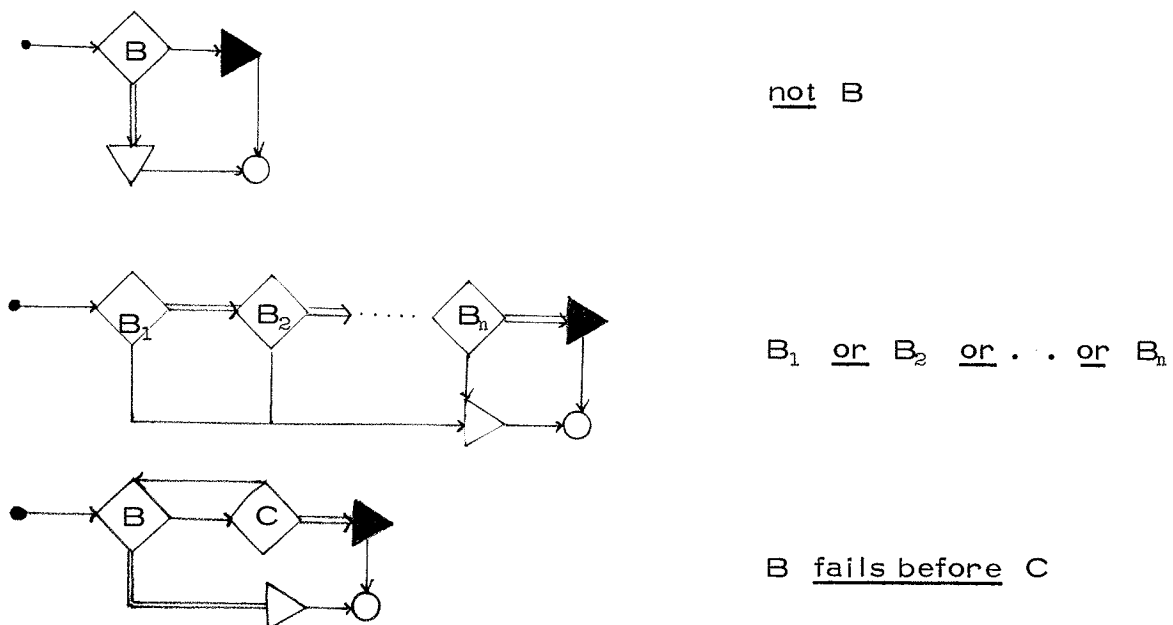


Figure 4: Condition diagrams.

The first two of these codes enable us to use nesting to build all Boolean combinations of diamonds. The third code is convenient but not necessary in that it is equivalent to: while B repeat C. Suppose we put the above flow diagrams in the place of the B node in the operator diagram for if B then P else Q. The result will be the original flow diagrams with P instead of the success node and Q instead of the failure node. It is not surprising that flow diagrams with failure nodes are called condition diagrams.



## 2.4 Activity model

Let us make our static flow diagrams dynamic. If we define an activity as a flow diagram with one or more buttons on its nodes, then we can take a finite set of activities as a suitable model for an algorithm under execution. The colour of the buttons on a node is significant:

<u>yellow</u>	this node is the activation point, it is currently being executed;
<u>green</u>	this node is the true reactivation point-if we return to this flow diagram from a successful activity it is changed into a yellow button;
<u>red</u>	this node is the false reactivation point - if we return to this flow diagram from an unsuccessful activity it is changed into a yellow button.

Activities are created by placing a yellow button on a new copy of a flow diagram. When a button reaches a success, failure, end, or resume node, we return from an activity to its creator.

Let us consider a typical computation using the flow diagrams in figure 1. Suppose we begin by placing a yellow button on the begin node of the first diagram (step 1). The button immediately moves to the P-node (step 2). Suppose P is an activity that is always successful, so our button moves to the A-node (step 3). Suppose A calls for the creation of the second flow diagram. We move our button to the begin-node of that diagram, and we place green and red buttons on the end and P nodes of the first flow diagram (step 4). The yellow button now moves to the B-node of the second diagram (step 5). Suppose B is a successful activity so this button moves to the success node (step 6). Recognizing that the kind of node it is resting on, the yellow button moves over to the green button and the green and red buttons are removed (step 7). As the yellow button is now on an end node and there are no other buttons, the computation stops (step 8).

In our example the second flow diagram ceased to be an activity when it lost its yellow button in step 7. This is known as subroutine discipline, but other disciplines are both possible and desirable. Therefore we introduce the rule:

a yellow button on a resume node is treated as if it were on an end node, except that a red and a green button are placed on the node indicated by the leaving edge.

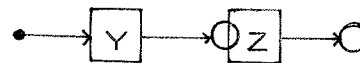
## 2.5 Clones, recursion and coroutines

In dividing an algorithm, it is often convenient to use clones - to allow distinct activities with the same underlying flow diagram. Suppose we allow end, success, failure and resume nodes to bear an extra title, and introduce the rules:

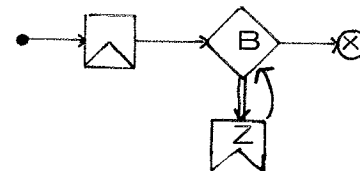
- 1) when a yellow button is moved from a node with extra title P, it goes to the most recently created activity with P as its underlying flow diagram;
- 2) unless the yellow button sat on a resume node, remove all buttons on activities created since the last P-activity.

Then powerful combinations of recursion and coroutines become possible. (5). Figure 5 gives a relatively simple example.

flow diagram X



flow diagram Y



flow diagram Z

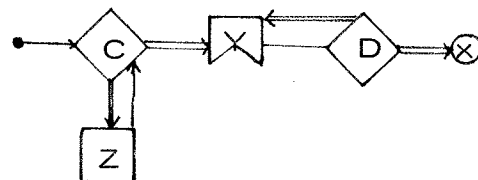


Figure 5: Simultaneous recursion and coroutines.

As an illustration of the way of expressing our new possibilities in SAGA we mention that the code corresponding to flow diagram Z is:

```
begin; while C repeat Z ;
      repeat resume Y while D ; end X
```

## 2.6 Parallelism and non-determinism

Why have we subjected ourselves to the restriction: only one activity has a yellow button? Our model can easily describe parallel activities. Suppose we agree that a yellow button on a node with title:  $B_1$  and  $B_2$  and ... and  $B_n$ , is converted to yellow buttons on new copies of each of  $B_1 \dots B_n$ . We can also agree that our original node is

successful if all the newly created activities are successful, and unsuccessful if at least one of the newly created activities is unsuccessful. We can convert this into an extension of the rules for moving yellow buttons on special nodes:

if a yellow button lands on a failure node, then all other yellow buttons are destroyed;

if a yellow button lands on an end, success, or resume node, then it is destroyed provided that it is not the last yellow button.

One may well ask whether we have achieved anything that could not be achieved by the condition diagram in figure 6 – a diagram which can be described using SAGA's or and not.

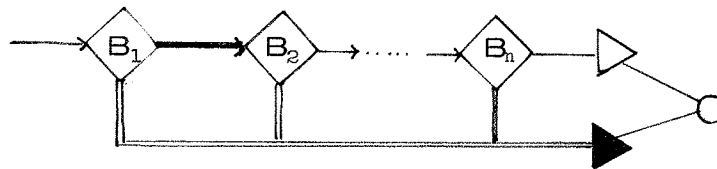


Figure 6: Pseudo parallelism.

The answer is : yes in the case when activities  $B_1 \dots B_j$  never terminate but  $B_{j+1}$  terminates and is unsuccessful. An illuminating way of expressing the difference is: in parallelism the sequence of computation steps becomes a tree, the flow diagram of figure 6 gives a depth-first search of this tree, our rules give a breadth-first search and both mechanisms are useful. Perhaps this remark will clarify some of the confusion about non-determinacy and backtracking. Consider an algorithm like: choose digits  $x$  and  $y$  such that  $x + y = 15$ . Assume the digits are taken in their usual order. If the algorithm is written as a SAGA code using or, it will find the solution  $x = 6, y = 9$ . If the algorithm is written as a SAGA code using and, it is impossible to predict which solution it will find.

Let us digress here to describe a way of handling the SAGA code –  $B_1$  and  $B_2 \dots$  and  $B_n$  – in a sequential virtual machine. The basic idea is is that we run  $B_0, B_1 \dots B_n, B_\infty$  as a ring of coroutines, where  $B_0$  and  $B_\infty$  are two codes that ensure we leave this ring correctly. Figure 7 shows the flow diagrams for  $B_0$  and  $B_\infty$ . They use the flow diagrams  $B_i$  that are defined as the result of inserting a resume  $B_{i-1}$  node in each edge of  $B_i$  and then replacing failure nodes by nodes with the title:

exit to dummy node in  $B_\infty$ .

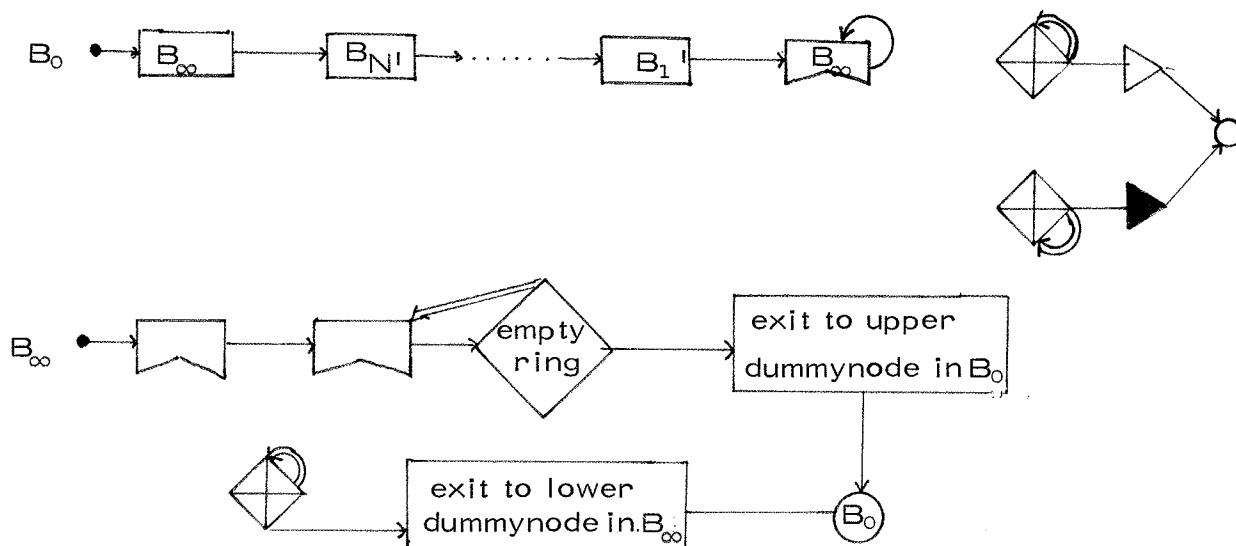


Figure 7: Flow diagrams for parallelism.

## 2.7 Data structures for the programmer.

So far the titles of non-special nodes in our flow diagrams have only referred to other flow diagrams. In our final model we allow them to refer to a set of data structures associated with the flow diagrams. A data structure has a name and a value. The value of a data structure can be either an atom, a construct, or the name of another data structure. In figure 8 some data structures are represented using the convention: small letters for names, capitals for atoms.

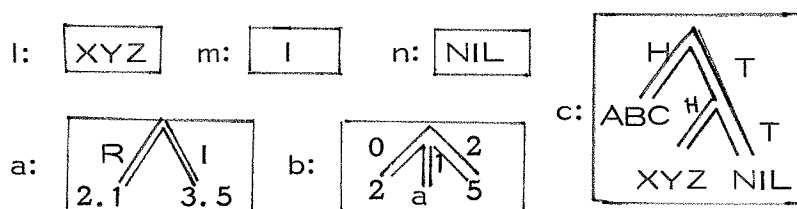


Figure 8: Some data structures.

A construct is defined as a finite list of pairs satisfying the requirements:

- a) the first component of each pair is an atom;
- b) the second component of each pair is either an atom, a construct, or the name of a data structure. Thus the value of `c` in figure 8 is the list:

( (H, ABC), (T, ((H, XYZ), (T, NIL))))

and the second component of its second pair is also a construct. Now the reason for introducing constructs is to manipulate implicit data structures. In figure 8 we can look upon a `[R]` as the name of a data structure whose value is the atom 2.1. Other examples from our figure are:

- `b[2]` is the name of a data structure with value 5 ;
- `b[1]` is the name of a data structure with value a ;
- `c[T, H]` is the name of a data structure with value XYZ ;
- `c[T]` is the name of a data structure with value ((H, XYZ), (T, NIL)).

Part of the power of SAGA lies in its ability to handle the general data structures implicit in the construct concept.

## 2.8 Expressions and places.

The distinction between the name of a data structure and its value ( 48 ) is reflected in SAGA's distinction between a place and an expression. The SAGA code - `place:=expression` - puts the value of the expression as the value of the data structure named by place. A simple example is the use of `d := a [1]` to put 3.5 as the value of `d`. A more complicated example using a reserved code name pure is given in figure 9. Another important SAGA code - `expression = expression` -

`b := pure (b)` in figure 8 gives `b :`

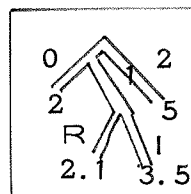


Figure 9: The removal of names from a construct.

tests whether two expressions have the same value. Perhaps we ought also to mention the other SAGA tests - atom (place), name (place), un-

defined (place) – the last of which tests whether a data structure has value NIL.

Our next task is to describe how data structures are created, for SAGA assumes that no local data structures exist when an activity is created and destroys them all when the activity is destroyed. The simplest way of creating a data structure is to use the code: define dataname = atom.

Examples of this and other SAGA codes for creating data are shown in figure 10.

define x = 1; complex a ; pointer n ; from 1 upto 3 A; prime b; gives

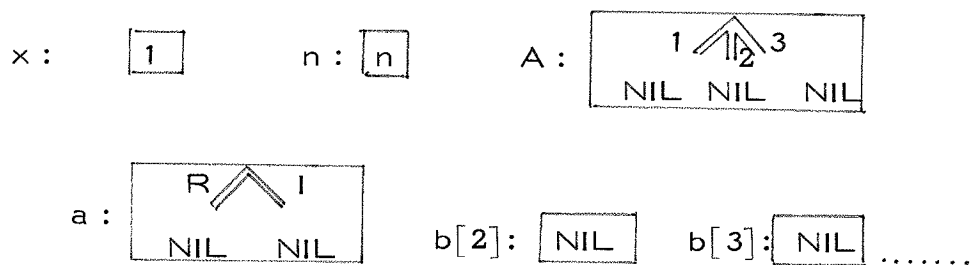


Figure 10: Data creation.

Let us close this section by describing the creation of data structures for formal parameters. SAGA allows codes to have parameters so that the programmer need not write many trivial variants of a code. At creation of a code with parameters the resulting activity is presented with a finite sequence of either names, atoms or constructs, the actual parameters. The SAGA code – value xyz – creates a data structure with name xyz and the value given by the corresponding actual parameter. The SAGA code – reference rst – creates a data structure with name rst and then checks whether the corresponding actual parameter is a name. If it is, rst becomes an alternative name of the same data structure, an alias; if it is not there is an error and computation is terminated. Suppose Add To List is the name of the SAGA code:

begin; value u; reference m;  $m[T] := M$ ;  $m[H] := u$ ; end.

Figure 11 shows various situations that arise if this code is activated by: Add to List (UVW, c).

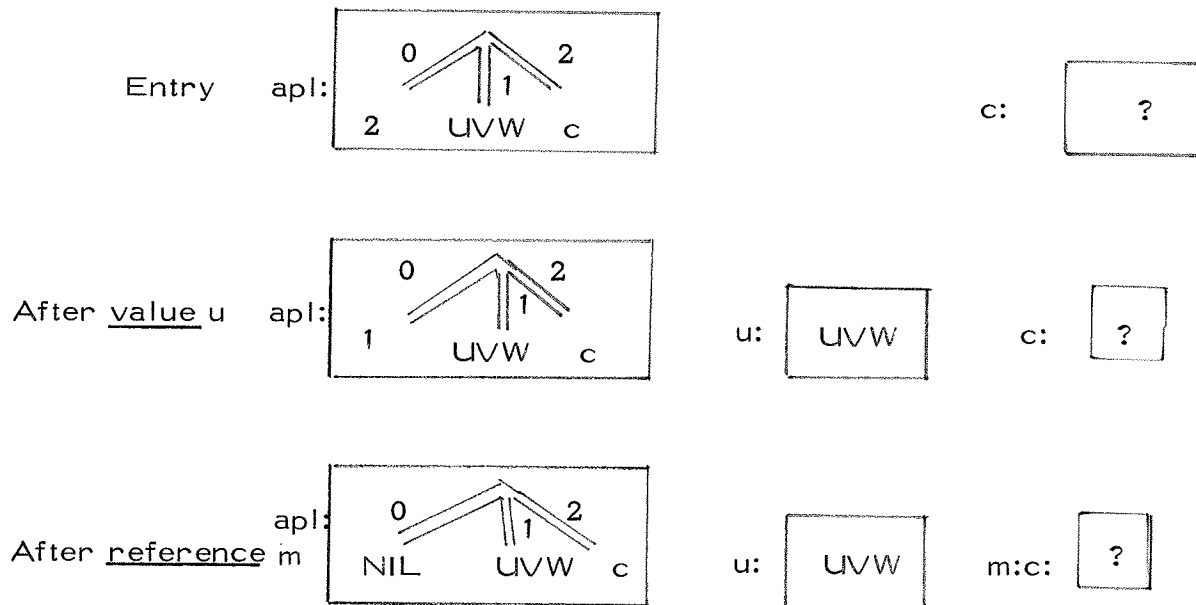


Figure 11: Parameter passing.

One should not be misled by this example into thinking that SAGA forbids general expressions. Given a call like – Add to list (pure (b), c[H]) – it converts pure (b) to a value and c[H] to a name before creating apl and entering Add to list. Furthermore the entry values of the actual parameters are available throughout the code in the data structure with name apl.

## 2.9 Types

The use of types in SAGA is somewhat unusual; in spite of the fact that a data structure is created with an associated type, this need not limit the possible values of the data structure in any way (see section 3.5 for the uses when there is a limitation). Before explaining why types are associated with data structures, we must describe how they are created. A type is a finite or infinite ordered list of atoms ( 27 ) different from NIL. As an illustration of the ways of specifying types, consider:

```
[REAL, IMAGINARY]
from 3 upto 7 from 4 down to 1 from 0 step 3 until 21
[A increment fetch next letter test no more letters]
[A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]
[2 increment next prime].
```

The last of these specifies an infinite type, while the two previous specifications give the same type. As a convenience to the programmer SAGA has a code – associate typename with type specification – that allows him to give names to types. Thus three of the above type specifications may well be given the names: complex, alphabet, prime.

We have already seen one important use of data types in figure 10: they enable us to have constructs as initial values of data structures. Admittedly this only gives simple constructs as initial values, but the following SAGA code shows that arbitrarily complicated constructs can be generated:

```
begin; associate list with [HEAD, TAIL]; list c; define I = LMN;
      c[HEAD] := XYZ ; c[TAIL] := c; c[HEAD] := ABC ;
comment now c is as in figure 8;
      if case I in list then P else Q;
comment equivalent to if I = HEAD or I = TAIL then P else Q;

      if some j in list satisfies undefined (c[j]) then S else T;
comment this is equivalent to if undefined (c[HEAD]) or
      undefined (c[TAIL]) then S else T;
      if all j in list satisfy atom (c[j]) then U else V;
comment this is equivalent to if atom (c[HEAD]) and
      atom (c[TAIL]) then U else V;
end.
```



This example also shows the use of types to express switching and the two kinds of parallelism (section 2.6).

Perhaps the most important use of types is in iteration. Suppose j is the name of a data structure that is either undefined or has a value in type T. Then the SAGA code – next j in T – is equivalent to:

```

select j = NIL : j := first element of T,
           j = last element of T: begin; j := NIL; failure; end,
           else j = element of T after (j)

```

except that SAGA automatically ensures that there is a data structure with name j and initial value NIL. As an example of the use of the new code for describing iterations we can take:

```

while next j in alphabet repeat B.

```

In fact such iterations occur so frequently that SAGA allows – while next j in – to be replaced by – for j – giving SAGA codes like:

```

for j from 2 upto 17 repeat B.

```

In closing this part of the paper let us emphasize that Algol's goto statements with their attendant labels have no equivalent in SAGA (50, 52, 54, 59). There are several arguments for taking this seemingly drastic step, despite the two counter arguments : the need for error exits from complicated codes, and the need for an emergency exit from an iteration. The SAGA possibility of writing codes like end P takes the sting from the first of these counter arguments, while the following example shows that the second is not so very compelling: the algol code

```

for j := 1 step 1 until 6 do
           if B then begin P; goto L; end ;
Q; L :

```

corresponds to either of the SAGA codes

```

if next j in from 1 upto 6 fails before B then Q else P
if some j in from 1 upto 6 satisfies B then P else Q.

```

### 3.0 The design of a virtual machine

In the rest of this paper we sketch the design of a virtual machine in which SAGA can be implemented simply and efficiently. We shall use the same gradual approach we have just used to present various aspects of SAGA. We began with a flow diagram model of an algorithm, moved on to an activity model of a computation, and finished by describing some simple codes using data structures. Here, the flow diagram model will suggest a traditional (von Neumann) machine, the activity model will modify the design somewhat, and the simple codes will suggest primitive operations and data referencing mechanisms. The emphasis is on showing that the machine can handle the most general SAGA situations, and the fact that there are more efficient ways of treating particular situations is ignored.

#### 3.1 Instructions and labels

Consider the flow diagram model again. We would like each node to correspond to a single instruction in our machine. The most direct way of doing this is to have instructions with two fields, one of which tells how many edges the node has, while the other gives the title on the node. But then we have a problem – how do we specify the destination of the edges that leave the node? There seems to be no good reason to abandon the usual sequential solution – if there is only one leaving edge, it leads to the node corresponding to the immediately following instruction. This motivates a (control address) register NEXT, which points to the instruction about to be obeyed and is incremented during the execution of unconditional instructions. We can also use NEXT for the false edge from a diamond, but the true edge must be treated differently. Let us introduce a (memory address) register LOCATION to point to the instruction corresponding to the node for the true edge. Noting that we need a new kind of instruction to load LOCATION, we can give our first specification for the machine ODIN:

- a) it can store a program, a sequence of instructions each of which has a mode field and a name field;
- b) an instruction in do mode with name P corresponds to a square node with title P;
- c) an instruction in if mode with name B corresponds to a diamond node with title B;

- d) an instruction in fetch mode with name D causes LOCATION to point to the instruction labelled D.

A glance at figure 12 should make this clear. The figure also illustrates the fact that the existence of the title dummy ensures that every flow diagram can be implemented. Furthermore it shows that this part of ODIN is very similar to a traditional von Neumann machine – if mode corresponds to transfers, do mode corresponds to the other operations, while fetch mode corresponds to the standard addressing mechanism. This similarity becomes even more striking in the second program of figure 12, where we use a new (accumulator) register RESULT, and we use fetch mode to access simple data structures.

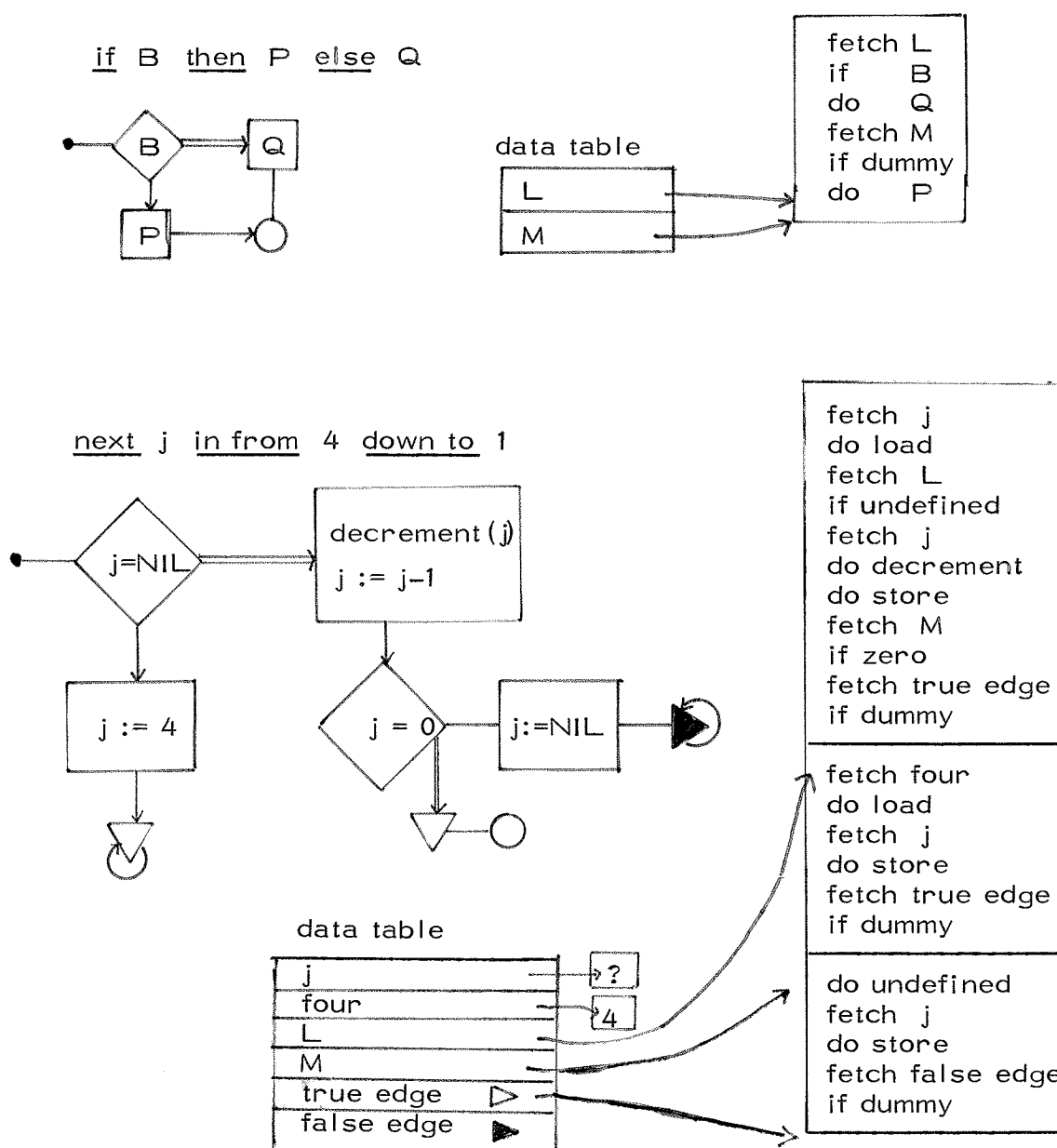


Figure 12: Possible implementations of two simple SAGA codes.

### 3.2 The activity record ring

Now for the activity model of section 2.4. An activity is represented in ODIN by an activity record. One of the fields in the activity record points to the name of the program for the underlying flow diagram, another points to the data table for this activity, while the other four fields are used when we leave this activity without destroying it. How should ODIN treat a do or if instruction if it does not recognize the contents of its name field as a primitive? A possible answer is:

- store **LOCATION**, **RESULT**, **NEXT** in the L, R, N fields of the current activity record;
- search a list of the names of known programs for a match on the name field, and terminate the computation if there is no match;
- create a record for the new activity;
- make **NEXT** point to the first instruction of the new program.

In figure 13 we show the ODIN situation after step 3 and 4 of the computation in section 2.4.

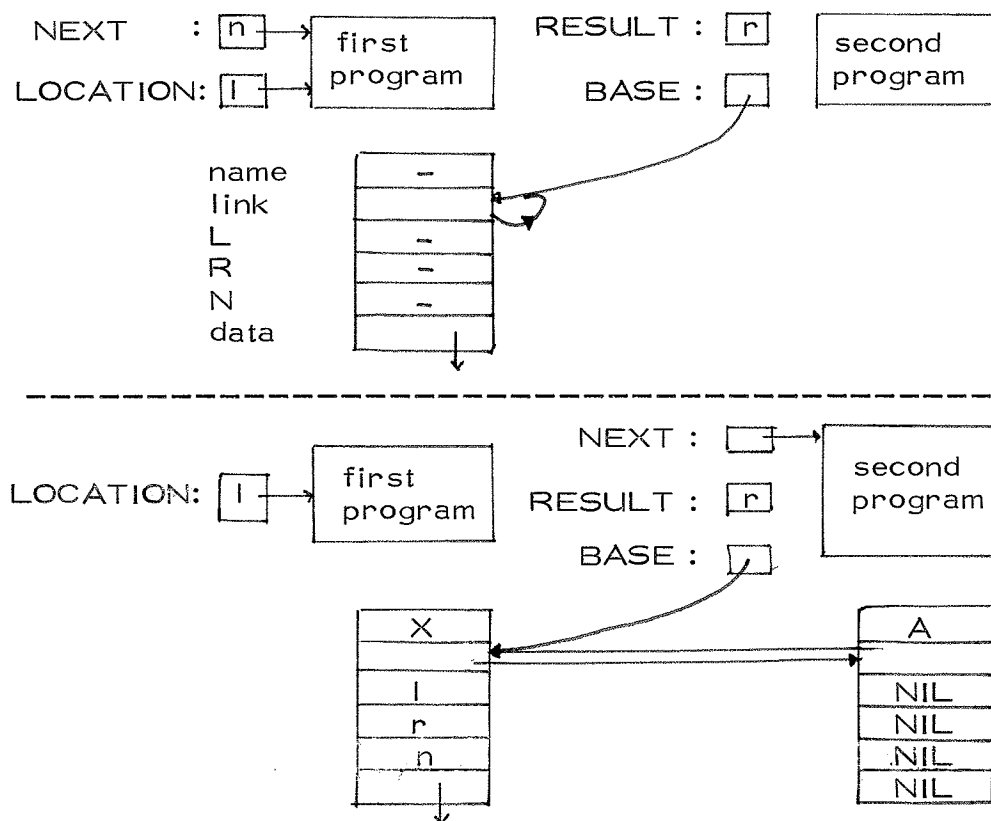


Figure 13: The creation of a new activity record.

With the remark that a yellow button corresponds to the pointer in NEXT, a green button corresponds to a possible pointer in the creator's N, and a red button corresponds to a possible pointer in the creator's L, we invite the reader to work out an appropriate way of handling success, failure and end nodes in a flow diagram (consult appendix 2). Here we discuss resume nodes in order to emphasize the fact that activity records are arranged in a ring. ODIN has a hardware register BASE which points to the bottom activity record. The link in that record points to the current activity record, and from there we return to the bottom activity record via links in the other activity records. Executing a resume node places the bottom activity record's link in BASE. In figure 14 we show a sequence of activity record rings that might arise in a computation using the programs in figure 14.

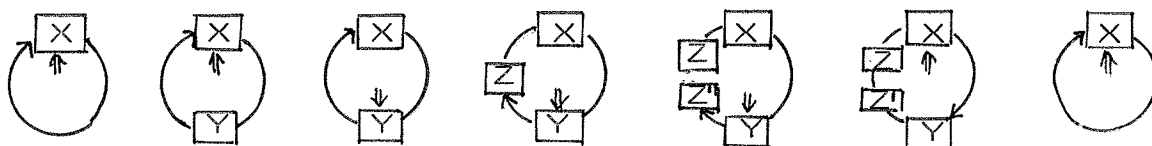


Figure 14: A sequence of activity record rings.

### 3.3 Implementation of data structures

If SAGA didn't have constructs, there would be no problem in implementing data structures. One could just follow the traditional von Neumann approach: treat data structures as labels and keep their values in the data table. However, constructs are useful and our virtual machine should be able to handle them neatly. Let us first consider pure constructs, constructs that do not use the name of a data structure. The easiest way of describing the ODIN representation of pure constructs is to refer to figure 15.

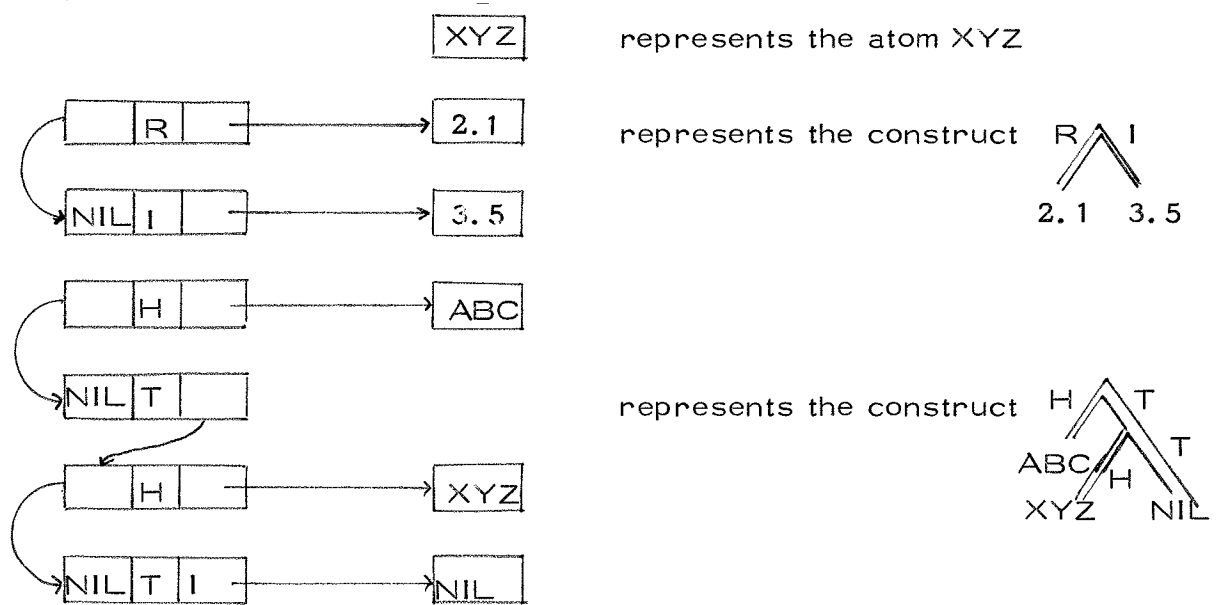


Figure 15: Pure constructs in ODIN

The most important point to note is that each construct has precisely one handle, a cell to which no arrow points. We can use this fact to implement data structures that have pure constructs as values. Such data structures can be represented by a data head, an entry in the data table containing the name of the data structure and a pointer to the handle of the value. Now consult figure 16. It shows that not only can we represent data structures with pure constructs as values, we can represent all data structures.

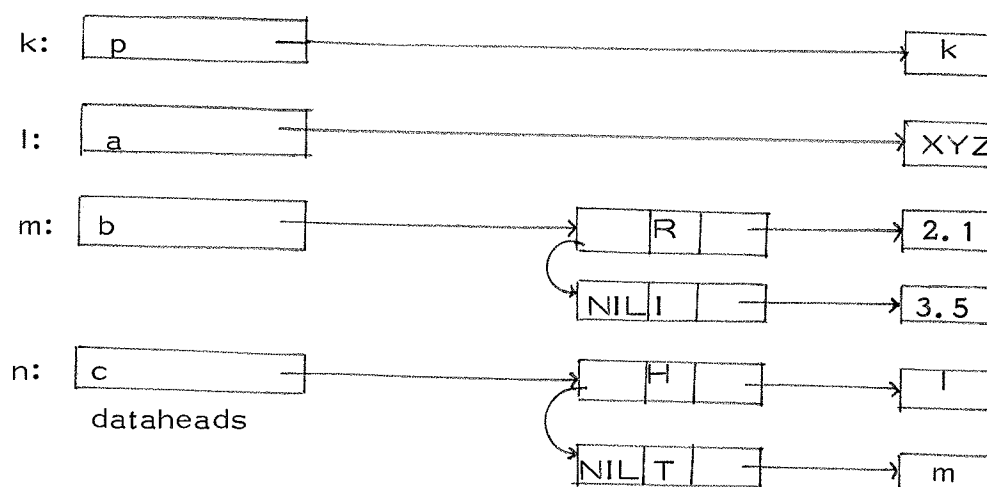


Figure 16: Data structures in ODIN

### 3.4 Primitives

Our next task is to find a way of implementing SAGA's place and expression in ODIN. This is easily done. For place we devise a program that puts a pointer to a handle in `LOCATION`, for expression we devise a program that puts a handle in `RESULT`. We also have no difficulty with the code - place := expression - we can put the instruction, do store, after the programs for place and expression. However, the implementation of - expression<sub>1</sub> = expression<sub>2</sub> - is not so clear, because we seem to have no room for the false exit in `LOCATION`. What if we let ODIN have a primitive operation, Compare, that causes `NIL` to be in `RESULT` if and only if: `RESULT` and `LOCATION` contain pointers to handles of identical values? Then our SAGA code can be implemented using the instructions do compare and if undefined.

There is another important situation in which it is convenient to let `RESULT` contain a pointer to a handle and not the handle itself. If `RESULT` contains a pointer to the actual parameter list at entry to a parameterised program, then the `R` field of the creating activity record can play the role of `apl` in section 2.8. Now ODIN can implement reference  $x$  (value  $x$ ) by creating a data head with  $x$  in its name field and a pointer to the value of (to a new copy of the value of) the corresponding actual parameter. Figure 17 is intended to make this clearer; it shows the representation of the data structures in the bottom line of figure 11.

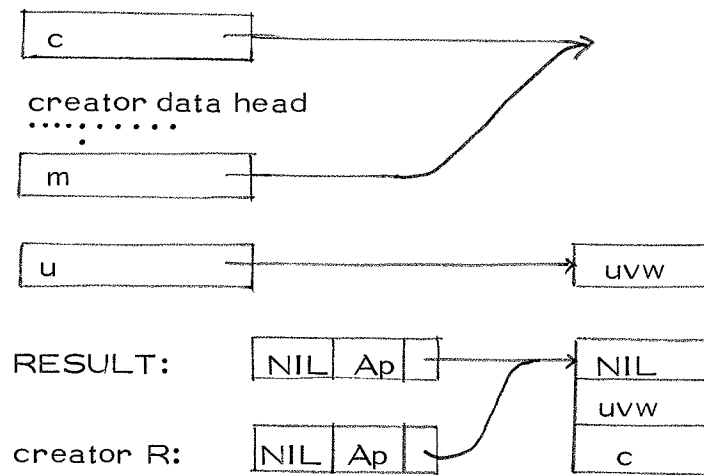


Figure 17

### 3.5 Non-standard representation

In the next two sections we describe how ODIN can use types to create data structures. To give an impression of what is needed, figure 18 shows the ODIN representation of the data structures in figure 10. We see that

- a new field has appeared in our data head, the code field;
- the data structures, which have a defined code field in their data head, also have a non-standard ODIN representation.

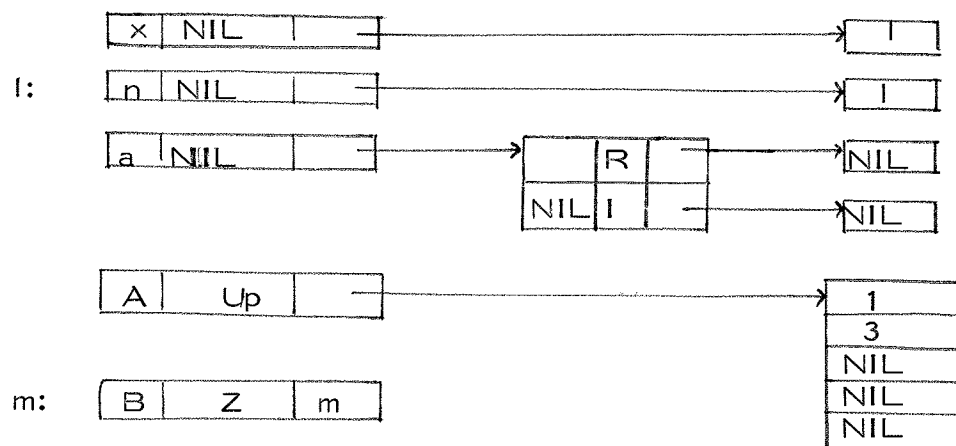


Figure 18: Newborn ODIN data structures.



The code field's and their attendant non-standard data representations are introduced, so that arrays, parameter-parsing and iteration can be handled efficiently by ODIN. But we pay a price: the range of possible values of data structures with non-standard representations is severely limited. However, the programmer is free to avoid this range limitation by using say `[1, 2, 3]` instead of `from 1 upto 3`.

Let us begin the discussion of code fields by considering the primitive `Up` in the data head for A. Whenever ODIN meets the instruction `fetch A`, it first puts a pointer to the value of A in `LOCATION`, and then obeys the instruction `do Up`. This checks that `RESULT` contains a value in `from 1 upto 3`, and exits with a pointer to the value of `A[RESULT]` in `LOCATION` if it does. The primitive `Up` is one possibility for the code field of a newly created data structure; others are:

<code>Down</code>	for type specifications like <code>from 7 down to -4</code> ;
<code>Step</code>	for type specifications like <code>from 3 step 2 to 7</code> ;
<code>Y</code>	for type specifications like <code>[A increment 1 test T]</code> .

The first two of these are primitives, but the third is not. As we shall describe later, `Y` is the name of one of the three programs `Y Y' Y''` constructed from the programs for `I` and `T`.

### 3.6 Type creation

For each type specification in a SAGA code, the corresponding program generates a data structure. The data structures for the first five type specifications in the first paragraph of section 2.9 are shown in figure 19. A natural question is: what should ODIN put in the name field of the data heads? The answer is: if the type specification is preceeded by

associate type name with

then the type name, otherwise a reserved type name.

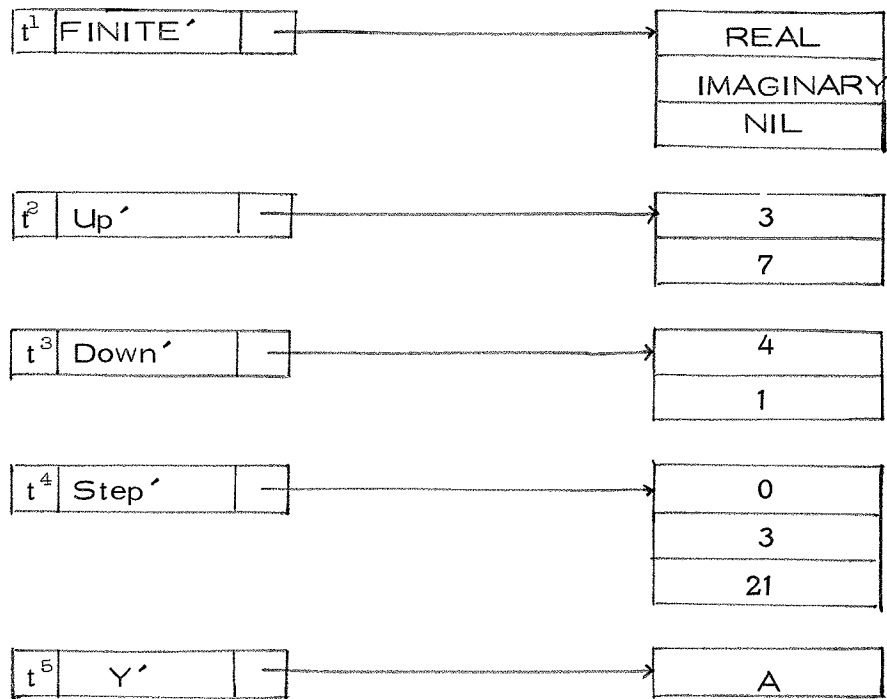


Figure 19: Type representation in ODIN

Now for the code fields of the type data heads. Each time ODIN meets the instruction – fetch type name – it executes the instruction `do N` where `N` is name in the code field. The consequences of this instruction depend on whether or not `RESULT` contains `NIL`. If this condition is met then our ODIN instruction will cause a data structure to be born. Figure 20 shows the result of fetching the data heads in figure 19 with `NIL` in `RESULT`.

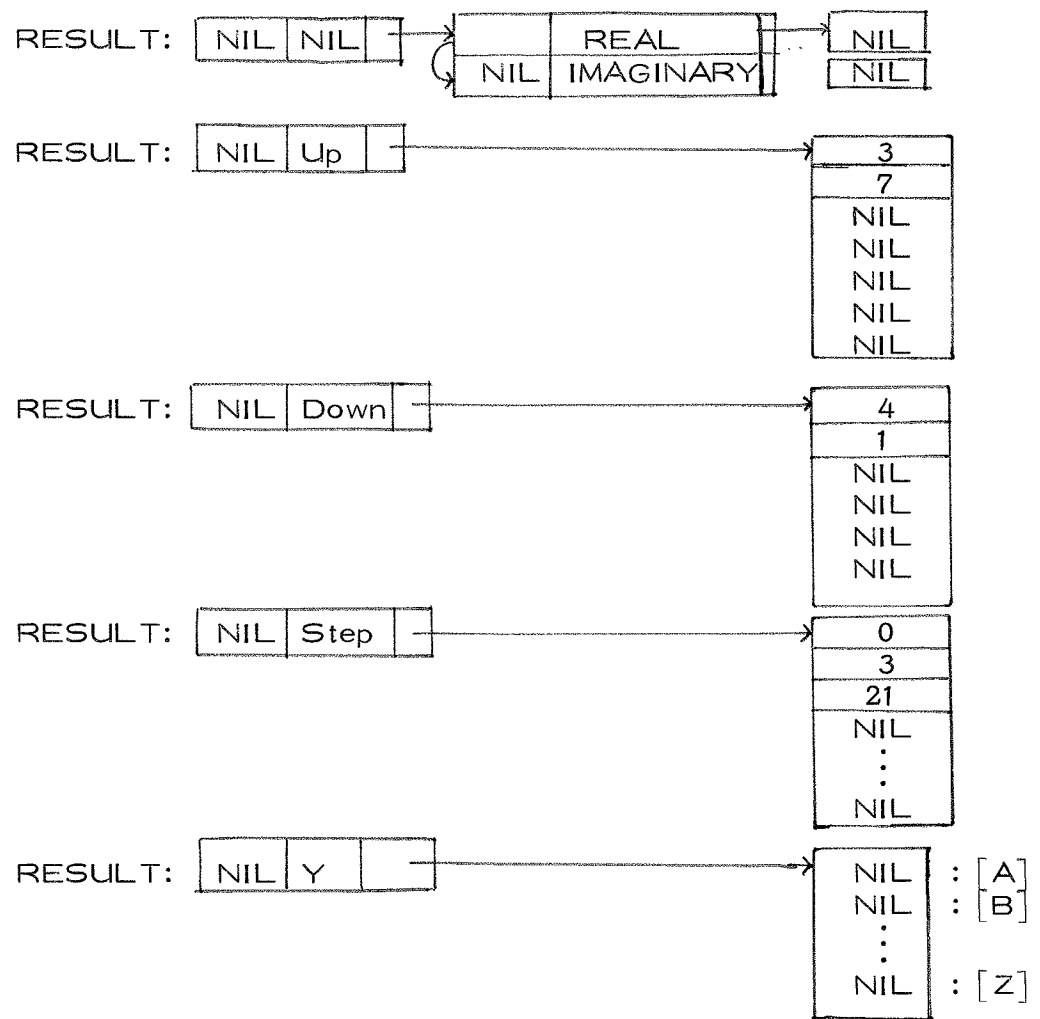


Figure 20: ODIN datastructures at the moment of birth

### 3.7 Iteration

For each occurrence of next, some and all in a SAGA code, the corresponding program generates a data structure. Since NIL is not a pointer we can also generate the data structure for j in next j in T by using the code field of the data head for T. Figure 21 shows the results of fetching the data heads in figure 19 with RESULT pointing to the name j.

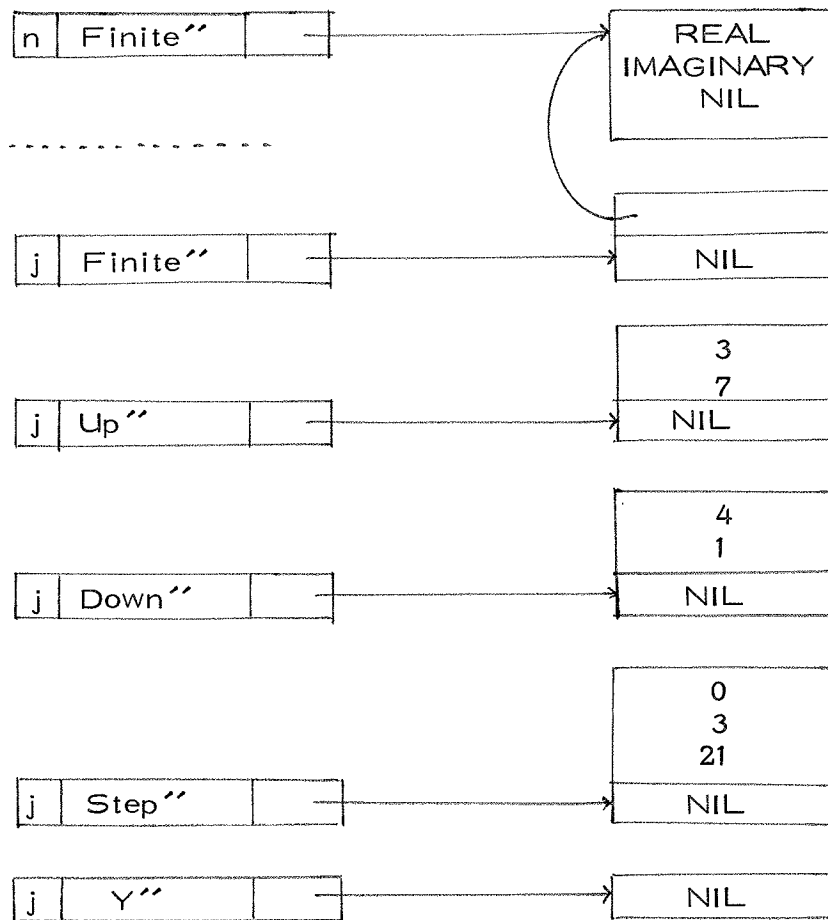


Figure 21: Index representation.

Why do we introduce these data heads? Consider the SAGA code: next j in from 4 down to 1. Once we have the data head for j, we can replace the long ODIN program in figure 12 by: fetch j; fetch false edge; if undefined. Instead of giving similar examples for each of the eleven primitives in figures 19–21, let us illustrate them by describing the ODIN programs for Y, Y' and Y'' associated with a type specification like [A increment | test T]:

program Y<sup>I</sup> This produces the data structure in the bottom line of either figure 20 or figure 21, if `RESULT = NIL` at entry it uses the programs for `I` and `T`;

program Y At entry `LOCATION` points to the first of a number of atoms, at exit it points to the atom whose selector matches `RESULT` - if there is no match, it exits to the program `Selector Error`;

program Y<sup>II</sup> At entry `LOCATION` points to the value `v` of an index `i`, at exit `i` is updated and `RESULT` contains the updated value (`NIL` if `v` was the last value).

Obviously `Y` and `YII` must also use the programs for `I` and `T`.

Because we have defined the data head for an index `j` in such a way that `j` is automatically updated when we execute fetch `j` we have a problem: `SAGA` allows one to evaluate an index without it being updated. To solve this problem we introduce a data mode. If there is a data head for `j`, the instruction data `j` is equivalent to fetch `j` except that the code field is ignored. What if there is no data head for `j`? Then we can use data `j` to create such a data head. If we agree on this, we can close a gap in our description of `ODIN` by explaining the implementation of `SAGA`'s define and pointer. Consider the top two data structures in figure 18. These can be created by the program:

data `x; NIL; 1; do head; data n; NIL; NIL; do store,`  
 where head is a primitive that makes `RESULT` point to the place in which a new data head would be created.

### 3.8 Space management

So far we have described our virtual machine as if it had an unlimited supply of many different kinds of storage cells. This section is devoted to showing that our machine need only have a traditional store with just one kind of element, a word. Because the reader probably knows how one compiles languages that allow recursion (53, 56, 58), this section need not be long.

Let us begin by looking at the degree of permanency of the various uses of storage space. We can distinguish between:

- fixed space the space needed for the process table and the programs for the different SAGA codes never changes;
- stable space the amount of space required by activities that are not current only changes when a new activity is created or an old one destroyed (or by references through formal parameters – see later);
- unstable space the amount of space required by the current activity for its record and its data structures expands and contracts frequently;
- volatile space the work space, required by ODIN when it is evaluating a SAGA place or expression, is not needed when the other kinds of space expand or contract.

In figure 22 we show the storage disposition sometime after the situation at the bottom of figure 13.

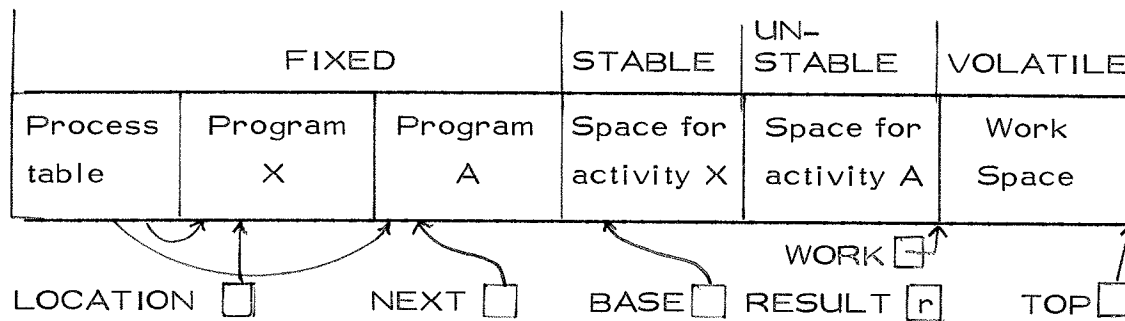


Figure 22: Storage disposition.

Now let us look at different ways an activity uses its space. At entry we create an activity record (13 words) and start executing instructions. From time to time these instructions will call for the generation of a new data head (4 words), a new data block (1 or more words) and perhaps a list of construct elements (3 words each). In our figures hitherto we have shown how construct elements are linked together to form a construct, but not how data heads are linked together to form the global data table. We get this table by starting with the top data head of the current activity and following links until we reach the bottom data head of the bottom activity. What use is such a large data table? So that one can use and change data structures that belong to another activity. This is both useful and dangerous ( 22 ). Since SAGA avoids the dangers by the draconian rule: only atoms can be assigned to non-local data structures, we can let instructions in fetch-mode search the global data table. But what of formal parameters called by reference? The example, Add to list, in section 2.8, suggests that they should be allowed more freedom. Therefore SAGA avoids the dangers by the less draconian rule: at exit from a parameterized code, we purify the constructs that are the values of reference parameters. Fortunately this rule can be implemented without much difficulty.

#### 4.0 The environment

In this final section we discuss briefly the interface between a program under execution and its environment. There would seem to be several relevant meanings of the word environment here:

- 1) the reserved code names which are available to all programs;
- 2) the facilities offered by the operating system, in which the virtual machine is embedded (if in fact we have implemented the machine in a multiprogramming system rather than on a microprogrammable computer);
- 3) the programmer watching his program execute, while waiting at his terminal.

The executing program can interact with each of these kinds of environment.

We can take input as an example of the first kind of interaction. Suppose, for simplicity, that there is only one input device and it transmits one character at a time. Let in be the name of a code that puts the atom corresponding to the next meaningful character from the device – NIL if there is none, spaces and line feeds omitted ... – in the cell to which LOCATION points. Consider a data head with in in its code field. Depending on how it is accessed (fetch or data mode) , a subsequent do load will give either the next input atom or the original value.

We can take file references as an example of the second kind of interaction. Suppose we have a character file F stored on peripheral equipment. If we devise a code get F that fetches the next character of F, we can use it in the role of in and treat F as if it were an input file. Analogously for output: a reserved code out for standard output, and a programmer code put F for output files. We note, in passing, that F functions as a push down stack if we have both put and get. More important, put F and get F need not be character oriented. Although the elements of F can be any pure construct, the most common case is when they have the same specifier structure – so F is a record file ( 12 ). With the remark that codes like rewind F are also needed for a satisfactory file administration, let us move on to the third kind of interaction.

Inspired by in and out, we may allow the programmer to provide himself with a running commentary on his executing program by letting him put the reserved names monitor i in the code fields of troublesome data heads. That was a luxury, now for a necessity: the programmer must be informed when his program meets an error. ODIN recognizes four kinds of error:

PROCESS ERROR when an activity tries to create another activity and it doesn't recognize its name;

DATA ERROR when an activity tries to access a data structure and it doesn't recognize its name;

SELECTOR ERROR when an activity tries to access an implicit data structure, and cannot match a selector;

PARAMETER ERROR when an activity has trouble with the actual-formal parameter correspondence.



But this is not enough. The programmer should be provided with powerful error-recovery mechanisms. He should be able to ask for a more or less detailed description of the current computation state: the process table, the activity record ring, the codes, the data tables, the values of data structures ... He should be able to replace the current computation state by another; in particular, to define a code for an undefined code name, or a data structure for an undefined data name. He should be able to restart the computation, after he has changed the current computation state to one more to his liking. Furthermore the whole of the above conversation should be conducted in the programmer's own conceptual framework (SAGA). This is hardly possible unless the conceptual framework of the underlying machine (ODIN) is very similar.

Let us devote our final paragraph to the virtues of incomplete programming. By this we mean the use of deliberately undefined code and data names. Why is this a good thing? There is a static answer and a dynamic answer. The static answer is that the concentration, needed to compose or understand an algorithm, should not be dissipated by worries about situations that are unlikely to arise. The dynamic answer is that many problems are most suitably solved by algorithms that require real time interaction.

## Appendix 1

### The definition of the language SAGA

It is customary for the inventors of languages to give a precise definition of the syntax in Backus-Naur form, an informal description of the semantics, and numerous examples ( 27 ). Since the first part of the paper is an informal description of the semantics of SAGA, we need only make a few supplementary remarks in this appendix. We also dispense with examples, because we have already given many and there is a long SAGA program in the next appendix. Furthermore our definition of the syntax is not complete because of typographical limitations on this paper. The major omission is the syntax of names. SAGA has an unlimited supply of data names, type names, code names, and atoms, all of which can be distinguished from one another and the underlined words in the rest of the syntax at a glance. In other words no name conflict can confuse the parsing of a SAGA program. Omitting name syntax has shortened this appendix and the use of a star for repetition of a metaconcept shortens it still further.

### Inner language

Prompted by ( 24 ) we devide the SAGA into an inner language, which gives the simplest significant SAGA codes, and an outer language which describes how simple codes can be combined into a complicated program

```

<ODIN program>::=<instruction>
<instruction> ::=do{code name}|if{code name}|while{code name}|
               fetch{data name}|<datainstruction><atom><atom>|
               <activity change>|<activity change><code name>
<data instruction>:=data{data name}|data NIL|label{data name}
<activity change>:=destroy|exit|wander|dummy|
                  success|failure|resume|blind|

```

These rules allow one to use virtual machine code in a SAGA program, should this be convenient.

$\langle \text{type specification} \rangle ::= \langle \text{scalar type} \rangle \mid \langle \text{finite type} \rangle \mid \langle \text{infinite type} \rangle \mid \langle \text{from type} \rangle$   
 $\langle \text{scalar type} \rangle ::= [ \langle \text{expression list} \rangle ]$   
 $\langle \text{finite type} \rangle ::= [ \langle \text{expression} \rangle \underline{\text{increment}} \langle \text{code name} \rangle \underline{\text{test}} \langle \text{code name} \rangle ]$   
 $\langle \text{infinite type} \rangle ::= [ \langle \text{expression} \rangle \underline{\text{increment}} \langle \text{code name} \rangle ]$   
 $\langle \text{from type} \rangle ::= \underline{\text{from}} \langle \text{expression} \rangle \underline{\text{step}} \langle \text{expression} \rangle \underline{\text{to}} \langle \text{expression} \rangle \mid \underline{\text{from}} \langle \text{expression} \rangle \underline{\text{up to}} \langle \text{expression} \rangle \mid \underline{\text{from}} \langle \text{expression} \rangle \underline{\text{down to}} \langle \text{expression} \rangle$

The use of types in SAGA was explained in section 2.9., but the examples only had atoms for  $\langle \text{expression} \rangle$ . In later syntactic rules we introduce the metaconcepts  $\langle \text{type definition} \rangle$ ,  $\langle \text{structure definition} \rangle$ ,  $\langle \text{implicit definition} \rangle$  and  $\langle \text{iteration} \rangle$ . All type specifications occur in instances of one of these. If a type specification occurs in an instance of a type definition, its expressions are evaluated at the time of the type definition. Otherwise the expressions of the type specification are evaluated at entry to the smallest named code or compound code that contains the type specification

$\langle \text{type definition} \rangle ::= \underline{\text{associate}} \langle \text{type name} \rangle \underline{\text{with}} \langle \text{type specification} \rangle$   
 $\langle \text{structure definition} \rangle ::= \text{define} \langle \text{data name} \rangle = \langle \text{expression} \rangle \mid \langle \text{structure kind} \rangle \langle \text{data name} \rangle$   
 $\langle \text{structure kind} \rangle ::= \langle \text{type} \rangle \mid \underline{\text{pointer}} \mid \underline{\text{value}} \mid \underline{\text{reference}}$   
 $\langle \text{type} \rangle ::= \langle \text{type name} \rangle \mid \langle \text{type specification} \rangle$   
 $\langle \text{implicit definition} \rangle ::= \underline{\text{next}} \langle \text{data name} \rangle \underline{\text{in}} \langle \text{type} \rangle \mid \underline{\text{case}} \langle \text{place} \rangle \underline{\text{in}} \langle \text{type} \rangle \mid \underline{\text{some}} \langle \text{data name} \rangle \underline{\text{in}} \langle \text{type} \rangle \underline{\text{satisfies}} \langle \text{code} \rangle \mid \underline{\text{all}} \langle \text{data name} \rangle \underline{\text{in}} \langle \text{type} \rangle \underline{\text{satisfy}} \langle \text{code} \rangle$   
 $\langle \text{place} \rangle ::= \langle \text{data name} \rangle \mid \langle \text{data name} \rangle [ \langle \text{expression list} \rangle ]$

To allow for incomplete programming SAGA places no static requirement like: In a program text, the definition of a data, type or code name must precede its use. On the other hand the execution of a SAGA program will be interrupted if such a name is used before it is defined.

Type names are defined in type definitions, and used in iteration, structure definitions, and implicit definitions. Data names are defined in structure definitions and used in iteration, implicit definitions and places. A code name is defined if it follows newcode in a program text, it is used if it occurs in a code call.

```

<expression list> ::= <expression> | <expression list>, <expression>
<expression>      ::= <place> | <atom> | <code call>
<code call>       ::= <code name> | <code name>(<expression list>)

```

Because expressions must begin with a code name, a data name, or an atom, a SAGA compiler has no difficulty with actual parameters (section 2.8.)

```

<simple code> ::= <ODIN program> |
                <structure definition> | <type definition> |
                <place> ::= <expression> | <code call> |
                <expression> = <expression> | <implicit definition>

```

### Outer language

Now we give the rules for combining simple codes into a SAGA program.

```

<code>          ::= <simple code> | <operation> | <condition> | compoundcode>
<operation>    ::= if<code>then<code>else<code> | <iteration> | <repetition> |
                select<alternative>* <code>: <code>else<code> |
                cycle<alternative>* <code>: <code>else<code> |
<alternative> ::= <code>: <code>,
<condition>   ::= not<code> | <code>or<code> | <code>and<code> |
                <code>fails before<code>

```

The SAGA distinction between operators and conditions is unnecessary, because of the conventions in section 2.1. for replacing nodes by flow-diagrams.

$\langle \text{iteration} \rangle ::= \text{for} \langle \text{data name} \rangle \langle \text{type} \rangle \text{repeat} \langle \text{code} \rangle$   
 $\langle \text{repetition} \rangle ::= \text{while} \langle \text{code} \rangle \text{repeat} \langle \text{code} \rangle \mid \text{until} \langle \text{code} \rangle \text{repeat} \langle \text{code} \rangle \mid$   
 $\quad \text{repeat} \langle \text{code} \rangle \text{while} \langle \text{code} \rangle \mid \text{repeat} \langle \text{code} \rangle \text{until} \langle \text{code} \rangle$   
 $\langle \text{compound code} \rangle ::= \text{begin}; \langle \text{statement} \rangle^* \text{end}$

Semantically until is equivalent to while not, and for  $\langle \text{data name} \rangle$  is equivalent to while next  $\langle \text{data name} \rangle$  in. We note that the metaconcepts,  $\langle \text{iteration} \rangle$  and  $\langle \text{compound code} \rangle$ , were used earlier in our description of the semantics of types and type specifications.

$\langle \text{statement} \rangle ::= \langle \text{code} \rangle; \mid \text{comment} \langle \text{symbol sequence without}; \rangle;$   
 $\langle \text{named code} \rangle ::= \text{new code} \langle \text{code name} \rangle : \langle \text{code} \rangle$   
 $\langle \text{program} \rangle ::= \langle \text{named code} \rangle^* \text{program end}$

The first code name in a SAGA program indicates the first code to be activated when the program is executed. A SAGA installation will probably have a number of reserved named codes. When a user presents a program, the installation automatically inserts these named codes before the user's program end.

## Appendix 2

### ODIN Manual

In order to define a virtual machine we must specify its storage structure, its instruction frame, and its primitives. We do this for ODIN in the first three sections of this appendix, while the final section describes how ODIN can be realized on a computer that has just one kind of storage element.

### Storage structure

In this section we describe the ODIN storage elements. For each kind of storage element we need to specify, their name, their fields, how they are addressed, and what the permissible addresses are. As SAGA is a convenient language for exact definition of virtual machines, we introduce:

```
new code the virtual machine ODIN:begin;
    associate instruction with [MODE, NAME];
    associate instruction address with [0 increment next integer];
    instruction address program;

comment ODIN instructions have mode and name fields. They are addressed by positive integers;
    associate codehead with [CODE NAME, FIRST INSTRUCTION];
    associate code head address with [0 increment next integer];
    code head address processtable;
    associate data head with [DATA NAME, CODE, DATA PLACE];
    associate data head address with [dha 0 increment next dha];

    data head address data table;
    associate activity record with [CODE, NAME, LINK, L, R, N, DATA];
    activity record creator; activity record current;
    associate activity record address with [ara 0 increment next. ara];
    activity record address activity record ring;

comment ODIN does not know that activity records form a ring via their LINK fields;
    associate data address with [da 0 increment next. da];
    data address memory;
```



```

mode=data : if undefined <name> then generate data head
            else if local name then place in location
            else generate data head and element,
mode=label : if local name then present DATA ERROR
            else generate label,
mode=wander: coroutine exit
else subroutine exit

```

new code create name:

```

if      some i in code head address satisfies
           , processtable [i, CODE NAME] = name
then    start new activity else present PROCESS ERROR

```

new code back up location: begin;

comment if SAGA provided simple arithmetic this code could be replaced by LOCATION:=NEXT-2. On the other hand this code would still work if instructions were not stored sequentially;

```

define k=NIL; define j=NIL;
repeat begin; k:=j; j:=i; next i in instruction address; end
until register NEXT=i; LOCATION:=k; end

```

new code global name: begin;

comment ODIN must search through all defined data names beginning with the most recently defined;

```

some i in [dha increment previous dha test undefined]
satisfies data table [i, DATA NAME]=name; mode:=i; end

```

new code put in LOCATION: begin;

comment when this code is activated mode contains the address of the data head for name. ODIN must change LOCATION and pay due attention to the code field;

```

LOCATION:=datatable [mode, DATA PLACE];
if undefined datatable [mode, CODE] then dummy
else begin;name:=datatable [mode, CODE];
           primitive condition or code name;end; end

```



new code local name: begin;

comment ODIN must search through the datanames for this activity.

This code does not use the fact that local datanames are distinct because of the way we define the execution of instructions in data or label mode;

some i in [dha increment previous dha test i = creator [DATA]]  
satisfies datatable [i, DATANAME]=name; mode:=i; end

new code place in location :LOCATION:=datatable [mode, DATAPLACE]

new code generate data head: begin;

comment data NIL instructions are used for formal parameters called by reference and in other situations that require an alias;  
 generate partial data head (DATANAME, CODE);  
 datatable [dha, DATAPLACE]:=RESULT; end

new code generate data head and element: begin;

comment ODIN creates a new data structure and places the address of its value in LOCATION;  
 generate partial data head (CODE, DATANAME);  
next da in data address; LOCATION:=da;  
 memory [da]:=data table [dha, DATANAME];  
 datatable [dha, DATANAME]:=name;  
 datatable [dha, DATAPLACE]:=da; end

new code generate label: begin;

comment ODIN uses instructions in label-mode so that it can refer to instruction addresses it needs to be able to do this in order to stop executing instructions sequentially (see fig. 12);  
 generate partial datahead (CODE, DATA PLACE);  
 data table [dha, DATANAME]:=name; end

new code generate partial datahead: begin; value x; value y;

next dha in datahead address;

data table [dha, x]:=program[NEXT];

next register NEXT in instruction address;

```

data table [dha, y]:=program [NEXT];
           next register NEXT in instruction address;
LOCATION:=dha; end

```

```

new code coroutine exit: begin;
comment ODIN moves around the activity record ring until it recog-
nises name or returns to its starting point;
define base=register BASE;
repeat move activity record until undefined (name)
           or name=current [CODENAME]; end
register NEXT:=current [N]; dha:=current [DATA]; end

```

```

new code subroutine exit: begin;
comment ODIN removes activity records until it recognizes name
or there are no more activity records;
repeat remove activity record until undefined (name)
           or name=current [CODENAME];
if mode=destroy then NEXT:=current [N] else NEXT:=current [L];
           dha:=current [DATA]; end

```

```

new code move activity record i begin; savestatus;
BASE:=activity recordring [BASE, LINK];
if base=BASE then destroy run program else dummy;
update creator and current; end

```

```

new code remove activity record:
if current [LINK]=BASE then destroy the virtual machine ODIN
else begin; activity record ring [BASE, LINK]:=current [LINK];
           update creator and current; end

```

```

new code start new activity: begin; save status;
comment ODIN has to create a new activity record and place it in the
activity record ring;
activity record new; new [NAME]:=name;
           new [LINK]:=activity record ring [BASE, LINK];

next ar in activity address; activity recordring [BASE, LINK]:=ar;
creator:=current; current:=new; activity record ring [ar]:=new;
register NEXT:=process table [i, FIRST INSTRUCTION]; end

```

new code save status: begin;

current [N]:=register NEXT; current[L]:=register LOCATION;  
 current [R]:=register RESULT; current [DATA]:=dha;  
 activity record ring [activity record ring [BASE, LINK]]:=  
 current; end

new code update creator and current: begin;

current:=creator; creator:=activity record ring [current[LINK]]; comment

### ODIN primitives

To save space we do not define all of the ODIN primitives in this section – indeed we only define two of the primitive conditions; end

new code primitive condition:

if select name=dummy:dummy,  
 name=undefined:RESULT=NIL  
 other: execute micro code,  
else exit  
then NEXT:=LOCATION else dummy

new code primitive operation:

select name=copy creator [L] into LOCATION: non-primitive operation,  
 name=load: use LOCATION to fill RESULT,  
 name=store: copy RESULT as directed by LOCATION,  
 name=compare: equality of constructs,  
 name=start button: prepare ODIN for SAGA program  
 other p: execute microcode  
else exit

new code non primitive operation: begin;

comment no primitive operation in ODIN changes the contents of LOCATION. Now one of the design goals of ODIN was to erode the destination between primitives and codes, so much that the user could invent new primitives at will. If he writes a code that always terminates successfully and finishes by executing

do copy creator [L] into LOCATION, then the code is none other than a new primitive operation;

register LOCATION:=creator [L];

comment ODIN has 36 primitive operations for internal register manipulations, using temporary work space and the like. Their names are copy X into Y where X and Y are one of: LOCATION, RESULT, creator [L], creator [R], current[L], current [R];  
end

new code use LOCATION to fill RESULT:

select LOCATION holds an instruction address:  
RESULT:=program [LOCATION],  
LOCATION holds an dataaddress: RESULT:=memory [LOCATION],  
LOCATION holds a codehead address:  
RESULT:=processtable [LOCATION],  
LOCATION holds a datahead address:  
RESULT:=datatable [LOCATION],  
LOCATION holds a datahead or construct element:  
RESULT:=memory [location(DATAPLACE)]  
else dummy

new code copy RESULT as directed by LOCATION:

select LOCATION holds an instruction address:  
program [LOCATION]:=RESULT,  
LOCATION holds a data address: try for size,  
LOCATION holds a codehead address:  
processtable [LOCATION]:=RESULT  
LOCATION holds a data head address:  
datatable [LOCATION]:=RESULT,  
LOCATION holds a datahead or construct element:  
memory [location(DATA PLACE)]:=RESULT  
else dummy



new code selector test: result(SELECTOR)=location(SELECTOR)

new code placefield test: begin;

LOCATION:=memory [location (DATA PLACE)];

RESULT:=memory [result (DATA PLACE)];

same value; end

new code link to follow:

select result [LINK]=NIL and location [LINK]=NIL: exit,  
result [LINK]=NIL or location [LINK]=NIL: exit equality for  
constructs,

else begin; RESULT:= memory [result[LINK]];

LOCATION:=memory[location[LINK]]; comment

We finish this section by describing the primitive operation that starts ODIN after a program has been loaded. Usually the instruction with address 0 is do start button; end

new code prepare ODIN for SAGA program: begin;

current [NAME]:=program [register NEXT]

current [LINK]:=ara0; creator:=current;

activity record ring [ara 0]:=current;

next register NEXT in instruction address;

register NEXT:=program [register NEXT]; end

program end

### Realization of ODIN on a traditional machine

In this final section we indicate how ODIN can be realized on a computer with words as its only kind of storage element. For simplicity we assume that a word is large enough to contain an instruction, an address, or an atom. Now we can describe the realization of the various kinds of ODIN storage element:

- instructions fill one word;
- code heads fill two words, one for each field;
- data heads fill four words, one for each field, and one for the address of first word for the preceding data head;
- activity records fill thirteen words;
- RESULT and LOCATION fill four words each, so they can hold the value or address of any storage element other than an activity record;
- other registers fill one word;
- construct elements fill three words, one for each field.

The next step is to drop the distinction between program, process table, data table, activity record ring and memory, and organize the storage elements as shown in figure 22 of section 3.8. The reason for the three new registers in figure 22: WORK, DATA and TOP, will become apparent later.

Let us begin by describing the realization of create name, global name, and local name. There is no problem with the first of these because the process table is in fixed space. The third starts with the storage element, whose address is in the register DATA, and continues via the links in the extra words of the data heads until it matches the given name or finds NIL in the link. The second also does this except that it repeats the process, after updating the register DATA from an activity record if it finds NIL in the link. We note that the above links contain addresses that are relative to the activity record for the current activity. We also want this to be true for the storage elements that contain the values of data structures. Suppose the volatile space contain such a value. Suppose the handle of the value is in RESULT. There is no problem if the value is an atom or a name, but what if it is a construct. This is only permitted in ODIN, if LOCATION contains the address of a

word in unstable space. If this condition is met ODIN can generate the additional unstable space it needs by moving the volatile space and updating WORK and TOP appropriately. ODIN can also use this technique to realize: generate data head and element generate data head, and generate label.

Let us continue by describing the realization of move activity record, remove activity record, and start new activity. First we describe why an activity record fills 13 words: The L- and R fields need 4 words, the other fields need one word each, and we need one word to store the exit value of WORK. Then we describe the new register CURRENT (CREATOR) which contains the address of first word of current (creator) activity record. Now we can describe remove activity record and start new activity. The former usually copies CREATOR into CURRENT and updates CREATOR and the link field of the bottom activity record; the latter puts the address of a new activity record in CURRENT and prepares ODIN for transfer to a new activity. Finally we describe move activity record. Because we have relativized most addresses to an activity record, this can be realized by: move bottom activity record and its space on top of the space for the current activity, collapse the vacant space, then change the data, work and link fields of all activity records. If we were to allow volatile space below other kinds of space, instructions in wander mode could be realized more efficiently. This does not seem to be a good idea because it makes the realization of more valuable parts of ODIN clumsy and inefficient.

Let us conclude by describing how code and data heads can be omitted from an ODIN realization. The advantages are obvious: we eliminate the need for possibly timeconsuming searches of the process and data table. Sometimes the advantages outweigh the disadvantages, so elimination of data and code heads should be a programmer option.



Suppose we allow instruction to have not only code names but also addresses of other instructions in their name fields. Then the running program need never use a code name because the contents of its **FIRST INSTRUCTION** field is known at load time. Can we use this technique to eliminate data heads too? Almost. There is no difficulty if all data heads are defined at the beginning of an activity and all have empty code fields. In this case the address of a data element relative to its activity record is known at load time and can be used in the namefield of an instruction instead of a dataname. We could provide a **DISPLAY** vector to handle non-local data addressing and the result would resemble the usual addressing mechanisms used for languages with recursion ( ). Even the presence of a code field in a data name gives no trouble: the first bit in each data element word indicates whether or not the following word is a code name. However the requirement that all data heads are defined at the beginning of an activity is a serious limitation. Not only is it illogical to collect data definitions at the beginning of a code text (as declarations) instead of where they are used, it is also inefficient to reserve memory space which will not be used. The reader may protest that this is a small price to pay for the advantages of dropping the process and data tables. But are these advantages so very great in the usual case of a large program on a small virtual machine. Such storage administration mechanisms as paging require some sort of data (process) table to say if the desired data (code) is immediately available or must be fetched. Furthermore we should not forget the virtues of satisfactory error recovery and incomplete programming: ODIN needs code and data names in order to converse intelligently with the user at a terminal.

## BIBLIOGRAPHY

1. T.E. Cheatham, The recent evolution of programming languages, Proc. IFIP 1971, N. Holland.
2. O.J. Dahl, K. Nygaard, Simula, an algol-based simulation language, Comm. ACM 9 (1966) 671-678.
3. E.W. Dijkstra, EWD 249, Notes on structured programming, 70WSK03, Technical University, Eindhoven, 1970.
4. A. Evans, PAL, a language designed for teaching programming linguistics, ACM conference proceedings, 1968.
4. J. Earley, Towards an understanding of data structures, Comm. ACM 14 (1971) 617-627.
5. A. Evans, PAL, a language designed for teaching programming linguistics, ACM conference proceedings 1968.
6. J.A. Feldman, P.D. Rouner, An algol-based associative language, Comm. ACM 12 (1969), 439-449.
7. D.A. Fisher, Control structures for programming languages, Ph.D. thesis, Carnegie-Mellon University (1970) AD 708511.
8. J.M. Foster, E.W. Elcock, P.M.D. Gray, J.J. Mebreger, A.M. Murray, ABSET, a programming language based on sets: Motivations and examples, Machine Int. 6 (1971), 467-492.
9. B.A. Galler, A.J. Perlis, A view of programming languages, Addison Wesley 1970.
10. J.V. Garwick, GPL, a truly general purpose language, Comm. ACM 11 (1968), 634-638.
11. ed. F. Genuys, Programming languages, Academic Press 1968.
12. C.A.R. Hoare, Record handling, in [10].
13. H.F. Ledgaard, Ten minilanguages: a study of topical issues in programming languages, Comp. Surveys 3 (1971) 115-
14. P. Lindblad Andersen, J. Jensen, P. Jensen, J. Steensgård Madsen, GROK, unpublished manuscript, Copenhagen University 1972.
15. C.H. Lindsay, S.G. Van der Muelen, Informal introduction to Algol 68, N. Holland 1971.
16. P. Naur, Programming by action clusters, BIT 9 (1969), 250-258.
17. A. Perlis, The synthesis of algorithmic systems, JACM 14 (1967), 1-9.

18. J.L. Reynolds, GEDANKEN - a simple typeless language based on the principle of completeness and the reference concept, Comm. ACM 13 (1970), 308-319.
19. A.M. Richards, BCPL: a tool for compiler writing and system programming, SJCC proceedings 1969.
20. R.A. Snowdon, PEARL, an interactive system for the preparation and validation of structured programs, SIGPLAN notices (1972), 9-26.
21. C.C. Strachey, Fundamental concepts in programming languages, to be published.
22. A. Wang, O.J. Dahl, Coroutine sequencing in a block structured environment, BIT 11 (1971), 425-449.
23. A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, Report on the algorithmic language ALGOL 68, Numerische Mathematik 14 (1969), 79-118.
24. M.V. Wilkes, The outer and inner syntax of a programming language, Comp. J. 11 (1968), 260-263.
25. N. Wirth, Program development by stepwise refinement, Comm. ACM 14 (1971), 221-227.
26. N. Wirth, Programming and programming languages, Int. Comp. Symposium, Bonn, 1970.
27. N. Wirth, The programming language Pascal, Acta Informatica 1 (1971), 35-63.
28. W.A. Wulf, D.B. Russell, A.N. Habermann, BLISS: a language for systems programming, Comm. ACM 14 (1971), 780-790.
- 29.
30. Per Brinch Hansen, The nucleus of a multiprogramming system, Comm. ACM 13 (1970), 238-250.
31. P.J. Denning, Third generation computer systems, Comp. Surveys 3 (1971), 175-216.
32. E.W. Dijkstra, Hierarchical ordering of sequential processes, Acta Informatica 1 (1971).
33. A.G. Fraser, On the meaning of names in programming systems, Comm. ACM 14 (1971), 409-416.
34. Project MAC conference on concurrent systems and parallel computation, ACM 1970.
35. J.G. Mitchell, the design and construction of flexible and efficient interactive programming systems, Ph. D. thesis, Carnegie-Mellon University (1970), AD 71 2721.

36. B. Randell, L. J. Kuehner, Dynamic storage allocation systems, Comm. ACM 11 (1968), 297-306.
37. J.E. Stoy, C. Strachey, OS6 - an experimental operating system for for a small computer, to be published.
- 38.
- 39.
40. R.W. Doran, A computer organization with an explicitly tree structures language, Austr. C.J. 4 (1972), 21-30.
41. P. Kornerup, Bruce Shriver, MATHILDA, A versatile unit for emulation enhancement, unpublished manuscript, Aarhus University 1972.
42. J.K. Illiffe, Basic Machine Principles, McDonald, Wiley 1971,
43. H.W. Lawson, Programming language oriented instruction streams, IEEE trans. c-17 (1968), 476-485.
44. O.B. Madsen , KAROLINE, a network computer project, RECAU 14-72, Aarhus University 1972.
45. R.F. Rosin, Contemporary concepts of microprogramming and emulation, Computer Surveys 1 (1969), 197-212.
46. R.F. Rosin, M.J. Flynn, Microprogramming : an introduction and a viewpoint, IEEE trans. c-20 (1971), 727-731.
47. A.B. Tucker, M.J. Flynn, Dynamic microprogramming: processor organization and programming, Comm. ACM 14 (1971), 240-258.
48. M.V. Wilkes, The growth of interest in microprogramming, Comp. Surveys 1 (1969), 139-145.
- 49.
50. E. Ashcroft, Z. Manna, The translation of 'goto' programs to 'while' programs, IFIP proceedings, Ljubiana 1971.
51. C. Bron, Outline of a machine without branch instructions, Inf. Proc. L. 1 (1972), 117-119.
52. E.W. Dijkstra, Goto statements considered harmful, Comm. ACM 11 (1968), 147-8).
53. D. Gries, Compiler construction for digital computers, Wiley 1971.
54. D.E. Knuth, R.W. Floyd, Notes on avoiding 'goto', Inf. Proc. L. 1 (1971), 23-31, (1972) 177.

55. B.H. Mayoh, The elimination of datastructures, to be published.
56. P. Naur, The design of the GIER algol compiler, part 1 BIT 3 (1963), 124-140; part 2 BIT 3 (1963) 145-166.
57. M. Richards, The portability of the BCPL compiler, Software - practice and experiences 1 (1971), 135-146.
58. N. Wirth, The design of a PASCAL compiler, Software - practice and experience 1 (1971), 309-333.
59. W.A. Wulf, Programming without the 'goto', IFIP proceedings Ljubljana 1971.