Context Dependent Names.

Q: "Who is there?"
A: "It is me."

Introduction.

Under regrettable circumstances I can truthfully make the statement: "My nose is bleeding.". Having only one nose I have identified the bleeding object considerably better than in a statement such as "My finger is bleeding.", the bleeding nose is even uniquely identified if ..... I were the only person or animal existing. This not being the case, the question "Which nose is bleeding?" can be raised and the proper answer would be "Edsger W.Dijkstra's nose is bleeding.". Having a very rare Christian name I can safely assume that my name identifies my person uniquely in the population of this world, and my second statement then identifies the bleeding nose among the human noses of this world. Of course one can then ask "Which world?" etc. Our conclusion must be that identification is a process of a recursive nature, that names will only identify an object, provided it is known in which context this name is to be understood; in actual fact, each identification stops with the proviso "If you understand what I mean.".

In automatic data processing, particularly in the aspect of it that I have called "Information Management" we are faced with the problem how to identify objects, how to name them and how to keep track of the context in which such a name is to be understood.

A potential way out is to decide, once and for all, upon a universe of discourse, a context in which each object is uniquely identified and to performs the processes in terms of those universal identifications. However, if I have to take care of my bleeding nose, it seems a somewhat roundabout way of doing if I first translate this task into "Take care of the nose of Edsger W.Dijkstra.", facing myself thus with the preliminary task to select from the world's population the human being thus identified. On the contrary, taking care of my bleeding nose is an action that I can take, unconscious of the presence of other fellow human beings and irrespective of how I am identified among them.

The problem to be solved is whether we can find a suitable technique for identifying objects with names that will have a meaning in the context in which they occur, without making any assumptions about the total size of the universe in which the process in question happens to be embedded, i.e. by keeping the identification problem at that side open ended.

To be more concrete: I may have two independent programs in both of which occur thirty objects, within each program suitably identified by the numbers (used as names) from 0 through 29. If these programs are sound by themselves, they should without any recoding remain so, when the two programs are executed concurrently in a multiprogrammed machine.

For the time being I shall stress the logical structure of such an identification technique, intentionally disregarding many aspects of efficiency. Once the structure stands clear, we may hope to see, where additional equipment can be used at good advantage.

Starting points.

There is no point in trying to conceal the fact that multiprogrammed systems are one of my major concerns.

Years ago the idea was that each individual user had at his private disposal a von Neumann kind of memory, addressed by positive integers, starting at zero up to a certain limit, that often had to be given a priori.

Embedding many such programs within a single machine consisted of allocating contiguous areas of memory space to each of the programs. The transition from address, as mentioned in the program, to the physical location in store consisted of the addition of a so-called "base address", viz. the starting point of the memory area alotted to the program in question.

Let us review this technique. Its kernel implies that by the process -either from the text as stored within the computer or as a computed intermediate result- a non-negative value is supplied to the selection process, which then establishes (by the addition of the appropriate base address) the identity of the object identified.

I feel a strong urge to try to retain the integers as the kernel of our identification technique. When elements of a set are numbered (from zero onwards) we shall call the integer values with such a specific role of identification within the set "an index".

There are two reasons to try to use indices. The first reason is that from the users point of view they have proved to be extremely useful. If a procedure body starts with the declaration

"real x,y,z"

it is for a translator a trivial renaming to generate for them in order the index values "0", "1" and "2", thus creating an object text that produces indices quite naturally. Arrays and stacks are well-known (and essential!) programming devices, in the use of which the index values are part of the subject matter of the process itself. When keeping the role of the index as central as possible, we may in case of doubt derive comfort from the respectable company of von Neumann and Peano.

The second reason is founded upon the possibility of implementation: modern electronic equipment is remarkably efficient in performing additions. In the following I shall refrain from speculations about large associative memories, and therefore the index controlled selection is taken as the kernel of the identification mechanism. The act of the selection under control of a single index shall be called "a primitive selection".

Such a primitive selection, however, can only be performed provided that it is understood with regard to which population of numbered elements the index value supplied should be used, e.g. provided that the base address to be incremented by the index is known. For lack of a better term I shall call the contents of this understanding "the condition of the primitive selection". The condition of a primitive selection sees to it, that the index supplied to it is related to the proper population.

So we assume indices to be here to stay. Let us now return to the multiprogramming technique described earlier, in which each individual user has a single set of consecutive integers at his disposal, the idea being that each individual user can now program as in a classical von Neumann type memory, starting at address zero and of the appropriate size. To my feeling there is something wrong in this approach. For one thing: the duty to embed the information in a reasonably well filled linear store often imposes a considerable burden on the shoulders of the individual programmer. And after each programmer has taken this trouble, the system as a whole is faced with the task to find a sufficiently large consecutive area available for such a program!

In other words: within such a system the condition of the primitive selection -see above- may differ from selection to selection; this facility, however, should not be restricted in purpose to distinguish between indices originated in different programs, but we should try to place this facility also in the hands of the individual programmer.

Locally total selections.

Suppose that a programmer operates upon two integers "i" and "j" and a vector, "k" say, itself of integer elements. The use of an integer value ~~indicating~~ as an index indicating by enclosing it within square brackets, we can rename

"L[0]" for "i",
"L[1]" for "j" and
"L[2]" for "k" .

(For the time being the capital letter "L" in front of the sqare opening bracket can be regarded as a noise symbol.)

In the case "L[0]" or "L[1]" it is clear, that the integer variable named "i" or "j" has to be selected; "k", however, identifies a vector and in order to select an element of this vector, we must supply a second index such as in

"L[2][0]" or "L[2][7]" .

If the zeroth and seventh element of the array "k" exist as integers variables, we may assume these integer variables suitably identified by the given successions of two index values.

I regard the processing of such a sequence of index values as a sequential process, for the value of the first index determines, whether the second one is applicable. It is then a succession of two primitive selections, the outcome of the first being nothing else than what we have called earlier "the condition of the next primitive selection".

I do not regard the condition of the next primitive selection as belonging to the subject matter of the precess, i.e. the information upon which can be operated at programmer's whim: it can only be used..... to control the next selection.

This forces upon us the recognition that the three primary objects, named "i", "j" and "k" are of vastly different nature.

The first two I call "terminal objects": their value, being integers, is part of the subject matter of the process, when they are selected no next index is permissible.

The last one I call "a non-terminal object": its "value" -in whatever domain expressed- is the condition of a next primitive selection. The selection of a non-terminal object requires a next index value to be supplied (and so on), until a terminal object is reached.

Remark 1. This formalism allows a distinction between the terminal object "L[3]" and the set "L[3]" having "L[3][0]" as its only (terminal) member.

I call a total selection the succession of one or more primitive selections, until a terminal object is reached. To be a little bit more explicit, I shall call them "locally total selections", leaving the question open, whether previous primitive selections are to be understood to condition the selection by the leading index.

Remark 2. I have strong reasons for my refusal to regard "the value of a non-terminal object" -i.e. the condition for a next primitive selection- as part of the subject matter of the process, more in particular as data of which arbitrarily many copies can be made at programmers desire. Within a certain context, the non-terminal object is invariantly identified by the index value (sequence) leading to its selection: within this context its value has only a meaning for the next primitive selection. If we ask, how this value is represented, we ask a question that is meaningless in terms of the subject matter of the process itself, but that can -and must!- have a meaning in the immediately surrounding universe in which this process is embedded: it may contain, say, a base address.

In the surrounding universe, it may be desirable to change the way in which the process is embedded: then one makes changes which are irrelevant to the process itself. On the other hand, such changes will result in updating of non-terminal values. On account of this lack of invariance of meaning, I feel obliged to disallow any uncontrolled copying of non-terminal values.

(Obviously: under the assumption of the declaration "real x,y", the assignment

"x:= y"

assigns to the variable "x" the current value of "y", i.e. an object, most explicitly independent of the future history of the variable y! It can be implemented thanks to the eternal existence and invariant representability of numbers.)

The context.

To the implicit conditioning of the selection by the leading index we give the name "the context".

By having different contexts we can attain that the same index values may point to distinct objects, a vital facility.

In the nomenclature used in a program or program part, we may rightly require the "resloving power of the nomenclature" to be sufficient to distinguish between the different objects that this (part of the) program could possibly refer to; it is irrealistic to require a resloving power able to distinguish between all objects ever named, now or in the future. Nevertheless we can view this program as embedded in the collection of all programs being written!

In other words: we allow different programmers to use the same identifications for local usage and if such an identifier is used in more than one meaning, this is an accident of combinatorial nature: the fact that their different programs happen to have been written.

The combinatorial nature of the origin of the need for more than one context is most clearly demonstrated by a closer inspection of the mechanisms neede for procedure execution.

Taking for granted the availability of some primitive "whistle", we can write a procedure, "twentywhistle", say, that performs the action "whistle" 20 times in succession:

```
"procedure twentywhistle;
 begin  integer i; i:= 20;
        while 0 < i do  begin whistle; i := i - 1 end
 end"   .
```

We can regard the procedure body as the constant value of the object named "twentywhistle" and we may talk about its life time. Within its life time it can be in action and it will be in action as many times as it is called.

With each call, however, the creation of a new variable "i" must take place, an object that will belog to, and the life time of which will extend over, the execution of "twentywhistle" as a result of this call. At the return, the local variable will be destroyed. In other words, it is the call of twentywhistle (and not its definition) that creates the variable as an object. In general we are bound to do so, as we have no guarantee that the intersection of the different periods of execution of "twentywhistle" will be empty. In multiprogramming, where different sequential processes may invoke the same procedure, such an overlay of execution times may occur by accident; in uniprogramming such an overlay is generated by structure each time a procedure invokes itself, a thing that recursive procedures generally do.

It is very tempting to regard procedure call and procedure return as actions of a very special nature, viz. as actions in which a new context is born and annihilated respectively. The declarations at the beginning of the procedure body are the commands controlling what context will be born.

(Note. I have reasons enough to give the procedure concept a central position in my considerations:
1) in 1962 (An attempt to unify the constituent part concepts of serial program execution) I have shown that the most natural conception of a variable is a procedure that upon call produces the value assigned to it.
2) in the multirunning system we are constructing at present, the status of a user program is that of an (anonymous) procedure, whose body is defined during "load" and that is called at "go".

The procedure concept thus covers a wide spectrum!)

Context Identification

0. Introduction.

We, i.e. somewhat external observers, can take a snapshot of the full program status somewhere in the middle of its execution, say when the program has called procedure A, which has called procedure B which has called procedure C. We can then observe the existence of four different contexts: the one that was the current one before A was called and that will restored to this status after A's completion, and those belonging to the activations of A, B and C. (Note: If A happens to call itself, e.g. when C = A, we can still distinguish four different contexts!)

On account of their nicely nested lifetimes they can conveniently be identified with the elements of a stack; thus giving the outer context the index value "0", the one belonging to A the value "1", the one belonging to B the value "2" and the one belonging to C the value "3".

Let us call this "the stack of activations"; upon procedure entry an element is added to it, upon return its top element is unstacked.

Such a stack of activations becomes an object associated with a single sequential ~~process~~ process. In a multiprogramming system one could consider a single "activation list" for the union of all activations of all programs running in the system. The objects -i.e. activations- will then no longer have nicely nested life times, but something could be done about that, e.g. giving to each new activation the lowest free index available. I most emphatically propose not to do so: creation of a new context implies then the analysis what index should be used for its identification in the total "activation list", and as a result concurrent context creations would have to exclude each other mutually in time. We could also remark that the introduction of the different stacks (one for each sequential process) is nothing more than the creation of a terminology that exploits nicely the life time relations as far as they are present; essentially this is the same argument.

It may be clarifying to inspect the consequences of the parallel processing as described in "Cooperating Sequential Processes", where the different constituent statements of a so-called "parallel compound" are activated in parallel and should be executed concurrently. Instead of a single element to be stacked as in the case of the procedure call, a number of elements have to be stacked together; associated with these elements is not a context (as in the case of a procedure call) but a new stack of activations, viz. one for each of the constituent parts of the parallel compound. We shall return to this in greater detail below.

Now a number of questions should be raised (and answered in some sense) e.g. "Why do we want to identify contexts by indices" and "Of what nature will the value be to be found in the activation stack?". To the last question we can give our usual vague answer "It will be the condition of the next primitive selection."

The need to identify them becomes apparent if we pay attention to the parameter mechanism, for instance the ever present implicit parameter: the return information. One of the parameters of a procedure shall be what context to restore to its status of "the current one" upon return!

1. The Stack of Activations.

In a single sequential process our activations have nested lifetimes and we can assume the values specifying the corresponding contexts to be stacked.

To my taste the difference between a stack and a push down store is that in the push down store accessibility is by definition restricted to the top element, whereas in a stack, interior elements may be accessed. That is, we regard a stack as a vector with a moving upper bound

"array stack[0 : MI]"

(where MI stands for "Maximum Index") in which terminology "to stack" can be expresses as

"MI:= MI + 1; stack[MI]:= "element to be stacked"".

Thus we ensure that each stack element is uniquely identified -within the stack and during its lifetime- by the index value giving its distance from the stack bottom.

If an expression to be interpreted under the current context is to be handed over as a parameter to a procedure with the effect that its actual interpretation will take place at a stage where the context relevant to the parameter is no longer the current one -but at that stage may be deep down the stack- then the context can be identified by its indexvalue in the stack of activations: this is an invariant identification in contrast to its distance from the then current top of the stack.

Within the program, the actual value of MI is without any meaning! The only operations in which MI plays a role are increase of it, decrease of it and selection under control of it or under control of a copy of an earlier value. It are these operations that have a meaning within the program, the interpretation of the values, identifying contexts, falls outside the scope of the context itself. It is like making with you an appointment for "tomorrow": I request you to increase the current date by 1, it is irrelevant whether you translate this in terms of the Julian or the Chinese calendar; I may process this appointment without calendar, such as children do "after one night sleeping".

During specification of parameter, at entering into or returning from procedures, the process has to operate secundum regulas artis upon the MI, associated with the implicitly understood stack of activations. The copies made from such MI values will only be used (to be copied or) to control the selection of the contexts. Such an MI is not a local variable of the process, it is a variable in whose value the process dutifully reports its behaviour.

I regard the MI, i.e. the stack length as part of the condition under control of which an element of this stack can be selected, because it gives the upper bound of the index values that can be processed by the next primitive selection.

2. The Tree of Activations.

Within a sequential process including procedural processing a stack of activations is implicitly understood. We shall now try to visualize the entrance of an n-fold parallel compound.

Then n new elements will be added (simultaneously) to the current stack of activations, each for one of the parallel statements; when the execution of the n parallel statements have all been completed, the n elements will be removed together and the stack of activations from which they have been removed will return to its former status of "the current one".

Upon creation each of the n elements will get as value the condition for selection in a new stack, that also will be created (either empty or suitably initialized). The n stacks thus created will act as the current stack of activations for the n different parallel statements. The n different stacks to be implicitly understood during the activity of the parallel compound are thus elements with an identity within the stack from which they blossomed out; the actual index values by which each stack is identified depends on the momentaneous filling of the "mother stack" at the moment that the parallel compound has to be entered.

For the interpretation of the n parallel constituents of the compound -i.e. the sequence of primitive selections to be performed before the appropriate context is identified and the next primitive selection suitably conditioned- one primitive selection more than before blossoming must be done: if before the entrance of the parallel compound a sequence of DC index values was sufficient to identify the current context within the universe, then now DC+1 index values will be needed.

This DC is a variable, gravely analogous to MI ("DC" being short for "Depth Counter"). During the lifetime of a stack of activations a constant DC-value is associated with it; the stacks created upon entry of a parallel compound will all have a DC-value associated with them that is one higher. The actual value of DC depends on the distance from the root of the univers, but inside the processes this value is as meaningless as the distance of the bottom of the stack.

In the previous section we have shown that as long as we have one sequential process, the current MI value will be sufficient to identify this context for further reference. If on top of this context parallel blossoming occurs, we do no longer know, to which stack of activations the fixed MI value refers. This problem is now solved by identifying the current context by a combination of the current values of MI and DC.

Thus we have achieved that contexts can identify themselves in a terminology independent of the size, complexity and total hiearchical structure of the universe in which they are to be understood.