Sequencing primitives revisited.

Let "db" equal the number of boys in the common dressing room of a swimming pool; the initial value of db obviously equals zero. Using the operators "inc" and "dec" respectively for increase and decrease by one of their integer argument(s), we can describe the life of a little boy --omitting all further irrelevant details-- as the cyclic process

do inc(db); dec(db) od

where the repeatable statement describes a single usage of the dressing room. And if we have a community of a hundred of such little boys, we have a hundred of such sequential processes.

There is, however, another way of looking at the usage of the dressing room. From the point of view of the dressing room --which can be assumed not to be awfully interested in the identity of the boys using it, there are just two sequential processes

do inc(db) od and do dec(db) od ,

describing what happens at its entry and at its exit respectively. These two processes, however, have to be synchronized so as to maintain the invariance of

$db \geq 0$.

(From the point of view of the dressing room we can even regard the physical boys as a means for implementing this synchronization restriction: they are very similar to the coins in one's purse!)

As a second example, consider the well-known version of Euclid's algorithm for the computation of the greatest common divisor GCD(A, B) of two positive numbers, using addition and subtraction only:

```
a:= A; b:= B;
while a ≠ b do if a > b then a:= a - b
                        else b:= b - a fi od;
print(a)
```

The first line establishes the relation "GCD(A, B) = GCD(a, b)" and both operations "a:= a - b" and "b:= b - a" leave that relation --for such are the properties of the GCD-function-- invariant. Furthermore, the first

line establishes $a > 0$ and $b > 0$

and also this relation is kept invariant; the loop terminates when a = b and then, because GCD(a, a) = a , we know the answer.

Also this program, however, admits another interpretation, viz. the two cyclic processes

do a:= a - b od and do b:= b - a od

have to be synchronized in such a fashion that the relation

P: $a > 0$ and $b > 0$

is kept invariant.

Let us compute the synchronizing conditions that follow from this requirement. Using the notation

wp(S, P)

for "the weakest precondition such that its truth prior to the execution of S will guarantee that the execution of S will establish the truth of the postcondition P", then from the axiom of assignment follows

wp("a:= a - b", $a > 0$ and $b > 0$) = $(a - b > 0$ and $b > 0)$

wp("b:= b - a", $a > 0$ and $b > 0$) = $(a > 0$ and $b - a > 0)$.

For each of the two statements we have here the necessary and sufficient precondition to establish the truth of the postcondition P. We are, however, only interested in maintaining P, that is, for each of the statements S, we are interested in the additional condition C, such that

(P and C) => wp(S, P) .

The computation of C is simple, but I shall explain it slowly, as I can only have heart-felt sympathy with any reader who, through unfamiliarity, gets confused. (Students and staff from Aarhus, Denmark, may remember how, early 1973, I got completely entangled during one of my lectures, when lack of preparation could not be compensated by experience!)

One uses the following theorems of propositional calculus

1) From P => C2 follows (P and (C1 and C2)) = (P and C1)

2) From P => non C2 follows (P and (C1 or C2)) = (P and C1) .

As first form of C, satisfying

$$(P \text{ and } C) \Rightarrow wp(S, P)$$

one chooses C equal to wp(S, P) itself and if C is a conjunction, theorem 1 allows us to drop any term that is implied by P and if C is a disjunction, theorem 2 allows us to drop any term whose negation is implied by P. In this example we only need theorem 1 and we arrive at

$$(P \text{ and } a > b) \Rightarrow wp(\text{"a:= a - b"}, P) \qquad \text{and}$$
$$(P \text{ and } b > a) \Rightarrow wp(\text{"b:= b - a"}, P) \quad .$$

And it is now very tempting --so tempting in fact, that I shall do it-- to write our program for the greatest common divisor as follows:

```
a:= A; b:= B;
do a > b: a:= a - b,
   b > a: b:= b - a od;
print(a)
```

We call a construct like "a > b: a:= a - b" a "guarded command" and for the time being we suggest the following syntax

< guarding head > ::= < boolean expression > : < statement >

< guarded command > ::= < guarding head > { ; < statement > }

where the braces {...} should be read as "followed zero or more times by the enclosed --in our example above, this is zero times--

< guarded command list > ::= < guarded command > { , < guarded command > }

The guarded command list is a semantically unordered list of guarded commands, separated by commas. (We have sticked to the usual convention to use the comma as separator in a list in which the relative order of the elements separated is semantically irrelevant, the above program could equally well have been written with

```
do b > a: b:= b - a,
   a > b: a:= a - b od        .)
```

A guarded command is "executable" when its associated guard --here the boolean expression with which the guarding head starts-- considered as a function of the initial state is true.

Basically we propose two constructs with guarded command lists. The first is a possible form of a statement:

< statement > ::= if < guarded command list > fi

If one or more of the guarded commands is executable, the execution of the whole construct consists of the execution of one of the executable guarded commands. If none of the guarded commands in the enclosed list is executable, the program is wrong --and in attempted execution we assume program abortion. Note that there is no point in introducing the default convention that, if none of the n alternatives mentioned in the list is executable, the whole construct reduces to the empty statement. If we wish to provide that alternative, we should add it as the n+1st alternative and write instead of

if B1: S1, ... , Bn: Sn fi

rather

if B1: S1, ... , Bn: Sn, non(B1 or ... or Bn): Slast fi

where Slast can be the empty statement. For the above we propose --but this can be regarded as syntactic sugar-- the abreviation

if B1: S1, ... , Bn: Sn else Slast fi ,

i.e. we also allow the format

< statement > ::= if <guarded command list > else < statement > fi .

If one or more guarded commands from the list are executable, one of them will will be executed, otherwise the statement following "else" .

What used to be written as if B then S1 else S2 fi can now be written

if B: S1, non B: S2 fi

or, with our last notational extension as

if B: S1 else S2 fi .

(In doing so, we can imagine ourselves to serve a dual purpose. On the one hand the use of else can be regarded as a hint to the implementation: after the evaluation of B the value of non B is known as well. On the other hand it is an explicit expression (for us!) of the fact that at least one of the alternatives is executable, stronger, thatif none of the preceeding ones is executable, the last one is and that, if the last one is executable,

it is the only one. In the following it will become apparent that the use of else is often to be avoided.)

Note that what we used to write as

if B then S fi

should be written now as

if B: S else fi

and not as if B: S fi

because in the old notation the latter would correspond to

if B then S else ABORT fi .

Secondly we note that in the case that more than one of the guarded commands in the list is executable, we have left undefined which of the ones will be chosen for execution. This is done intentionally; in this respect we are proposing what could be regarded as a non-deterministic machine. On the one hand we have the duty to see to it that this non-determinacy is only introduced "when it does not matter", on the other hand --and that is at this stage more important-- whenever "it does not matter" our notation does not force an arbitrary or premature decision upon us.

The second construct with guarded command lists that we propose is

< statement > ::= do < guarded command list > od .

Here we do allow that the alternatives are exhausted, i.e. it is not a fatal error if none of the guarded commands is executable: in the case of exhaustion, the execution of the whole do ... od construct is regarded as completed. The complementary rule, however, is that this is the only way in which the do ... od construct can terminate and that, as long as at least one of the guarded commands is executable, one of the executable ones will be selected for execution. In the case of more than one executable command, it is again undefined which one will be selected, we postulate, however, that then they will be selected in "fair random order", i.e. we disallow the non-determinacy permanent neglect of a permanently executable guarded command from the list.

We have to postulate this, because in the construct

do B1: S1, B2: S2 od

we would not like to exclude --nor to advocate for that matter!-- that both B1 and B2 might be true, while S1 has reduced itself to the empty statement and the "possible" execution of S2 will eventually cause both guards to become false. In that case we cannot allow the non-determinacy to be so bad as to choose the executable first alternative S1 all the time.

Again, if more than one of the guarded commands is executable we have what we can regard as a non-deterministic machine. We have to see to it that whenever we introduce this non-determinacy, "it does not matter", on the other hand "whenever it does not matter" our notation relieves us from the duty to make an arbitrary choice.

But even in the case of mutually exclusive B's --i.e. no non-determinacy-- the text

do B1:S1, B2:S2 θd

seems to have advantages. In the older notation we would be forced to choose between one of the following versions

1)
```
while (B1 or B2) do
    if B1 then S1 else S2 fi
od
```
more or less obscuring the fact that S2 will only be executed provided B2 holds initially

2)
```
while (B1 or B2) do
    while B1 do S1 od;
    while B2 do S2 od
od
```
which is not too bad, but not too nice either --with the possible execption of the first outer repetition, for instance, the first inner loop will always be executed at least once.

3)
```
repeat ready:= true;
    while B1 do S1; ready:= false od;
    while B2 do S2; ready:= false od
until ready
```

Clearly our alternatives are getting nastier and nastier. Besides that, without any further information there is no reason to assume that our new version with the guarded commands would lead to a greater number of evaluations of the boolean expression B1 and B2.

* * *

As we have pointed out, in the case of a non-deterministic choice between two or more guarded commands, we must ensure that "the choice does not matter" and before proceeding with more examples, we had better be sufficiently clear about

1) what we mean by "does not matter" and
2) how we ensure that fact.

There is a snag about non-determinacy. We recall that we denote by

wp(S, R)

the weakest precondition such that its truth prior to the execution of S will guarantee that the execution of S will establish the truth of the postcondition P; in particular it guarantees that the execution of S will terminate successfully, which in general could be prevented by either endless looping --e.g. do true: S od-- or by abortion --e.g. if false: S fi--. Proper termination is guaranteed if the initial state satisfies wp(S, T), where "T" is used to denote the condition that is satisfied by all states. The most important properties are

1) For any S, wp(S, F) = F , where "F" is used to denote the condition that no state satisfies. (This is called the "Law of the Excluded Miracle".)
2) For any S and any Q and R such that Q => R , we have wp(S, Q) => wp(S, R) . (As a result wp(S, Q) => wp(S, T) for any Q.)
3) For any S, Q and R we have (wp(S, Q) and wp(S, R)) = wp(S, Q and R)
4) For any S, Q and R we have

if S is deterministic: (wp(S, Q) or wp(S, R)) = wp(S, Q or R)
otherwise only: (wp(S, Q) or wp(S, R)) => wp(S, Q or R) .

In the fourth relation, the non-determinacy replaces the equality by an implication. If initially wp(S, R) is not true, the truth of R after execution of S is not excluded.

Note. For a given S, wp(S, R) is uniquely defined for any postcondition R, even if S is non-deterministic!

Finally I introduce --inspired by, but possibly deviating from, C.A.R. Hoare-- one further notation, viz.

$$\{P\}\ S\ \{R\}$$

to be read as "P is with respect to S a safe precondition for the postcondition R" and asserting that the truth of P prior to the execution of S is sufficient to guarantee that S cannot terminate without establishing the truth of R. Note that in this assertion we do not guarantee that S will terminate properly, we don't assert that S will produce the desired result, we only assert that S won't produce the wrong result. We can relate this assertion to our previous formalism:

$$\{P\}\ S\ \{R\}$$

~~is equivalent to the statement that P is a solution of the equation~~ implies

$$(wp(S,\ T)\ \underline{and}\ P) \Rightarrow wp(S,\ R)\ .$$

There are two typical applications of the concept of "a safe precondition". Firstly, if we can establish a safe precondition and can establish separately that the execution of S will terminate properly, then we have derived a sufficient precondition --not necessarily the weakest one, but often we don't need that, so why bother? The advantage is that the reasoning establishing the safeness of the precondition now need not be encumbered by arguments establishing proper termination (the latter arguments often being of quite a different nature). Secondly, we may be quite content with the assertion itself as we are often content with programs that either do the job or "give up" --e.g. a compiler refusing a source program, the compilation of which violates some capacity limits--, provided that we can be certain that "giving up" will be exceptional and not desastrous.

For our constructs with guarded commands we can now formulate the following properties.

Let S be "if B1:S1, ... , Bn:Sn fi" and let for $1 \leq i \leq n$ hold: $\{P\ \underline{and}\ Bi\}\ Si\ \{R\}$, then we can assert about S: $\{P\}\ S\ \{R\}$. Note that S may fail to terminate properly because none of the guards is true, or

because the guarded command selected fails to terminate properly.

Let S be "do B1:S1, ... , Bn:Sn od" and let for $1 \leq i \leq n$ hold: {P and Bi} Si {P}, then we can assert: {P} S {P and non(B1 or ... or Bn)}. Note that also this S may fail to terminate properly, but here either because the state with no guard true fails to occur, or because a guarded command selected fails to terminate properly.

Let S be "if B1:S1, ... , Bn:Sn fi and let for $1 \leq i \leq n$ hold: (P and Bi) => wp(Si, R), then (P and (B1 or ... or Bn)) => wp(S, R) . Here we assert termination.

To give an example. Let S be

```
do y ≠ 1: x:= x + 1,
   y ≠ 1: x:= x - 1,
   y ≠ 1: y:= 1 od
```

Above we have made explicitly the rule, that a loop of this type terminates. What, however, in the mean time has been done with x is in general undefined. I venture the following assertions about S:

wp(S, x = x0) = (y = 1 and x = x0)

wp(S, y ≠ 1) = F

wp(S, y = 1) = T

and this is all there is to be said about S, we have captured its semantics completely.

Before turning to next examples, I once more return to Euclid's algorithm

```
if A > 0 and B > 0:
   a:= A; b:= B;
   do a > b: a:= a - b,
      b > a: b:= b - a
   od
   print(a) fi
```

(By supplying "A > 0 and B > 0" as a guard for our previous algorithm we have made the condition imposed on the arguments explicit: if they don't satisfy it, the program as a whole will be aborted.) When we did derive the program we have already used the invariance of

P: GCD(A, B) = GCD(a, b) and a > 0 and b > 0 .

The loop theorem just mentioned tells us that upon termination we know also

$$\underline{\text{non}}\ (a > b\ \underline{\text{or}}\ b > a)$$

from which we conclude $a = b$, with the consequence that "print(a)" indeed produces the desired answer.

Does the loop terminate? Yes, for each execution of a guarded command from the list decreases $a + b$ by at least 1 and therefore, in view of the invariance of P, this can happen only a finite number of times. We see here that "termination" --i.e. all the guards false-- is here our goal; when synchronizing cyclic processes such as in an operating system, the situation "all the guards false" is called "a deadly embrace" and there one aims at avoiding this situation, because the show must go on! I hope to have shown the close connection between termination of sequential processes and deadlock in parallel programming.

Let us now try another problem. For $N \geq 0$, we are requested to assign to an integer variable "a" such a value that the relation

$$R: \qquad a^2 \leq N\ \underline{\text{and}}\ (a+1)^2 > N$$

is established. With "$a^2 \leq N$" as our P to be kept invariant during the loop, the program

$$a := 0;\ \underline{\text{do}}\ (a+1)^2 \leq N:\ a := a + 1\ \underline{\text{od}}$$

follows directly. We arrive at a more interesting program when we introduce a local variable (as a means for weakening R) and choose as our relation

$$P: \qquad a^2 \leq N\ \underline{\text{and}}\ b^2 > N$$

Clearly, $(P\ \underline{\text{and}}\ a+1 = b) \Rightarrow R$, and therefore our first sketch of the program can be (the first line is only one of the many ways in which P can be established easily)

```
a:= 0; b:= N + 1;
do a+1 ≠ b: "bring a and b closer together
             under invariance of P" od .
```

Let "d" be the amount, by which the difference $b - a$ is going to be decreased. Without loss of generality, we can assume that in the operation "bring a and b closer together under invariance of P" only one of the two variables will be changed each time, because the other one can be adjusted in the (or a) next execution of the repeatable statement.

Under these life-simplifying restrictions the repeatable statement will be somthing of the structure

```
d:= ......;
if ......: a:= a + d,
   ......: b:= b - d fi
```

Let us first find out the guards. The repeatable statement as a whole has to maintain the invariance of P. Let us derive in the usual manner the corresponding weakest preconditions:

wp("a:= a + d", P) = $((a + d)^2 \leq N$ and $b^2 > N)$

wp("b:= b - d", P) = $(a^2 \leq N$ and $(b - d)^2 > N)$.

Now our theorems about the if...fi construct tell us, that we can drop the terms implied by P and our repeatable statement becomes

```
d:= .......;
if (a + d)^2 ≤ N: a:= a + d,
   (b - d)^2 > N: b:= b - d fi .
```

This, however, would lead to abortion, if both guards were false. That must be excluded, so the negation of the one has to imply the truth of the other: e.g. $(a + d)^2 > N$ (the negation of the first guard) must imply $(b - d)^2 > N$. This implication is ensured if $(a + d) \leq (b - d)$, i.e. $2d < b-a$. In order to assure termination, d should be positive, but any value of d satisfying $0 < d \leq (b - a)$ div 2 will do. The larger the value of d, the faster our program, and my final suggestion for this program is therefore

```
a:= 0; b:= N + 1;
do a + 1 ≠ b: d:= (b - a) div 2;
                  if (a + d)^2 ≤ N: a:= a + d,
                     (b - d)^2 > N: b:= b - d fi
od
```

(Remark. If $2d < b - a$, then both guards may be true and it does not matter which of the two guarded commands is executed. In this example they could even be executed both, but that is not typical.) I think this example a beauty. The assignment to "d" is to ensure that the partial operator if ... fi is not invoked outside its domain, but, working backwards, we can derive what obligation this implies. (In Canterbury, last September, I have shown in essence the derivation of the same algorithm, but that was a clumsy affair, compared with the above!)

(26th November 1973). Since I wrote the previous pages of this report last week, I saw that Don Knuth pointed out in a letter to Tony Hoare, that this use of the comma violates all rules of interpunction: one should not use the comma as major separator between pieces of text that internally may contain the semicolon as minor separator. I agree and in the following text --this whole report is an experiment in notation!-- I shall use --inspired by the vertical bar "|" of Peter Naur's representation of BNF-- a fat vertical bar "[]" instead, i.e.

< guarded command list > ::= <guarded command > { [] < guarded command > } .

The next example I am going to code is known as "The Problem of the Dutch National Flag". In frontof a row of buckets, numbered from 1 through N, there is a minicomputer. Each bucket contains one pebble and each pebble is either red, or white of blue. The minicomputer can permute the pebbles because it has two controllable mechanical hands, controlled by the instruction "swap(i, j)" ($1 \leq i, j \leq N$): if $i = j$, this is the empty command, if $i \neq j$, the pebbles in buckets nr.i and nr.j respectively are interchanged. Also the machine has a movable eye used to compute the function "look(i)" ($1 \leq i \leq N$) of type colour: its value is the colour of the pebble in bucket nr.i (currently in bucket nr.i, I mean). The minicomputer must be programmed in such a way that it will rearrnge the pebbles in the order of the Dutch National Flag, i.e. red, white, blue. There are, however, three constraints.
1) We know nothing about the numbers of red, white or blue pebbles, the program must even work in the case of missing colours
2) It is a minicomputer with such a small store that the program may not make use of internal arrays
3) It is assumed that the evaluation of the function "look" is so time-consuming that we require that each pebble is "looked at" at most once.

The argument is the following. On account of the last requirement we have somewhere half way the computational process four kinds of pebbles: established red, established white, established blue and uninspected. We have to keep track of which is what, we cannot use an internal array for that purpose, we therefore use also the row of buckets and there contents as a memory element. We can represent then the information with three internal pointers, r, w and b according to the following convention:

for $1 \leq k < r$: the pebble in bucket nr.k is established red,
for $r \leq k \leq w$: the pebble in bucket nr.k is uninspected
for $w < k \leq b$: the pebble in bucket nr.k is established white
for $b < k \leq N$: the pebble in bucket nr.k is established blue.

Once we have chosen this general intermediate state, our problem is nearly solved, because both initial state (all pebbles uninspected) and final state (no pebbles uninspected and the remaining ones sorted out) are particular cases of the above described general state. So that one is established and under its invariance the number of uninspected pebbles is decreased one at a time. Inspection of the pebble in bucket nr. w gives on the average less swaps to perform than inspection of the pebble in bucket nr.r and we arrive ultimately at the following program

```
begin int var r, w, b; colour var v;
      r:= 1; w:= N; b:= N;
      do r ≤ w: v:= look(w);
                if v = red: swap(r, w); inc(r) []
                   v = white: dec(w) []
                   v = blue: swap(b, w); dec(b, w)
                fi
      od
end
```

and I prefer the last part over what I used to write down

```
                if v = red then swap(r, w); inc(r)
                   else if v = blue then swap(b, w); dec(b) fi;
                        dec(w)
                fi
```

I know that I am now talking about minute details --some of my readers might already wonder why I bother!-- but let us analyse the difference as completely as possible, so that we might learn from it. We have seen that decrementing "w" is part of the reaction in cases white and blue, so we decide to group them together in our first binary cut,where we ask for "v = red". (I have shown this example in its old version for many audiences and I remember my "justification" more or less going as follows "What the repeatable statement has to do is to decrease the difference w - r, either

by increasing r or by decreasing w , etc". What a waste of words!) In passing we note, that this way of partitioning becomes definitely unattractive, if for some reasons, we may expect a minority of red pebbles, and that is suddenly a quite different sort of consideration! If the pebble is not red, we discover that waht has to be done in the case of a white pebble is really a subset of what has to be done with the blue pebble and as no programmer would think of writing

```
if v = white then else swap(b, w); dec(b) fi; dec(w)
```

(you may omit the else but not the then!) we "discover" that we must ask explicitly for a blue pebble. If there happens, quite erroneously, to be a grey pebble as well, it is unconsciously treated as a white one. Then we learn by sad experience, that this is not too good from the point of robustness, so we introduce, as element of the well-trained programmer's competence, the notion of "defensive programming".... It just means, that educating him with the existence of "else" as the default case, he is always invited to choose, in a curious mixture of static and dynamic efficiency considerations, which of the cases he is going to treat as the default.... If we assume the availability of a mill with some capacity for concurrency and if the tests can be done concurrently, the whole argument disappears completely. As long as that is not the case, let us at least create a mental platform where we can ignore such details!

If a number of guarded commands from a list have a common tail, we might wish to save writing; we can do so by introducing an abbreviation, which syntactically can be presented as an alternative for the guarding head

< guarding head > ::= [< guarded command list >]: < statement >

allowing us to write

```
if v = red: swap(r, w); inc(r) []
  [v = white: []
   v = blue: swap(b, w); dec(b)]: dec(w)
fi
```

but this is now presented not so much as something that has to do with "sequencing control" -decisions and the like- it is just a short-hand notation, an abreviation. (As the different elements of the guarded command list are semantically unordered, we are absolutely free to order them

in the case of common tails as we see fit: it is a merging tree.)

* * *

The original reason to undertake these experiments was the following observation. Many a program shows the following phenomenon

```
while non null(x1) do
        begin temp:= tail(x1);.....
```

(this is taken from an example from R.M.Burstall), where the function "the tail of a list",i.e. what remains after removal of its first element, is undefined if its argument happens to be an empty list. It is a clear example of a partial function. In a program using such partial functions or operators, it turns out that much of the sequencing control in the upper level, using such functions or operators, is no more than ensuring that the partial operators or functions will not be invoked outside there domain. Now this is kind of silly: it implies that the upper level program has the duty to reflect in its explicitly stated sequencing commands what measures are to be taken in order to prevent invcation outside the domain, while anybody with any experience at all will know that for robustness sake the implementation of a thing like "tail" will start.... by checking that the upper level program satisfies that rule, that the upper level programmer has indeed met his obligations! Since I saw this, it has been hurting me, not so much for reasons of computational (in)efficiency, but perhaps still more for logical reasons: to present from below a partial function or operator does not seem to be a nice interface. I have been looking for a long time, what we should have instead, and the idea of the guarded command seems to come closer to it than anything I have seen before. More precisely, from below we offer so-called "guarded primitives" and syntactically they can be used as guarding heads, i.e. we have the alternative form (I give the old one from page 2 as well)

< guarding head > ::= < boolean expression > : < statement > |
< guarded primitive > .

Let me show, how these can be used; I use an example that I owe to W.H.J.Feijen. Let V be a set and let e be a variable of the type of the elements of V.

We consider two guarded primitives

sel(V, e): the guard of this primitive is the initial value of "non empty(V);
the corresponding action is to select any element from V, to remove it from V and to assign it to the variable e.

sub(V, e): the guard of this primitive is the initial value of "e in V";
the corresponding action is to remove this element from V.

It is now requested to make a program establishing whether the set BO is contained in the set AO. We introduce two variable sets, A and B and maintain the invariance of the following relation

(AO contains BO) = (A contains B)

(i.e. both sides are either true or false). Now we start massaging the sets A and B, but under invariance of the above relation. When A is empty, the question (A contains B) is easily answered: it is true if and only if B is empty as well. So after initialization we try to"empty" A, which we can do with the guarded primitive "sel". In order to maintain the invariance any element removed from A has to be removed from B as well, if B contains it. And we are led to the following program

```
A:= AO; B:= BO;
do sel(A, e); do sub(B, e) od od
answer:= empty(B)
```

(Remark: We have not exploited the fact that A and B are sets which, by definition, do not contain different elements. The "do sub(B, e) od" would old-fashionedly have been coded as if e in B then sub(B, e) fi: this construct now appears as a special case of a loop, viz. where the repeatable statement never needs to be executed more than once. In our version the program will also work for "collections of values": it will establish whether any value occurring at leat once in B, also occurs at least once in A.)

We are not finished with this example yet: the only possible change of B is that it loses an element: we are not interested in the final value of B, we only want to know of the final value of B is empty or not: as a result, if B becomes empty halfway, then we can stop te repetition, we can "extrapolate" the final value of B. And this suggests the following program

```
A:= A0; B:= B0;
do non empty(B) and sel(A, e); do sub(B, e) od od;
answer:= empty(B)
```

where we have used yet another form of guarding head, viz.
< guarding head > ::= < boolean expression > and < guarded primitive > .

This might seem to be at first sight a little piec of ad-hoccery, it is not. In order that something be evoked, in general we have two conditions: it must be possible to do it --and the guarded primitive presents that as a condition from below-- but you must also be interested in doing it, and that is clearly somthing that must be specified by the upper level programmer! Once you have seen this separation of reasons why to do something, classical programs, in which these different conditions are often lumped together in a single boolean expression, can strike you as unnecessarily obscure.

I shall give a final program, doing a merge sort, merging three sets A, B, and C into a fourth one; initially D is empty, finally A, B and C are empty. I assume a guarded command "selmin", that I have written in the form of a parallel assignment when used. Its guard is non-emptiness of the sets mentioned at the right hand side; its action is to select from eacht of the sets the minimum value and to assign this vector of values to the vector of variables at the left hand side. The sets are not changed. The operator "move(e, V, W)" transfers -if possible- an element with value e from V to W; if there is no such element, it is the empty statement.

```
do (a, b, c):= selmin(A, B, C); d:= min(a, b, c);
   if a = d: move(a, A, D) []
      b = d: move(b, B, D) []
      c = d: move(c, C, D) fi
od {comment: now at least one of the three input streams is empty};
if empty(A): do (b, c):= selmin(B, C); d:= min(b, c);
                  if b = d: move(b, B, D) [] c = d: move(c, C, D) fi
             od []
```

```
        empty(B): do (a, c):= selmin(A, C); d:= min(a, c);
                     if a = d: move(a, A, D) [] c = d: move(c, C, D) fi
                  od []
        empty(C): do (a, b):= selmin(A, B); d:= min(a, b);
                     if a = d: move(a, A, D) [] b = d: move(b, B, D) fi
                  od
     fi {now at least two of the three input streams are empty};
     if empty(A) and empty(B): do (c):= selmin(C); move(c, C, D) od []
        empty(B) and empty(C): do (a):= selmin(A); move(a, A, D) od []
        empty(C) and empty(A): do (b):= selmin(B); move(b, B, D) od
     fi {now all the three input streams are empty}
```

It is in this example quite clearly that the guarded primitive controls repetition, while --some sort of-- negation of the guard is used for selection: they express our interest.

Concluding remarks.

I regard what I have sketched as an aspect of a programming language, but it could be "an abstract programming language", i.e. a programming language for which we don't have a processor that, without further human intervention, will process our programs with acceptable efficiency. I do not care about that too much, it is a very one-sided view to regard programs as "things to execute automatically": they are also things to design, to enjoy, to embellish, to talk about and to use as a means of communication, either with yourself or with someone else. About algorithms. It is primarily with the latter aspects in mind, that the above has been written. Very tentatively written, even.

26th November 1973

BURROUGHS
Plataanstraat 5
NUENEN -4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Research Fellow