STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

REKENAFDELING

MR 46

An attempt to unify the constituent concepts of serial program execution.

by

Edsger W. Dijkstra

Paper to be presented at the Symposium
on Symbolic Languages in Data Processing. Rome March, 1962

Januari 1962

A machine defines (by its very structure) a language, viz. its input language; conversely, the semantic definition of a language specifies a machine that understands it. In other words: machine and language are two faces of one and the same coin. I am going to describe such a coin. I leave it entirely to you to decide which of these two aspects of the subject matter of my talk you think the most important as it is rather ridiculous in both aspects. The language I am going to sketch is prohibitively difficult for a human user and the machine I am going to describe is of a perverse inefficiency.

Therefore, if my mental construction, nevertheless, has a right to exist it should derive this from other qualities. My machine derives this, to my taste and judgement at least, from its extreme simplicity and elegance, from the uniform way in which it performs its (at a first glance) rather different operations; the justification for my language are its clearness and the unusually high degree of unambiguity, derived from a strict sequential interpretation and an explicit indication in the program to perform operations, which are usually implicitly understood (and therefore apt to misunderstanding). If one wishes to do so one may regard my machine and my language as being conceived for the purpose of clarification.

Before I really start with my description I should like to warn you of two intentionalomissions. The system I am going to present is the result of a careful choice between a great number of "neighbouring possibilities". I shall not give my motivations for these choices, I shall even leave the consciously rejected alternatives unmentioned. In other words, I refrain from introducing my system at least in some respects as, say, a "local optimum". As this diminishes the convincing power of my presentation, I personally regret this omission. I have to omit these motivations, however, for the sake of brevity.

The other question I shall not touch is the question of how to implement this system with the aid of a conventional machine. One might even raise the question - as I did myself to check that I was not thinking nonsense - whether it can be implemented at all, no matter how crudely. You have to take my word for it that it can be done. I have worked out a method of implementation to a degree that could convince, I think, the most suspicious auditor of the possibility. But it is my intention not to show you the particulars of this implementation, because I had to incorporate too many arbitrary decisions in it which, when mentioned, would only divert the attention from the essentials. In particular, the question of storage allocation will remain untouched.

My machine operates on (and under control of) units of information which I call "words". Without loss of generality I can restrict myself to a finite number of different words, each represented by the same number of bits.

The machine distinguishes between different kinds of words, say numbers, operators, variables and separators. For the time being we shall confine our attention to the first two of these, "number words" and "operator words".

A normal arithmetical operation, say the addition or the multiplication of two numbers, has two number words as input and one word, also representing a number, as output. The rules according to which a numerical value should be attached to (i.e. derived from the bits of) a number word are embodied in the workings of the arithmetic unit, which has the usual property that these same rules apply to both input and output: the output of the arithmetic unit can be fed into it again at some later stage of the process. As we assume that the properties of the arithmetic unit are constant in time,we may say that the number words have "a fixed meaning". As the fixed interpretation of number words is coupled to the constant properties of the arithmetic unit it is not so surprising that we shall denote the basic arithmetic operations by operator words (" + "," - "," * "," / ", etc.) the meaning of which can also be regarded as fixed.

The machine works under control of a program which primarily consists of a string of words. For the time being I shall confine myself to pieces of program prescribing the evaluation of arithmetic expressions.

Let us consider the expression that would normally be written down as

5 + 39 / (7 + 2 * 3) - 6 ;

in the usual postfix notation (also known under the name "Reversed Polish Notation") this would give rise to the following sequence of numbers and operators (successive items in this sequence for the sake of representation on paper being separated by spaces)

5 39 7 2 3 * + / + 6 - .

The well known mechanism especially devised for the sequential evaluation of such an expression is what I prefer to call a "stack". (This device has been invented and generalized independently by so many people that it is known now under a great variety of names, such as "push down list", "nesting store", "cellar", "last-in-first-out-memory" etc.) If we regard the above sequence of numbers and operators as the string of words representing a piece of program, the machine reads this string word by word from left to right. If it reads a number word, this number (i.e. a copy of this number word) is added to the top of the stack, if it reads an operator word the operation in question is performed at the top of the stack. In illustration I give on successive lines the successive pictures of the top of the stack where the top is at the right hand side of the line.

```
.....  5
.....  5  39
.....  5  39  7
.....  5  39  7   2
.....  5  39  7   2  3
.....  5  39  7   6
.....  5  39  13
.....  5  3
.....  8
.....  8  6
.....  2
```

and the net result of the execution of this little piece of program is that the value of this expression has been added to the stack.

As clearly shown in the above example the machine starts by copying the program text word by word onto the top of the stack. Sooner or later this has to be interrupted, otherwise our machine would just be a copying machine. In the above system the process of copying is interrupted by the occurrence of an arbitrary operator in the program text. The function of an operator, therefore, is a double one: firstly it indicates that the copying has to be interrupted for a while, because now an operation has to be performed, secondly it specifies this operation. I propose to separate these two completely different functions: from now on arithmetic operators are primarily treated in exactly the same way as numbers are treated, i.e. the operator word is copied into the stack as well. Everytime the process of copying has to be interrupted I shall indicate this in the program explicitly by the insertion of a special word, introduced now and represented by "E" (from "Evaluate"). My machine now takes the following form. It reads the program text word by word, from left to right. By "reading" is meant the following: if the word read is unequal to "E" a copy of it is added to the stack, if the word read is equal to "E", it is not copied but, instead, the operation takes place as specified (primarily) by the top word of the stack.

According to these rules the program prescribing the evaluation of the expression of our previous example will now consist of the following string of words:

5 39 7 2 3 * E + E / E + E 6 - E

and under control of this piece of program text (i.e. when this string of words is "read by the machine") the top of the stack will be in succession as shown in the following lines:

```
.....  5
.....  5  39
.....  5  39  7
.....  5  39  7   2
.....  5  39  7   2  3
.....  5  39  7   2  3  *
.....  5  39  7   6
.....  5  39  7   6  +
.....  5  39  13
.....  5  39  13  /
.....  5  3
.....  5  3   +
.....  8
.....  8  6
.....  8  6   -
.....  2
```

As said above the machine performs the operation specified by the top word of the stack when it reads the word "E" in the program text. We shall restrict ourselves to such programs that at such a moment the top word of the stack is indeed an operator word (and not, for instance, a number word). Furthermore we shall restrict ourselves to the case that the immediately underlying stack words are in accordance with any requirements that the execution of the operator at the top may set. (For instance, in the case of the binary arithmetic operations illustrated above the two immediately underlying words must be numbers.)

In other words: if an operand of an arithmetic operation happens to be an expression we substitute for this expression its numerical value before the operation is called into action, thus appealing to the fact that, primarily, the arithmetic operations are defined only when supplied with numerical operands

We regard the replacement of a (sub)expression by its numerical value as a "substitution", and we indicate explicitly when these substitutions have to be performed, although, linguistically speaking, this is rather redundant: "3 + 4" will always be equal to "7", no matter when we perform this addition.

This situation, however, changes radically as soon as variables - in contrast to constant numbers - are taken into account. (In the following we shall denote variables with small letters, reserving capital letters for "special words", such as "E" and others, to be introduced below.) Let us assume that we have to compute the value of the expression

" x + 4 "

at a moment that the value of the variable x equals 3. This means that in the above expression we must substitute for "x" its numerical value at that moment; only after having done so we can perform the arithmetic substitution (" 3 + 4" being replaced by " 7 "). Given something dependent on x (viz. the expression " x + 4 ") we create a result (viz."7") which, thanks to the fact that we have substituted for x its present value, is made <u>in</u>dependent of the future history of x. We have fixed an "instantaneous picture" of the variable x. Obviously I insist upon indicating explicitly <u>when</u> this instantaneous picture of the variable x (which is varying in time!) has to be taken.

Now we are going to harvest the first fruits of our labour for the mechenism for this explicit indication is already introduced. The piece of program prescribing the evaluation of the expression

" x + 4 "

now takes the following form:

" x E 4 + E "

and under the above assumption the successive pictures of the stack are

```
.....  x
.....  3
.....  3  4
.....  3  4  +
.....  7
```

Our machine invites us to describe the fact that "the value of the variable x equals 3" in slightly other wordings, viz. that the state of the process is such that reading the word "E" at a moment that the top word of the stack is "x" results in the replacement of this top word by the number word "3". The variable on the top of the stack is thus regarded as a variable operator which, upon evaluation, is replaced by something dependent on the state of the process at that moment; in this case it is an operator the execution of which sets no special requirements on the immediately underlying stack words. (The similarity between operators and variables will be further stressed by our next example.)

All words read in the text are added to the stack except the word "E" which causes the machine to perform a substitution. For reason to be explained below we should like to have also the possibility of adding the word "E" to the stack. The framework for this extension, however, is already present.We introduce a special operator, denoted by the word "P" (from "Postponement"), which effects upon evaluation a fixed substitution, viz. its replacement by the word "E". We shall illustrate the use of the operator "P" in the next example.

In this example we have three variables, named "x", "y" and "plinus". Suppose the state of the process to be such that reading "plinus" "E" generates the word "+" on top of the stack. When reading the text:

" x P E y P E plinus E P E "

the top of the stack will show in succession

```
.....  x
.....  x  P
.....  x  E
.....  x  E  y
.....  x  E  y  P
.....  x  E  y  E
.....  x  E  y  E  plinus
.....  x  E  y  E  +
.....  x  E  y  E  +  P
.....  x  E  y  E  +  E
```

and the top of the stack thus contains the string of words which, when read as a piece of program, would effect the evaluation of the expression "x + y". If the value of the variable "plinus" had been "-" we would have generated (the string of words corresponding to) the expression "x - y".

What we have done amounts to a partial evaluation of the expression "x plinus y", the result again being an expression. In our previous examples the final addition to the stack always consisted of a single number. But a number is a trivial example of an expression and generating not only numbers but also more general expressions as intermediate results is therefore an obvious extension of the normal practice.

Up till now we have described the generation of words on top of the stack but not what we are going to do with these words. Furthermore we have assumed that with respect to a given variable the process could be in such a state that evaluation of this variable would give rise to a previously defined substitution, but how this definition should take place is not mentioned in the above. These two gaps in our picture will both be filled by the introduction of the assignment operators.

For the assignment of a single word value, as in " x := 3" we could write in our program

" 3 x := E "

resulting in the stack pictures:

```
.....  3
.....  3  x
.....  3  x  :=
.....
```

Upon evaluation of the assignment operator ":=" the machine investigates the immediately underlying word. This must be the variable to which an assignment has to take place; the next underlying word is assigned to this variable (a process, about which more below) and the three words on top of the stack (which have now been processed) are removed from the stack. Until further notice -i.e. a new assignment to the variable "x"- the evaluation of this variable will result in the replacement of the top word of the stack by the word "3".

But for the interchanging of left and right hand side this is closely analogous to the assignment statement as known in ALGOL 60. But we need more than that for, in general, the assigned value will not consist of a single word, but of a string of words and we must therefore have a means of indicating how deep in the stack the assigned value extends. The simplest way to do this is to insert in the stack a marker, say the special word "T" (from "Terminal") at the bottom side of the assigned value. Furthermore we introduce another assignment operator ":-" (called the "string assignment" in contrast to the "word assignment" introduced in the previous paragraph). Upon evaluation of this operator the machine investigates the top of the stack in the downward direction. The first word (immediately under the operator ":-") must be the variable to which a value has to be assigned. Thereafter the machine continues its word by word investigation in the downward direction until it meets the special marker "T": the words passed in this way form together the string that acts as the assigned value.

The simplest way to add a "T" to the stack would be just to insert the word "T" in the proper place in the program under control of which the stack is being filled. This arrangement, however, will not do; for reasons to be explained later we need the possibility of generating a "T" on top of the stack under control of a program that itself does not contain this word. We can do this with the same trick that enabled us to generate an "E" on top of the stack. We introduce a new operator, denoted by the word "S" (say from "Seperator" or because it precedes the "T" in the alphabet) which upon evaluation is replaced by the word "T" and we make it a rule that this will be the only way in which words "T" are added to the stack.

Using all this we have an alternative way to write the assignment statement " x := 3", viz.

" S E 3 x :- E "

giving in the top of the stack in succession:

```
.....  S
.....  T
.....  T  3
.....  T  3  x
.....  T  3  x  :-
.....
```

The net effect of this is equivalent to the previous form using the word assignment ":=".

Let us use the more powerful assignment in an example which is an extension of one of our earlier ones, viz. the one describing the partial evaluation of the expression " x plinus y ". The result of this partial evaluation was an expression depending on the variables "x" and "y";suppose that we want to call this expression "z". For this purpose we write in the program:

" S E x P E y P E plinus E P E z :- E " .

When the last "E" of this string is going to be read the top of the stack will be as follows (under the same assumption with respect to the value of "plinus"):

..... T x E y E + E z :-

and after the execution of this assignment the above words will have been removed from the stack, the word "T" inclusive. Until further notice the evaluation of the variable "z" will imply the execution (the "reading") of the string assigned to it. Upon evaluation of the variable "z" the machine therefore must have access to the first word of this string; when it starts reading this string, however, it must detect the last word of this string. We propose that the assignment operator sees to this by adding again an end marker and for this purpose we can use the very same word "T". Upon evaluation of the variable "z" the string assigned to it will be read as a piece of program, from left to right, until the end marker "T" is met. The new situation resulting from the last assignment can conveniently be represented by:

" z $\rightarrow$ x E y E + E T ".

In exactly the same way our previous assignments

" 3 x := E " or " S E 3 x :- E "

will both give rise to the situation, represented by

" x $\rightarrow$ 3 T ".

One of the most illuminating aspects of this arrangement is that the usual distinction between "numbers" and "instructions" has completely vanished. The value of a variable is defined as a piece of program, evaluation of this variable implies the execution of this piece of program.

Furthermore we should like to draw attention to a certain form of duality between the assignment on the one hand and reading a text on the other. When the machine reads a piece of program text, the top of the stack is filled under control of this program text. In the assignment "readable text" is created under control of the contents of the stack. The duality can also be illustrated by taking into consideration the accessibility requirements. The words in the stack need only be accessible in the direction from top to bottom. If an assignment statement converts the top of the stack into readable text, however, the consecutive words thereby become accessible in the other direction.

Finally, the stack is reserved for "anonymous intermediate results", whereas readable text -in principle, at least- is always "named", for we create it by assigning it to a variable.

The attentive reader will have noticed that, along with the representation of the value of a variable, we have silently introduced two more complications in our machine.

The first one, the occurrence of the word "T" in program text and the machine's "immediate reaction" to it is a relatively simple one. As we have described the organization, the word "T", when read in the text, is <u>not</u> copied on top of the stack! Instead, it causes the machine to go on reading at the first word following in the string after the "E" that caused this evaluation of the variable in question. In other words, it acts as a "Return" at the end of a closed subroutine.

But the evaluation of a variable may call for the evaluation of other variables (even for the evaluation of itself): the pragmatic definition of the evaluation of a variable is basically a recursive one and the mechanism one needs to follow a recursive definition is .... another stack! I call this second stack "the stack of activations" in contrast to the first which I call "the anonymous stack". One of the functions of the stack of activations is to control the reading process. When the evaluation of a variable starts the stack of activations expands, when the corresponding word "T" is read, it shrinks to its previous size. (In the usual terminology of machine structure: the stack of activations contains a stack of "order counter values", its top element being, by definition, "the present order counter"; in this same terminology its older elements act as a stack containing the "return addresses".)

Note. We could try to merge our two stacks into one. This merging would present itself in a completely natural fashion if the two should expand and shrink "in phase" with one another. In general, however, this is not the case and trying to merge the two stacks into a single one would give a highly unnatural construction.

We shall use the stack of activations for yet another purpose, to satisfy a very fundamental need, viz. the creation of new variables. In the above I have used special words ("x", "y", "plinus" etc.) to denote variables and I have carefully avoided using the term "identifier". I have used the term "variable" in connection with a single, unique object, existing for some period of time and capable of taking on different values in succession. This concept of a variable is to be distinguished carefully from the "identifier" as used in ALGOL 60, because one and the same identifier may be used to point to a host of objects, to a great number of different variables.

First of all we meet the fact that one and the same identifier may play different roles thanks to the fact that it occurs in more than one declaration. A lexicographical rule then tells us which one of these declarations applies everywhere, where the identifier in question may be used. This form of multiple use of one and the same identifier could be removed by a simple process of renaming.

But there is a much more subtle case of "multiple use of one and the same identifier", viz. as soon as a certain block occurs in one or more nested activations (as in the case of a recursive procedure). In other words: one and the same identifier then refers sometimes to this variable, sometimes to another.

In actual fact: the identifier stands for a variable and in order to indicate clearly for which variable it stands I intend to denote explicitly the moment when a variable has to be substituted for an identifier.

For the sake of convenience -to be more exact: convenience for the machine and not for the hypothetical user- I intend to use the same identifiers for the local variables of every activation. (What I call "an activation" is closely analogous to a block or a procedure body, as known in ALGOL 60.) I use for this purpose the special identifier words "L0","L1", "L2", etc..

When the machine starts the evaluation of a variable, the stack of activations increases by one item. At the start this item also contains a note that up till now no local variables have been introduced in this activation.

If the machine reads the word "E" at a moment that the top of the anonymous stack contains one of the identifier words (say "L2") then it investigates the top item of the stack of activations. If it is the first time that this identifier has to be evaluated in the present activation the machine creates a new variable for it (and may give this variable an empty value) and makes in the youngest item of the stack of activations a note to this effect. Then it replaces the top word of the anonymous stack by the variable just created for it. At a next evaluation of the same identifier at a moment that the same activation is still (or again) the present one, the machine finds in the top item of the stack of activations the note left there at the first evaluation of this identifier and the top word of the stack is replaced by the very same variable.

Now we can show a more complicated example. Let the values of the variables "x", "y" and "complus" be as represented by:

```
    " x →  10  23  T  "
    " y →   5  -2  T  "
" complus → L0  E  :=  E
            L1  E  :=  E
            L2  E  :=  E
            L1  E  E   +  E
            L2  E  E   L0  E  E  +  E
            T  ".
```

If we now read the text

```
" S E x E y E complus E z :- E "
```

the net effect will be that we can represent the new value of "z" by:

```
" z → 15 21 T "     ,
```

and what we have done can be interpreted as the addition of two complex numbers.

In ALGOL terminology: "complus" is a procedure with four numerical parameters, all called by value. The simple structure of the process allows the first of these to remain anonymous even in the procedure body. Furthermore, it is a kind of "type procedure", be it one that, syntactically speaking, takes the place of two primaries.

Let me end with a trivial example. Suppose that we want to write "plus" instead of "+". After the assignment

" S E + P E plus :- E " ,

which gives rise to the situation

" plus → + E T "

the expressions

" x E y E plus E "

and

" x E y E + E "

are completely equivalent. This example is included to show as clearly as possible the arbitrariness of our primitives.

Conclusion.

I am fully aware that the sketch is definitely incomplete. In particular conditional reaction and some equivalent of the go to statement should be incorporated if one wishes to make a system out of this. For the moment I leave these out and I do so for two reasons. Firstly for the sake of brevity and secondly because I have not decided yet: I know of several possible ways but none of them fully satisfies me.

With some versions of these facilities I have made slightly more elaborate programs. They showed me both the power and the weakness of my Language, its power being its flexibility and its unambiguity, its weakness being the fact that using it intelligently proved to be far beyond at least my powers.

If nevertheless I claim attention for this project Ido not do so only because it charms me and may charm others as well. This report is the condensation of my meditations after we had completed our implementation of ALGOL 60. This implementation was conceived at high speed and the main justification for the numerous decisions taken in those hectic months was the recognition that our conceived constructions would lead to our goal and would do the job, in some way or another. The Machine described in this report, however, represents an extreme of the continuous spectrum of possible implementations of an algorithmic language which (as is the case with ALGOL 60) caters for recursiveness. In this quality it has been very clarifying for me personally: it has helped me a great deal in the appreciation of the various (initially disconnected) tricks we have incorporated intuitively and it has clearly shown us a number of alternative solutions. Therefore the hope is justified that translator construction and machine design in the future will benefit from these considerations.

Furthermore, the Machine presented here is so ridiculously inefficient that every practical implementation of a practical algorithmic language in all probability can be regarded as an optimization of it, an optimization which is permissible thanks to certain restrictions in the language. It may be useful to compare a proposed language with my language; during the process of language construction it may be helpful in the timely detection of "expensive features". Whether such an expensive feature will be included or not is more or less a political question but quite apart from how such a question is answered it is nice to know what one is doing.

Finally the language described in this report (or a language devised along similar lines) may prove to be a suitable means for the formulization of the semantic definition of an algebraic language. The lack of such a rigorous semantic definition is one of the recognized shortcomings of the official "Report on the Algorithmic Language ALGOL 60" and having seen the tremendous amount of trouble caused by this defect, I most sincerely hope that this report will contribute to the effort to avoid this mistake the next time an algorithmic language is to be devised.

Acknowledgements.

A great number of people have contributed to this, consciously or not. Besides all my colleagues at the Computation Department of the Mathematical Centre, Amsterdam, I should like to mention Dr.M.V. Wilkes and Prof. J. McCarthy, who proved to be inspring listeners, and in particular Mr. M.Woodger: his judgement and his comments (I remember his lack of enthusiasm for my first trials in this direction now with gratitude) have been a great help for me.