STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

MR 79

Recursive Definition of Syntax and Semantics

A. van Wijngaarden

1966

MC

REPRINTED FROM

FORMAL LANGUAGE DESCRIPTION LANGUAGES FOR COMPUTER PROGRAMMING

Proceedings of the
IFIP Working Conference on
Formal Language Description Languages

Edited by
T. B. STEEL, Jr.

RECURSIVE DEFINITION OF SYNTAX AND SEMANTICS

A. VAN WIJNGAARDEN
Netherlands

1966

NORTH-HOLLAND PUBLISHING COMPANY - AMSTERDAM

RECURSIVE DEFINITION OF SYNTAX AND SEMANTICS

A. VAN WIJNGAARDEN
Netherlands

1. INTRODUCTION

In a former paper [334] a mechanism was described which interprets a text, called a program, and delivers another text, called the value of the text so far read. Knowledge of the working of this machine enables the writer of the program not only to describe the process he wants to describe, but also the language he is using. It acts, therefore, both as language and as metalanguage. In [334] most of the emphasis was laid on showing how a language of the ALGOL type (but much more general) could be defined in this way, using only few definitions. We shall now investigate in more detail some metalinguistic properties, without concerning ourselves with the quality of the language to be described.

The machine which interprets the text was considered to consist of two parts, a preprocessor and a processor. The preprocessor was not formalized and may vary from case to case. The processor was formalized to a high degree and does not vary.

2. THE PREPROCESSOR

Let us first turn our attention to the preprocessor. This rewrites the text into an equivalent one in a more restricted language. Indeed, a language may contain many pseudoconcepts, *viz.*, concepts expressible in other concepts of the language. It is therefore advantageous to split the definition of the language into two corresponding parts.

By way of example, we take ALGOL 60 and see which are pseudoconcepts in its case.

The first pseudoconcept we find is the comment, which has no semantic meaning at all. Hence, in any occurrence outside strings one may delete certain sequences of basic symbols completely.

Similarly, outside strings the sequence ") ⟨letter string⟩ : (" may be replaced by ",". Or again, outside strings the basic symbol " **array** ", if not preceded by "⟨local or own type⟩", may be replaced by " **real array** ". This might perhaps not seem a simplification, but it simplifies the description of the language.

Our next victim is the **for** statement, which can be rewritten in its defining sequence of statements.

Next comes the function designator and the corresponding type procedure. Replace the type procedure by a nontype procedure with one more formal parameter, and replace the assignment to the procedure identifier by an assignment to that formal parameter. Replace all primaries in expressions in an assignment statement by auxiliary variables; let the statement be preceded by a sequence of statements assigning the required values to these auxiliary variables, and let it be followed by one or more statements assigning the value of the left-part auxiliary variable to the actual left-part variable or variables.

This last precaution is required by the nature of formal parameters. Inside a procedure body a formal parameter is either passed on, or one requires its value, or one wants to assign to it or execute a **goto** statement leading to it. Replace the actual parameters in a procedure call, which also may contain expressions, by the identifiers of procedures with two parameters. These procedures (depending on the value of the second parameter) assign the value of the first one to the actual parameter (output), or the value of the actual parameter to the first one (input), or execute a **goto** statement leading to the actual parameter, etc. In the procedure body the formal parameters are then replaced by the corresponding calls.

Conditional expressions that are not actual parameters can be removed by text splitting, e.g., "a := **if** b **then** c **else** d" is replaced by "**if** b **then** a := c **else** a := d". If such conditional expressions occur in actual parameters, the corresponding formal parameters of which are called by name, this text-splitting procedure will not work, but this case has already been taken care of by the preceding measures.

By such an intricate but still lexicographical process, one not only eliminates the function designator but actually defines what it means. We might note that the ALGOL 60 Report [244] contradicts itself on this point.

When one has performed the mentioned reductions of the text, it will have a much simpler appearance. Procedure calls will no longer appear in expressions and actual parameters, and conditional expressions will no longer exist. This enables us to do away with switches, labels, and **goto** statements. The switch declaration is replaced by a corresponding procedure declaration containing **goto** statements, and a **goto** statement referring to a switch element is replaced by the corresponding procedure call. For instance, "**switch** S := $S1$, $S2$, $S3$" is replaced by "**procedure** $S(n)$; **value** n ; **integer** n ; **if** $n = 1$ **then goto** $S1$ **else if** $n = 2$ **then goto** $S2$ **else goto** $S3$", and correspondingly "**goto** $S[i]$" is replaced by "$S(i)$". That the switch list can be replaced by a statement in this way explains why the difficulty of function designators and conditional expressions in a switch declaration could be deliberately overlooked above.

Next remove all multiple labels, renaming references to the removed ones. This obviously reduces the number of labels, though the steps I am about to outline may not seem promising. Provide each procedure declaration with an extra formal parameter - specified **label** - and insert at the end of its body a **goto** statement leading to that formal parameter. Correspondingly, label the statement following a procedure statement, if not labeled already, and provide that label as the corresponding extra actual parameter. Also, label each statement following a **goto** statement, if not labeled already, and complete a conditional **goto** statement that is an **if** statement by " **else goto** *L*", where *L* stands for the label immediately following the statement. Label each block, if not labeled already, except the outermost one, and label the first statement of each block, if not labeled already. If a label that is not the first inside a block is not preceded by a **goto** statement, then insert a **goto** statement leading to that label. Enclose each sequence of statements between two successive labels in the brackets **begin end**, if not already enclosed in that way.

The structure of the program obtained by this process is remarkable. It consists, after the insertions corresponding to the performed procedure calls, of a sequence of elements, *viz.*, compound statements and blocks, containing no **goto** statements except at each end to link the element to another. It is now completely harmless to insert at the end of each block an unlabeled **goto** statement leading to the first statement of that block, since this statement will never be executed. So far, we have only increased the number of labels and **goto** statements. But now we can perform the following operations:

i) write before each label **procedure** ;
ii) replace the colon following it by a semicolon;
iii) strike each **goto**.

The program is then again syntactically correct, contains no labels or **goto** statements, and defines exactly the same sequence of operations as before.

This sketch may suffice to show the power of preprocessing. ALGOL 60, reduced in this way, is seen to contain only a few concepts, such as:

i) some arithmetical and Boolean operations;
ii) assignment;
iii) the procedure with or without parameters, call by value, and call by name;
iv) locality and "own" concept.

In [334] it is shown that, with some minor modifications of the language, the concepts under ii) and iii) can be identified and expressed in some simple rules of preprocessing and processing. The concepts under iv) need more care, but concepts such as those under i) are simply dealt with by the processor, as we shall show.

First, however, we want to be somewhat more specific about the precise role of the preprocessor than we were in [334]. For a language like ALGOL 60 in which a program is a fixed text, it suffices to separate the preprocessor and processor completely, so that the processor can process the preprocessed text without needing further assistance from the preprocessor. However, for the description of languages that enable the generation of pieces of program by the program itself, this does not hold. In this case the preprocessor must continuously stand by to preprocess new pieces of text that have been generated. If the preprocessing can be defined as idempotent, then any text, generated or not, can always be preprocessed before being processed. If this is not the case, the preprocessed text must be distinguished from unpreprocessed text. One might visualize the unpreprocessed text as written in black ink, whereas the preprocessor turns out text in red ink.

3. THE PROCESSOR

We now turn our attention to the processor, which by evaluating the preprocessed text produces its value *v*, a dynamically varying text. This text *v*, on the other hand, is recursively scanned by the processor for two reasons. Either the processor wants to determine the value of a piece of text, or it wants to ascertain whether a truth in *v* is applicable to the question it is concerned with. Apart from some loose remarks concerning locality, and so on, *v* consists of a sequence of truths, separated by commas. Some of them are of a syntactic nature such as, *a* **in** ⟨letter⟩, which states that *a* is one of the values that the metalinguistic variable ⟨letter⟩ may take.

Others are of a semantic nature. They contain the equality sign, = , and they may contain the metaoperator, **value** , which operates on the immediately following metaprimary. Any sequence of symbols, for that matter, can be turned into a metaprimary by enclosing it in the metabrackets { }. Examples are:

value ⟨name 1⟩ = ⟨name 2⟩,

10 - ⟨digit 2⟩ = 9 - **value** {⟨digit 2⟩ -1},

2 + 1 = 3

In the evaluation scan the processor is interested in the semantic truths, but in order to know whether one applies, it has to undertake an applicability scan. Both scans are performed backwards, i. e., the truths in *v* are examined one by one in order, starting with the last one contained in *v*. The applicability scan may assert that a truth is applicable. This means that the quantity, whose value must be determined, say, ⟨name 1⟩, or that value itself, is identical with the left-hand side of the truth after permissible substitutions. The evaluation scan then applies this fact by applying the same substitutions to the right-hand side. If the truth takes the form

value ⟨name 1⟩ = ⟨name 2⟩,

then the required value is simply ⟨name 2⟩, and the evaluation scan is ended. If the truth takes the form

⟨name 1⟩ = ⟨name 2⟩,

then the required value is **value** ⟨name 2⟩, and a new evaluation scan is started to find this value. Since ⟨name 2⟩ may itself contain the operator **value** , this may be a complex affair, evaluation necessarily starting from the inside and also from left to right.

In order to find out whether or not a substitution in accordance with the truths in *v* will make the left-hand side of a truth identical with an entity under consideration, the applicability scan applies a systematic parsing process to the left-hand side until it has success. Which parsing process is used is not relevant here, but it must be defined in order to guarantee unambiguous interpretation. If, for instance, the left-hand side of the truth contains p primaries and the entity under consideration g primaries, $g \geq p$, then these g primaries can be parsed into $g_1, g_2, \ldots g_p$ sequences of primaries, $g_1 + g_2 + \cdots g_p = g$. The sequence $g_1 g_2 \ldots g_p$ can be considered as a number in the base g. Then a simple parsing scheme is to investigate the parsings in increasing magnitude.

The applicability scan then compares each primary under consideration with the corresponding primary in the truth under consideration to see whether they are identical or, if the latter contains a metalinguistic variable, whether they can be made identical by permissible substitution that generaates another independent applicability scan. If all parsings have been tried without success, the next truth is investigated. If v is exhausted in this way, this result defines nonapplicability. The applicability scan, therefore, always yields its answer in a finite number of steps. The evaluation scan also yields a definite answer, since at the bottom of v we presume that we will find

value ⟨name 1⟩ = ⟨name 1⟩,

which always applies if nothing else has done so.

The descriptive power of our metalanguage can be increased considerably by assuming that the machine understands the logical operators ¬ (not) and → (implies). As an example, we define a row, say, as a sequence of letters, none of which are equal, by

{⟨letter 1⟩ **el** ⟨row 1⟩} → {⟨letter 1⟩ **el** ⟨row 1⟩ ⟨letter⟩},
⟨letter 1⟩ **el** ⟨row⟩ ⟨letter 1⟩,
⟨letter 1⟩ **el** ⟨letter 1⟩,
⟨row⟩ ⟨letter⟩ **in** ⟨row⟩,
{⟨letter 1⟩ **el** ⟨row 1⟩} → ¬{⟨row 1⟩ ⟨letter 1⟩ **in** ⟨row⟩},
⟨letter⟩ **in** ⟨row⟩,

where, in passing, an auxiliary operator **el** is defined.

As a more complicated example, we conclude by giving a partial description of decimal arithmetic, *viz.*, the addition and subtraction of two integers. Since in a completely formalized description of a language the sequences of letters chosen to represent metalinguistic variables may be chosen arbitrarily, we abbreviate "digit" to "di", "unsigned integer" to "ui", and so on, in order to save space. The definition is very slightly redundant to increase efficiency and cleaner output.

0 **in** ⟨di⟩, 1 **in** ⟨di⟩, 2 **in** ⟨di⟩, 3 **in** ⟨di⟩, 4 **in** ⟨di⟩,
5 **in** ⟨di⟩, 6 **in** ⟨di⟩, 7 **in** ⟨di⟩, 8 **in** ⟨di⟩, 9 **in** ⟨di⟩,

$$\begin{array}{l}
\langle\text{di}\rangle\ \textbf{in}\ \langle\text{ui}\rangle,\ \langle\text{ui}\rangle\,\langle\text{di}\rangle\ \textbf{in}\ \langle\text{ui}\rangle,\\
+\ \textbf{in}\ \langle\text{pm}\rangle,\ -\ \textbf{in}\ \langle\text{pm}\rangle,\ \langle\text{pm}\rangle\,\langle\text{ui}\rangle\ \textbf{in}\ \langle\text{in}\rangle,\ \langle\text{ui}\rangle\ \textbf{in}\ \langle\text{in}\rangle,\\
0\ \textbf{in}\ \langle\text{ze}\rangle,\ \langle\text{ze}\rangle\ 0\ \textbf{in}\ \langle\text{ze}\rangle,\ \langle\text{ze}\rangle\ \textbf{in}\ \langle\text{ui}\rangle,\\
+\ \langle\text{ui 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{ui 2}\rangle = \langle\text{ui 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{ui 2}\rangle,\\
-\ \langle\text{ui 1}\rangle + \langle\text{ui 2}\rangle = \langle\text{ui 2}\rangle - \langle\text{ui 1}\rangle,\\
-\ \langle\text{ui 1}\rangle - \langle\text{ui 2}\rangle = -\ \textbf{value}\ \{\langle\text{ui 1}\rangle + \langle\text{ui 2}\rangle\},\\
\quad\langle\text{ui 1}\rangle + -\ \langle\text{ui 2}\rangle = \langle\text{ui 1}\rangle - \langle\text{ui 2}\rangle,\\
\langle\text{ui 1}\rangle\,\langle\text{di 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{ui 2}\rangle\,\langle\text{di 2}\rangle = \textbf{value}\ \{\langle\text{ui 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{ui 2}\rangle\}\ 0\\
\qquad\qquad\qquad\qquad\qquad\qquad + \textbf{value}\ \{\langle\text{di 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{di 2}\rangle\},\\
\langle\text{ui 1}\rangle\,\langle\text{di 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{di 2}\rangle = \langle\text{ui 1}\rangle\ 0 + \textbf{value}\ \{\langle\text{di 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{di 2}\rangle\},\\
\langle\text{di 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{ui 2}\rangle\,\langle\text{di 2}\rangle = \langle\text{pm 1}\rangle\,\langle\text{ui 2}\rangle\ 0 + \textbf{value}\ \{\langle\text{di 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{di 2}\rangle\},\\
\langle\text{ui 1}\rangle\ 0 + \langle\text{di 2}\rangle = \langle\text{ui 1}\rangle\,\langle\text{di 2}\rangle,\\
\langle\text{di 1}\rangle + \langle\text{ui 2}\rangle\ 0 = \langle\text{ui 2}\rangle\,\langle\text{di 1}\rangle,\\
\langle\text{ui 1}\rangle\ 0 - \langle\text{di 2}\rangle = \textbf{value}\ \{\langle\text{ui 1}\rangle - 1\}\ 0 + \textbf{value}\ \{10 - \langle\text{di 2}\rangle\},\\
10 - \langle\text{di 2}\rangle = 9 - \textbf{value}\ \{\langle\text{di 2}\rangle - 1\},\\
\langle\text{di 1}\rangle\,\langle\text{pm 1}\rangle\,\langle\text{di 2}\rangle = \textbf{value}\ \{\langle\text{di 1}\rangle\,\langle\text{pm 1}\rangle\ 1\}\ \langle\text{pm 1}\rangle\ \textbf{value}\ \{\langle\text{di 2}\rangle - 1\},\\
\langle\text{in 1}\rangle\,\langle\text{pm}\rangle\,\langle\text{ze}\rangle = \langle\text{in 1}\rangle,\ \langle\text{ze}\rangle + \langle\text{ui 1}\rangle = \langle\text{ui 1}\rangle,\ \langle\text{ze}\rangle - \langle\text{ui 1}\rangle = -\ \langle\text{ui 1}\rangle,\\
\langle\text{ze}\rangle\,\langle\text{pm}\rangle\,\langle\text{ze}\rangle = 0,\\
0+1 = 1,\ 1+1 = 2,\ 2+1 = 3,\ 3+1 = 4,\ 4+1 = 5,\\
5+1 = 6,\ 6+1 = 7,\ 7+1 = 8,\ 8+1 = 9,\ 9+1 = 10,\\
\{\langle\text{di 1}\rangle + 1 = \langle\text{di 2}\rangle\} \rightarrow \{\langle\text{di 2}\rangle - 1 = \langle\text{di 1}\rangle\}.
\end{array}$$

DISCUSSION

GORN
You're analyzing a number of the concepts in ALGOL in terms of what you feel are more basic concepts, and you feel that there is a preprocessor that eliminates the less basic ones and replaces them by bigger expressions - perhaps, by more basic ones. Am I to understand that you feel it would actually be done this way in the program? All at once, before the processor gets to work?

VAN WIJNGAARDEN
Yes and no. You remember that we actually had two machines here. Here is a preprocessor; that is a processor. Here comes the original text written by a man who writes, let us suppose, in ALGOL 60, which is quite a language. If I look at the [ALGOL] Report, I say to myself, must I define *all* this by the processor - all these rules? This is far too much for me! So I say, let's first take all the nonessential things out of ALGOL. Now, this is a task for the preprocessor - to look

at this text and say, "I'll translate this text into reduced ALGOL and then define only reduced ALGOL". By the way, I have to define this preprocessor, and this preprocessor depends upon a specific language.

GORN
You're saying that even if it isn't efficient, the preprocessor could do all the elimination before the processor would have to work - without any loss of meaning.

VAN WIJNGAARDEN
Sure. You see, I do not change any identifier.

GORN
The first things you eliminate are things like comments, I noticed. The implication is that what follows the symbol **comment**, as far as the processor is concerned, has no meaning. Is that right?

VAN WIJNGAARDEN
If I read the ALGOL book, it says that elimination of this piece of text does not influence the computation in any way. If I took a procedure, for instance, that counts the number of basic symbols in the program, then it could not be affected by the elimination of this piece of text. What you want to mean by this, I don't know.

GORN
All right then, the implication is that raw data - especially what follows **comments** - is in itself meaningless. And I feel there is an important meaning that is very basic; the meaning of raw data, including what follows **comments**, is the demand that it be allocated to storage to be properly retrievable. For instance, what follows the comment? You might want to print it out at the end of a process - you don't just want to throw it away.

VAN WIJNGAARDEN
The preprocessor can do all this kind of stuff. But this has nothing to do with anything.

GORN
That's what bothers me. You have to decide where to start, even if it's the waste-basket, and this might have to go on while the process is going on. Now let me give you some more examples...

VAN WIJNGAARDEN
Excuse me, I do not agree with this thing. ALGOL 60 describes the computational processes, and it does not describe the process to be done with this process.

GORN
You brought up the question that an array declaration, for instance, is one of the difficult parts of ALGOL, and I say - for some people with some machines - one of the *most* difficult parts. The problem there is precisely this question of properly allocated storage.

VAN WIJNGAARDEN
Have you heard of the 5-10 procedure?

GORN
Well, that is also allocation of storage. Now, the next things you eliminate are things like goto's, switches, labels. You make a remark that multiple labels are superfluous and tend to be eliminated at this stage. Is this correct?

VAN WIJNGAARDEN
Yes.

GORN
This means that the processor will never see again (after the preprocessor has worked) the multiplicity of labels that you might have. Now suppose that I have a

switch declaration in which I use labels L1, L2, L3, and another one in which I use N1, N2, N3; and L1 happens to be the same as N1 but the others are not. Now, the processor does a certain amount of cycling among labels in the first group of three and cycling among labels in the second group of three, and if you eliminate the multiplicity of labels you have lost the cycling that you wanted in the processor.

VAN WIJNGAARDEN
No-no! If you see what a goto statement means, then there is nothing between the initialization of the goto statement and where it leads. It just says that its successor has been designated to be that point.

GORN
This has to be done by a modular counter. You've lost that.

VAN WIJNGAARDEN
What is that? A modular counter, what is that?

GORN
It counts: 1,2,3; 1,2,3; 1,2,3.

VAN WIJNGAARDEN
That's outside ALGOL. It's a metaconcept I don't understand.

GORN
In general I'm worried about the preprocessor doing all the elimination first because it may be an important part of the efficiency of the processor that the original program you wrote would indicate how you want to do it, and the preprocessor would lose that.

VAN WIJNGAARDEN
No-no-no! Well, at least, I did my best to make all these changes into labels go through exactly the same sequence of operations that was performed by the modified process. I did my best, at least, to do that.

GORN
What about translating dynamically?

VAN WIJNGAARDEN
In a language like ALGOL? Because the text is a fixed piece of information you cannot do this thing.

DIJKSTRA
I am somewhat baffled, I might say, in many ways. You describe an algorithm for a preprocessor that says that it does something. Can you prove its correctness? Because I can't see how it works under all circumstances.

VAN WIJNGAARDEN
I did my best. [Laughter] I cannot take this expression out; but on the call side I do not know, in general, whether an actual parameter is, at least, the preprocessor that looks at this step and doesn't understand the meaning of it - does not know whether it will be used as input or as output or as both. Be careful even in the case that it is an expression. There is no guarantee that in some other case it couldn't be used as an output. In the case of a procedure - in the case of, say, "$x > 0$" - then assign it; otherwise take its value. You do it in the inside of the body, of course. But then, of course, you don't know what's on the outside. Therefore, instead of the actual parameters, you give a set of procedure identifiers, which have two parameters. The second parameter tells how the thing should react, and the first one is the formal parameter. In case 1 assign the formal value to the actual; in case 2 assign the actual value to the formal; in case 3 transfer control to the actual. Inside, you know, you can follow the procedure and then you can tell which case actually applies. You may simply pass it on. If you really use it, you can tell which case applies and you can provide the proper second parameter. If it's the left-hand side of an assignment statement, for instance, you do it in the assign state. And then, that way, I get out all expressions for the actual parameter. OK?

DIJKSTRA
Yes. The next question consists, after the insertion corresponding to the procedure calls, of a sequence of statements.

VAN WIJNGAARDEN
If you have performed the procedure call, this is equivalent to insertion of a sequence of statements.

DIJKSTRA
I thought we were talking about preprocessors.

VAN WIJNGAARDEN
Now look! The preprocessed text will be such that, after the execution, the program will consist of these elements.

DIJKSTRA
I just don't understand. You have your text. You make another text that remains without procedure calls. Now, you say that somewhere or another you make the insertions; so that we do without procedure calls. Now we have only goto's.

VAN WIJNGAARDEN
What! What? What? You have only *statements* - sequences of *statements*. You have no goto's whatsoever! You have only sequences of statements, and these sequences of statements are *exactly* the same sequences of statements that would have been there in the other case.

HOARE
It seems to me your method of describing the processor is sufficiently powerful to give a complete, precise, and even elegant description of the preprocessor as well, which, of course, is an important part of the semantic definition of a language. If this is true, it becomes attractive to put the preprocessing rules into the bottom of the processor itself, and thereby eliminate a somewhat arbitrary distinction between "pre" and "pro." Furthermore, we avoid the slight difficulty involved in calling in the preprocessor again to deal with programs generated at one time.

VAN WIJNGAARDEN
I fully agree, or partially fully agree, with you. [Laughter] I only meant to disagree for psychological reasons. It's difficult for me to describe all the possible things that people might introduce in languages. I only want to be restricted to not too wild ideas. You see what I mean?

GARWICK
I agree with Dijkstra. I have great difficulty in understanding your replacement of type procedures to ordinary procedures. I can't see that, if you have a function with side effects. It's important to know where you start from - left or right, or whatever you do. I can't see how this comes into it.

VAN WIJNGAARDEN
The evaluation of the primary is exactly what the [ALGOL] Report says. Of course, you have to evaluate the primaries. And then, you have to evaluate the expression, using those primaries. Listen to my words! The preprocessor rule says in which order you have to put these statements. In front of it is your rule saying in which order you want them evaluated. I cannot say, of course, because I do not know whether your order depends on the temperature or what. It is not up to me, you see. [Laughter] This question of order comes up as a preprocessing rule. This is one of the most difficult things, you see, this question of undefinedness in the language. I try to put the burden of it on the preprocessor, rather than on the processor. The only thing I could do is write the truth in the language so illegibly that you couldn't read it. [Laughter]

GARWICK
I have a second question. I really admire the way you get rid of the goto statements, and thus really simplify things. Do you think that similar simplifications are possible in all reasonable programming languages? This is a very vague question and it really only requires a vague answer.

VAN WIJNGAARDEN
Mr. Garwick, if I answered "Yes" to that I would be admitting that there are reasonable programming languages other than ALGOL. [Laughter] This was not made as a political statement. [Laughter] In a language like COBOL there are superfluities that can be preprocessed out. There's nothing wrong with this. You can use this symbol instead of that symbol, or another way of describing things. You can map all different ways of describing something into one way. Perhaps, tomorrow, you can add another one. I don't know. That's not my affair. You put it to the preprocessor. That was the idea.

VAN DER POEL
I have two questions. Where can I find the syntax and semantics of the ⟨letter⟩ for an unspecified letter; and you even assume that ⟨letter 1⟩ is the notation for designating a particular letter that is different from, say, ⟨letter 2⟩. Can you first answer that question?

VAN WIJNGAARDEN
As a matter of fact, in the presentation given here, some things have not been defined, although explicitly used, on the assumption that either the reader would accept that a proper definition could be given, or that he would give intuitively the desired interpretation, or that he would not recognize the difficulty. [Laughter]

VAN DER POEL
My second question is: You assume that a parsing scheme is given that is unambiguous. I am not convinced that this can be done unless you tell exactly how you do it. You have been so specific in describing the better part of it. Why not then describe the preprocessor parsing, for reading, scanning backwards in its own metalanguage. You have assumed that an idempotent preprocessor exists. Why not assume also that an idempotent processor exists as well?

VAN WIJNGAARDEN
Here are surely some misunderstandings, and they are probably caused by my short way of writing and my poor way of talking. First of all, if there is such a thing as an idempotent processor then I just give you an example where it is not so. A not-idempotent processor might, for example, double the number of periods in a program. Then, if you applied it another time the number of periods would grow another level. I don't know what kind of language somebody wants to describe, and which preprocessing rules he wants to give. Now, about this other question, this parsing scheme. I tried to define it, but this parsing scheme is actually part of the machine. I cannot describe the machine itself in this language because the description of the machine is everything I cannot express, you see, by definition. I could, of course, describe it a little bit in some other logical language, and if you do not trust that, I could go a little bit deeper, but in the end I must stop. That was the idea. Now, I tried to describe exactly how I did the parsing. I gave one possible parsing scheme; I'm not interested in whether you take this one or have one of your own. You have to give one.

SAMELSON
I would like some more specific information. Apparently the processor works in the same way that the preprocessor does. Does it share truth tables? Does it have a common one? Does it have its own? And how is the division between processing rules and preprocessing rules achieved? Who does it?

VAN WIJNGAARDEN
My idea was the following. I want to separate them. If somebody wants to describe

a language this way, it's up to him to take his choice. And if he says "no preprocessor - just a processor", fine. Let him go along and define his own language that way. If he wants to say some things before, however, if he wants to define a reduced language, and over that, define a set of rules - "Instead of a period, you write this; instead of a comma, you write that" - he adds some rules to a preprocessor. It's up to him. The preprocessor belongs to the language, and I have nothing to do with it. That's why I can't formalize the preprocessor. I have nothing to do with it.

SAMELSON
I want to ask another question. You replaced goto's by procedures. I would have thought that a goto was a much more basic concept than a procedure. I want to know why you chose a procedure as a basic concept.

VAN WIJNGAARDEN
Well, because I cannot miss the procedure as a basic concept, and I didn't know how to explain the goto in my language. So I asked, "What's wrong with the goto?" I didn't know how to deal with it, so the only thing to do was simply declare it not there. OK?

MCCARTHY
I would like to contrast the approach you have taken and the approach I have taken. Specifically, ALGOL contains some strings of symbols, and then it contains some data that is not defined as strings of symbols; for example, the real numbers, and so forth. Now, I don't like strings of symbols, and you do like them, and so we have gone at ALGOL in opposite directions, almost. Namely, I have gotten rid of strings of symbols by talking about abstract syntax, while you have put strings of symbols into the data by asking for explicit representations of real numbers as strings of symbols. I think time is too short to contrast these approaches by asking which approach is better for which purpose. But I just wonder if I got the contrast straight?

VAN WIJNGAARDEN
I have two answers to this. First of all, you refer to the metaconcept "strings of symbols" and not to what we mean in ALGOL by "strings". You use "strings of symbols" to mean "sequences of symbols". This, by the way, has caused a lot of trouble, because people use the human term "string" in place of the ALGOL technical term. There is nothing wrong with this, of course. We could use the word "quotation". I think it would be wise to introduce into ALGOL this word "quotation" instead of "string". Then, in our discussions we could use the word "string" for just what we mean it to be, namely, a sequence. Now, you say that numbers are not strings in that sense. Now, I know exactly what a number is; it is a string of digits. It may be preceded by a plus or minus sign, and it may be preceded by a decimal point. There is no other thing in ALGOL that is a metaconcept called "number" of which this is the number. To me the number 13 is just the sequence of symbols 1, 3. I have never seen a "number".

MCCARTHY
Numbers are mathematical objects that are represented. It appears to me that the way definitions are given in the ALGOL Report, "numbers" are just the way that constants are to be presented in the language. At least, many people have taken it that way, rather than that the entities on which calculations are made have subscript "tens" on them.

VAN WIJNGAARDEN
It doesn't say so. It doesn't say so.

STRACHEY
Two short points: One is that I think you have introduced an extra way to call formal parameters, and that is that you call a type procedure an actual parameter.

VAN WIJNGAARDEN
I have no "type" procedures.

STRACHEY
I know, but if there is one in the source language, it must be eliminated by a different thing. You must have to do something different inside.

VAN WIJNGAARDEN
Excuse me, I'm so sorry, let's get this thing straight. I described, as an example, a preprocessor to a language you all know - ALGOL 60. I showed how, by a set of successive reductions, you could get a reduced language. I don't say that you could do this for any language, but for ALGOL 60 I have shown that you can do this. Of course, I was very careful first to take the function designator out, and so this trouble would never occur.

STRACHEY
Sorry. That's not a very important point and I don't want to stress it. Now, the other point is, your technique has been to take all the things that people think are important in languages and replace them by all the features that everybody left out. [Laughter] That is to say, you remove things and replace them by procedures that are not function designators. I would much prefer to move in exactly the opposite direction. The last thing I would want to do is remove a function (or, at least, what I call a function) because it seems a much better-understood mathematical entity than a procedure - which I call a routine - which is a complicated command. This is the direction we would like to go in - looking at the basic ideas that underly programs. I don't like the idea that your processor does not include functions, and I would very much like it to do so.

VAN WIJNGAARDEN
Of course, this is a matter of taste. I took this direction because the concept of "procedure" I can combine with all formality concepts, and the assignment into one or, I think, three lines of truth. It is so simple; it's a much more basic concept to me, than a thing like a function. This was a cheap way of doing it.

MCILROY
The idea of a preprocessor that reduces, preserving the computational and structural aspects of everything as much as possible, is very appealing to me, especially to a compiler writer. My only objection is how far you went, and I'm afraid you went a bit too far in the elimination of the goto, because this actually changes the temporal existence of values. If every goto is replaced by a procedure call, then this means that the entire history of the computation must be maintained. I'm a bit concerned about this limitation.

VAN WIJNGAARDEN
I suppose you have a certain implementation of a procedure call in mind when you say that. But this implementation is only so difficult because you have to take care of the goto statement. However, if you do this trick I devised, then you will find that the actual execution of the program is equivalent to a set of statements; no procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program: one million or two million ends. If one procedure gets to the end, that is the end of all; therefore, you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure to return. That is the whole difficulty with procedure implementation. That's why this is so simple; it's exactly the same as a goto, only called in other words.