The Mathematical Language Automath, its Usage, and Some of its Extensions*

N.G. de Bruijn

1. INTRODUCTION

1.1. Automath is a language which we claim to be suitable for expressing very large parts of mathematics, in such a way that the correctness of the mathematical contents is guaranteed as long as the rules of grammar are obeyed.

Since the notions "mathematics" and "expressing" are rather vague, we had better discuss a specific example. Assume we have a very elaborate textbook on complex function theory presenting everything from scratch. That is, we start with chapters on logic and inference rules, set theory, the number systems, some geometry, some topology, some algebra, and we never use anything that is not derived, unless it has been explicitly stated as an axiom. Assume the book has been most meticuously written, without leaving a single gap. Then we claim it is possible to translate this text line by line into Automath. The grammatical correctness of this new text can be checked by a computer, and that can be considered as a final complete check of the given piece of mathematics. Moreover we claim that it is possible to do so in practice. The line by line translation will be a matter of routine; the main difficulty lies in the detailed presentation of such a large piece of mathematics. The mere labour involved in the translation will not increase if we proceed further into mathematics.

1.2. Automath was developed in 1967–1968 at the Eindhoven University of Technology, The Netherlands. The author is indebted to Mr. L.S. van Benthem Jutting for very valuable help in trying out the language in several parts of mathematics, and both to him and to Mr. L.G.F.C. van Bree for their assistance with the programming (in ALGOL) of processors by means of which books written in Automath can be checked. In particular, Mr. Jutting is currently translating Landau's "Grundlagen der Analysis".

*Reprinted from: Laudet, M., Lacombe, D. and Schuetzenberger, M., eds., *Symposium on Automatic Demonstration*, p. 29–61, by courtesy of Springer-Verlag, Heidelberg.

1.3. In this paper we shall not attempt a complete formal definition of Automath, for which we refer to [de Bruijn 68b]. Nevertheless we hope to make the language intuitively clear in this paper. After all, the author feels that very little is essentially new in Automath, that it is very close to the way mathematicians have always been writing, and that the abbreviating system used in Automath has been taken from existing mathematical habits. But the way we handle propositions and assertions will be novel, among other things.

1.4. One of the principles of the language is that the reader (be it a human being or a computer) never has to search in the previous text for definitions or arguments. The text presented to him tells him precisely where to find information needed for checking that text.

1.5. We indicate the possibility of building languages defined in terms of Automath but adapted to special purposes (superimposed languages, see Sec. 10). This is one of the reasons for keeping Automath as primitive as possible. Actually it is little more than what might be called *the art of substitution.* Automath has an even more primitive sub-language PAL (see Sec. 4), but PAL is definitely too primitive to deal with things like predicates, quantifiers and functions. As a preliminary, we shall introduce a simple language SEMIPAL, which is *not* a sublanguage of PAL.

1.6. An Automath *book* is a sequence of lines written according to the rules of grammar. An important feature is that things which have been derived in a book (e.g. inference rules, definitions, theorems) can be applied later in that same book. It turns out to be possible that even very primitive parts of mathematical logic can be explained in that book, and therefore it is unnecessary to feed that kind of logic into the grammar.

1.7. There is one vital thing that we do not attempt to formalize: the interpretation. When reading or writing a book in a formal language like Automath, we try to be constantly aware of the relation between the text and the (mathematical or non-mathematical) objects we imagine that the text refers to. It is in this sense that many words occurring in the book (*identifiers*) are *names* of the objects outside. The book itself deals with names only. There may be several different interpretations, and there seems to be no way to discuss these interpretations in the book.

2. PRELIMINARY DESCRIPTION OF THE LANGUAGE

2.1. An Automath book is written in lines. Everything we say is said in a certain context; we shall attach a *context indicator* (or *indicator* for short) to every line. Usually the context structure can be described by a set of nested blocks (see 3.10), such as in a system of natural deduction. Lines written in a block have a kind of validity inside that block.

The context structure will make it possible to express a certain functional relationship. On top of that we have another way of dealing with functions: something that is essentially Church's lambda conversion calculus. Although these two features do not make each other entirely superfluous, they create a certain abundancy in the language. By virtue of this abundancy, many things can be written in various ways. One might experience this as a drawback, but, on the other hand, it gives something of the flexibility of every-day mathematical language.

2.2. In every line a new name (an *identifier*) is introduced. It is very essential that to every identifier a *category* is attached. In every-day language this amounts to stating what kind of a thing we are talking about. For example, we might introduce the identifier "two" and say that its category is "integer". We shall not admit that "two" has several categories simultaneously. This may have the drawback that we have to invent different notations for the integer 2 and the complex number 2. Accordingly, we have to express ourselves by means of one-to-one mappings of the integers into the complex numbers, instead of care-free identification. (We should not forget that care-free identification is a matter of tradition. The average mathematician is not inclined to identify a unit matrix with the number 1, but he identifies all 1's he knows as long as they belong to one of the "number systems".)

In connection with the above example we remark that it is by no means necessary to write mathematics in such a way that "two" has the category "integer". Another possibility, as well rooted in existing habits as the previous one, is to write that both "two" and "integer" have the category "object", and to add that "two $\in$ integer" is a true statement. If we do this, there is no harm in saying that "two $\in$ complex number" is also true.

2.3. It will be possible to introduce new categories. For this purpose we use the special symbol `type`. For example, we may introduce the identifier "integer" and attach the category `type` to it. This will have the consequence that later in the text (at least in the context where "integer" was introduced) we have the right to use "integer" as the category of an identifier.

2.4. Another feature of Automath is an abbreviation system which is essentially taken from existing conventions in mathematics; this can make the labour of writing and reading bearable, especially if we select suggestive identifiers for all notions introduced in the book. In essence, this abbreviation system occurs already in SEMIPAL.

3. STRUCTURE OF THE LINES

3.1. A line consists of 4 parts:

(i) an indicator,

(ii) an identifier,

(iii) a definition,

(iv) a category.

3.2. In every line the identifier part (ii) is a symbol that has not been used in previous lines. (This stipulation is unusual in every-day mathematics: a symbol like x is used repeatedly in different senses. But assuming we have infinitely many symbols available, it would do no harm to replace all these x's by different symbols whenever necessary.)

An identifier used as identifier part of a line will be called a *proper identifier*. There is a second kind of identifiers: those that play the rôle of *bound variable*. Again, in constrast with existing habits we shall use each bound variable only once, and a bound variable has to be different from previously introduced proper identifiers.

There are three kinds of proper identifiers: *block openers*, *primitive notions*, and *compound notions*. This depends on the definition part of the line. If the definition part is —, then the identifier part is called a block opener (or "free variable"). If the definition part is PN, then the identifier part is called a primitive notion. If the definition part is an expression (see Sec. 3.3), then the identifier part is called a compound notion.

There is a second classification of identifiers, which bears no relation to the classification above. Some identifiers are *object names*, others are *types*. An identifier is a type if and only if it is the identifier part of a line whose category is `type`. All other identifiers (including bound variables) are called object names.

3.3. The definition part of a line is either an expression or one of the symbols PN or —. If the definition part is an expression, that expression is composed of

(i) proper identifiers of previous lines;

(ii) bound variables;

(iii) the symbols

, ( ) ⟨ ⟩ [ ]

which are used as separation marks.

3.4. The category part of a line is either the symbol `type` or an expression. If it is an expression, we can say the same things as in 3.3.

3.5. The indicator part of a line is either the symbol 0 or a block opener introduced in a previous line.

The indicator is used in order to describe context.

3.6. A book is organized as a string of lines, but the context indicators induce a second structure in the form of a rooted oriented tree. The root is the symbol 0, the other vertices are the identifiers of the lines of the book. The edges are all oriented towards the root. The edge starting at the identifier x points to the indicator of the line that has x as its identifier part.

3.7. As an example we take the following book:

indicator	identifier	definition	category
0	a	PN	`type`
0	x	—	....
0	b	....	....
x	c	PN	`type`
x	y	—	`type`
y	z	—	....
y	d	....	....
0	e	....	....
x	w	—	....
w	f	....	`type`
y	g	....	....

In this example we have written .... in order to suppress expressions we do not intend to discuss at this moment. (So "...." is not a symbol used in Automath, but in our discussion about Automath.) In this example x, y, z, w are block openers, a and c are primitive notions, b, d, e, f, g are compound notions.

The tree of this book is

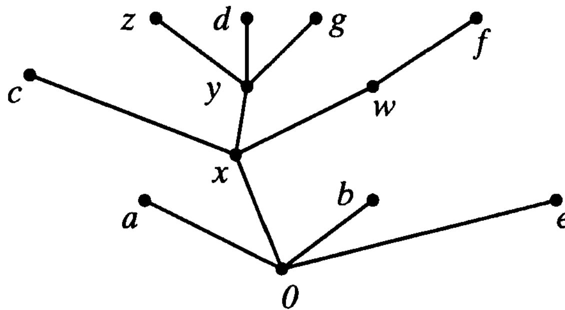

Figure 1

3.8. It has to be remarked that the tree is a combinatorial thing, and that the way it is drawn in a plane is quite irrelevant.

Note that the primitive notions and the compound notions are end-points of the tree. The block openers are usually not end-points.

To every point $\neq 0$ of the tree we can attach the definition part and the category part of the line of which that point is the identifier part. If we do this, the tree contains all the information of the book, and can be referred to as the *tree of knowledge*. But one thing the tree does not reveal: it does not show the order of the lines in the book. If we want to know whether the tree is grammatically correct, it is useful to know the order of the lines. Given the set of lines of a valid book, there may be several ways to arrange them. The only condition an arrangement has to satisfy is that no expression occurring in a line contains identifiers of later lines. All such arrangements produce legitimate books.

Anyway, if we want to extend the book by a further line then the order of the previous lines is irrelevant. At that moment, it is only the tree of knowledge that counts.

3.9. If p is a point of an oriented rooted tree, different from the root, then we can consider the subtree of all those points of the tree for which the oriented path to the root passes through p (p itself is the root of the subtree). In the case of our tree of knowledge, we shall refer to these subtrees as *blocks*. In the tree of 3.7, the point x determines the block containing x, c, y, z, d, w, f, g; the block opener of that block is x.

3.10. Quite often a book has been written in an order that makes the block structure immediately clear. This is the case if every block consists of a set of

consecutive lines. In this case we shall say we have a *nested book.* (We remark that it is not always possible to transform a correct book into a correct nested book simply by rearrangements of the lines. In order to get a nested book we might have to duplicate pieces of the text.)

In a nested book we can indicate the block structure by means of vertical bars in front of the lines. Corresponding to each block we draw a vertical line spanning all lines belonging to the block. We agree that if block B is contained in block A, then the line for B is drawn to the right of the line for A.

Once the lines have been drawn, the indicators can be omitted since they can be retraced. In the example below we present a nested book twice, once with indicators, once with bars. The version with the bars is certainly more readable for the human mathematician. A computer will of course prefer the one with the indicators.

0	a	PN	type
0	x	—	type
x	y	—	....
y	b	....	....
x	z	—	....
z	c	PN	....
z	w	—	....
w	d	....	....

a	:=	PN	type
x	:=	—	type
y	:=	—	....
b	:=	....	....
z	:=	—	....
c	:=	PN	....
w	:=	—	....
d	:=	....	....

As in this example we shall always separate identifier part and definition part by the symbol := which suggests that the identifier on the left is defined by the expression on the right.

Needless to say, the vertical bars and the symbol := do not belong to the language. They are just devices for easier reading. Quite often we shall print both the vertical bars and the indicators.

3.11. Sometimes we shall talk about the *indicator string* of a line. If the indicator is 0, the indicator string is empty. In all other cases the indicator string describes the reversed path from the indicator in question to the root of the tree (excluding the root). For example, the indicator string of the last line in the example of 3.7 is (x, y), the one of the last line in the example of 3.10 is (x, z, w).

4. HOW TO WRITE PAL

4.1. PAL is a sublanguage of Automath, in the sense that every correct PAL book is also a correct Automath book. PAL is quite easy to learn. In PAL we do not use the lambda conversion, and we have no bound variables.

Let us take an example. At this stage the reader must not expect an example with deep mathematical significance, since that would require quite a long book. The interpretation we have in mind is this one: Assume that *nat* (natural number) and *real* (real number) are available as categories. If a and b are given reals, then their product is introduced as a primitive notion. If n is a natural number and x is a real, then the power x^n is introduced as a primitive notion. If n is a natural number and y is a real number, then we define $d(y) := y^n$; $e(y) := d(y) \cdot y \ (= y^{n+1})$; $f(y) := d(y) \cdot d(y) \ (= y^{2n})$; $g(y) := e(d(y))$ $(= y^{n(n+1)})$. In PAL this can be written as follows:

(indicator)	(identifier)		(definition)	(category)	
0	*nat*	:=	PN	**type**	1
0	*real*	:=	PN	**type**	2
0	*a*	:=	—	*real*	3
a	*b*	:=	—	*real*	4
b	*prod*	:=	PN	*real*	5
0	*n*	:=	—	*nat*	6
n	*x*	:=	—	*real*	7
x	*power*	:=	PN	*real*	8
n	*y*	:=	—	*real*	9
y	*d*	:=	$power(n, y)$	*real*	10
y	*e*	:=	$prod(d, y)$	*real*	11
y	*f*	:=	$prod(d, d)$	*real*	12
y	*g*	:=	$e(d)$	*real*	13

This happens to be a nested book in the sense of 3.10, but that does not have any consequence for the present discussion. It is also a very simple case in the sense that the categories are all very simple.

Although we are not going to do it in this paper, it may help the reader to provide the identifier parts (as far as they are not block openers) with the indicator strings in parentheses. That means that he writes $prod(a, b)$ in line 5, $power(n, x)$ in line 8, $d(n, y)$ in line 10, $e(n, y)$ in line 11, $f(n, y)$ in line 12, $g(n, y)$ in line 13. This makes it easy to see what we intend with the other expressions: $prod(d, d)$ indicates that both a and b in $prod(a, b)$ are replaced by d. Now what does $e(d)$ mean in line 13? By line 11, e depends on two variables (n and y). We agree that we add the letters of the indicator string of line 11 on the left until we have enough entries. So $e(d)$ has to be interpreted as $e(n, d)$: the first entry of the string n, y is added on the left. In general: if p is introduced with indicator string $(x_1, ..., x_n)$, and if $k < n$, then $p(Z_1, ..., Z_k)$ has to be interpreted as $p(x_1, ..., x_{n-k}, Z_1, ..., Z_k)$.

4.2. Before we describe the rules of PAL, we first describe a simpler language to be called SEMIPAL. This language is different from PAL and Automath in that it does not attach a category to a line. Its relation to PAL is simple. If we just cancel from a correct PAL book the entire category column, then we get a correct SEMIPAL book. (On the other hand, we can always transform a correct SEMIPAL book into a correct PAL book just by adding **type** as the category of every line.)

4.3. The rules of SEMIPAL are given in this and the next section. The reader may take the 13 lines of Sec. 4.1 for an example, by just cancelling the category column.

(i) As the first line of the book any one of the lines

0	....	:=	PN
0	....	:=	—

is acceptable. (Here "...." stands for an arbitrary identifier.)

(ii) We can add an $(n+1)$-st line to a correct SEMIPAL book Λ of n lines by writing

$$u \quad .... \quad := \quad \Sigma\,,$$

where u is either 0 or one of the previous block openers, and Σ is either —, or PN, or an expression valid at u, a notion to be defined presently.

4.4. The notion *expression valid at u* (where u is an indicator) is relative to the given correct book Λ. We define it by recursion.

(1) If b is a block opener, either equal to u or contained in the indicator string (see Sec. 3.11) of u, then b is an expression valid at u.
Example: At y the expressions n, y are valid.

(2) If b is the identifier of a line of Λ, but not a block opener, and if the indicator of that line is either 0 or u or contained in the indicator string of u, then b is an expression valid at u.
Example: At y the expressions *nat*, *real*, d, e, f, g are valid.

(3) Let b be the identifier part of one of the lines of Λ, and assume that b is not a block opener. Let n be the length of the indicator string of b. Let k be a second integer, $0 < k \leq n$. We assume that $\Sigma_1, ..., \Sigma_k$ are expressions valid at u. If $n > k$ we have the extra assumption that the $(n-k)$-th entry of the indicator string of b is an expression valid at u (that is, it is equal to u or contained in the indicator string of u). Then

$$b(\Sigma_1, ..., \Sigma_k)$$

is an expression valid at u.

4.5. In the SEMIPAL book that is obtained from the example of Sec. 4.1 (by omitting the category column) we give a few examples of expressions valid at y:

$$n;\quad y;\quad f;\quad prod(d,f);\quad e(d);\quad power(n,f);$$
$$power(f);\quad d(y);\quad d(n,y);\quad e(prod(e,e)).$$

4.6. As a preparation to discussion of normal forms, we define the *completion* of an expression valid at u. Let Σ be an expression valid at u; its completion Σ' will also be valid at u.

(i) If Σ consists of a single block opener, then $\Sigma' = \Sigma$.

(ii) Let $\Sigma = b(\Sigma_1, ..., \Sigma_k)$ (see the end of Sec. 4.4) and let $u_1, ..., u_n$ be the indicator string of b. Then

$$\Sigma' = b(u_1, ..., u_{n-k}, \Sigma_1, ..., \Sigma_k) .$$

If $k = 0$, $n - k \neq 0$ this has to be read as $b(u_1, ..., u_{n-k})$, if $k \neq 0$, $n - k = 0$ as $b(\Sigma_1, ..., \Sigma_k)$, if $k = n - k = 0$ it has to be read as just b.

4.7. An expression is said to have *normal form* (in the sense of SEMIPAL) if it contains no compound notions (see Sec. 3.2).

Let Σ be an expression valid at u. We shall define, again recursively, a reduction to normal form Σ^*. We first complete the expression Σ to Σ' (4.6).

If Σ' is a single identifier, but not a compound notion, then we take $\Sigma^* = \Sigma'$.

If Σ' is a single identifier and if that identifier is a compound notion, we define Σ^* to be the normal form of Ω, where Ω is the definition part of the line whose identifier part is Σ'.

If $\Sigma' = b(\Sigma_1, ..., \Sigma_n)$ with $n > 0$, and if b is a primitive notion, then we take

$$\Sigma^* = b(\Sigma_1^*, ..., \Sigma_n^*) ,$$

where Σ_i^* is the normal form of Σ_i $(i = 1, ..., n)$.

If $\Sigma' = b(\Sigma_1, ..., \Sigma_n)$ with $n > 0$, and if b is a compound notion, with indicator string $u_1, ..., u_n$, then we obtain Σ^* as follows. Let Ω^* be the normal form of the definition part of the line whose identifier is b. In Ω^* replace every occurrence of u_i by Σ_i^* (the normal form of Σ_i). This gives Σ^*.

Warning: The substitution of the Σ_i^* for u_i is only carried out for explicit occurrences of u_i in Ω^*, and not for new u_i's that arise *after* substitution (the Σ_i^*'s themselves may contain u_j's).

As an example we give the normal form of the expression $e(d)$ of line 13 in the example of Sec. 4.1:

$$prod(power(n, power(n, y)), power(n, y)) \ .$$

4.8. Two expressions Σ_1, Σ_2 both valid at u are called *definitionally equal* if they have the same normal form. If we want to show definitional equality it is not always necessary to compute these normal forms; it will often suffice if we can transform both forms into a single form by partial reduction.

If we replace an expression in a correct SEMIPAL book by a definitionally equal one, we get a new correct SEMIPAL book. The normal forms of corresponding expressions in both books will be the same.

4.9. We shall describe the notion of a correct PAL book in two stages. We start with a book Λ written according to the preliminary description of Sec. 3. That is, the definition part of a line is —, or PN, or an expression; the category part is **type** or an expression; the indicator part is 0 or a previous block opener. By a certain duplication operation to be described presently, we get something which we shall require to be a correct SEMIPAL book Λ'. Finally, we shall require certain conditions regarding the categories.

The duplication means the following thing. We replace every line

$$u \quad a \quad := \quad \Omega \quad \Sigma$$

(where u may be 0 or a block opener, Ω may be an expression or — or PN, Σ is an expression or **type**) by two lines

$$u \quad a^+ \quad := \quad \Sigma$$
$$u \quad a \quad := \quad \Omega \ ,$$

unless Σ is **type**, in which case we write the single line

$$u \quad a \quad := \quad \Omega \ .$$

We of course assume that for every identifier we can create an entirely new identifier by adding the plus sign.

As an example we deal with the first 5 lines of the book of Sec. 4.1:

0	*nat*	:=	PN
0	*real*	:=	PN
0	a^+	:=	*real*
0	*a*	:=	—
a	b^+	:=	*real*
a	*b*	:=	—
b	$prod^+$	:=	*real*
b	*prod*	:=	PN

4.10. We define the notion "correct PAL book" by recursion. The definition will be such that if Λ is correct, then Λ' is a correct SEMIPAL book.

A one-line book is correct if and only if that line has one of the following two forms:

0	....	:=	PN	`type` ,
0	....	:=	—	`type` .

Now assume that a book Λ consisting of n lines is a correct PAL book. We shall state the conditions for any line to be added.

(i) The indicator u is either 0 or a block opener of Λ.

(ii) The definition part is either —, or PN, or an acceptable expression at u (see Sec. 4.11 for this).

(iii) The category part is either **type**, or an acceptable expression at u with category **type**. In the case where the definition part is an expression (see (ii)), we require that the category part is definitionally equal (in the sense of the SEMIPAL book Λ') to the category of that expression.

4.11. Let u be one of the block openers of the SEMIPAL book Λ' obtained by duplication of Λ. We will define a collection of expressions that we call *acceptable* at u; to each one of these expressions we will attach what we will call a *category*. The latter is either an expression or the symbol **type**. The expressions to be considered for acceptability, as well as their categories, will only contain identifiers of Λ, and no identifiers with + signs attached to them. The acceptable expressons will be automatically valid at u in the sense of Sec. 4.4.

The description of "acceptable" closely resembles the one of "valid" (Sec. 4.4).

(1) Let b be one of the following:

- a block opener whose indicator string is contained in the indicator string of u;
- the identifier of a line of Λ (but not a block opener) whose indicator is either 0, or u, or contained in the indicator string of u (cf. (1) and (2) in Sec. 4.4).

Then b is an acceptable expression at u, and its category is the category part of the line whose identifier is b.

(2) Let b be the identifier part of one of the lines of Λ, and assume that b is not a block opener. Let n be the length of the indicator string of b. Let k be a second integer, $0 \le k \le n$. We assume that the expressions $\Sigma_1, ..., \Sigma_k$ are acceptable at u, with categories $\Omega_1, ..., \Omega_k$. If $n > k$ we have the extra condition that the $(n-k)$-th entry of the indicator string of b is either equal to u or contained in the indicator string of u. Let $v_1, ..., v_k$ be the last k entries in the indicator string of b. We require, for $i = 1, ..., k$, that

$$v_i^+(\Sigma_1, ..., \Sigma_{i-1}) \tag{1}$$

is definitionally equal (in the sense of Λ') to Ω_i. (If $i = 1$ we have to read (1) as v_1^+. If any of the v_i^+ does not occur in Λ', we have to read (1) as **type**, and the condition is just that $\Omega_i = \textbf{type}$.)

Under these conditions we proclaim $b(\Sigma_1, ..., \Sigma_k)$ to be acceptable at u, and we give it the category $b^+(\Sigma_1, ..., \Sigma_k)$. If b^+ does not occur in Λ', the new expression $b(\Sigma_1, ..., \Sigma_k)$ is given the category **type**.

One minor modification should be made: we promised that the category would not be an expression containing identifiers with plus signs. Therefore we replace $b^+(\Sigma_1, ..., \Sigma_k)$ by the result of an application of a substitution such as described at the end of Sec. 4.7.

5. HOW TO USE PAL FOR MATHEMATICAL REASONING

5.1. In Section 4 we explained how to express things by means of PAL. Seemingly, expressing things covers only a small part of mathematics, for usually we are interested in proving statements. Mathematics has the same block structure as we have in PAL, but there are two ways to open a block. One is by introducing a variable that will have a meaning throughout the block, the other one is by making an assumption that is valid throughout the block. We shall be able to deal with the second case as efficiently as with the first one, if we represent statements by categories. Saying we have a thing in such a category

means asserting the statement. This can be done in three ways: by means of —, or PN, or an expression. These three correspond to assertion by assumption, by axiom, by proof, respectively.

5.2. As an example we shall deal with equality in an arbitrary category. The following piece of text introduces equality as a primitive notion, and states the three usual axioms.

0	ξ	:=	—	**type**	1
ξ	x	:=	—	ξ	2
x	y	:=	—	ξ	3
y	*is*	:=	PN	**type**	4
x	*reflex*	:=	PN	$is\,(x,x)$	5
y	*asp* 1	:=	—	$is\,(x,y)$	6
asp 1	*symm*	:=	PN	$is\,(y,x)$	7
asp 1	z	:=	—	ξ	8
z	*asp* 2	:=	—	$is\,(y,z)$	9
asp 2	*trans*	:=	PN	$is\,(x,z)$	10

This book is *not* a nested one since line 5 does not belong to the block opened by y. Even so, the vertical bars, with an interruption at line 5, can be helpful.

We now show how this piece of text can be used in later parts of the book. Assume we have the following lines (in some order) in the book:

0	η	:=	....	**type**
0	a	:=	....	η
0	b	:=	....	η
0	*known*	:=	....	$is(\eta, a, b)$

We wish to derive a line:

0 *result* := .... $is\,(\eta, b, a)$.

We have to find a definition part for this line. What we want is to apply line 7. The indicator string is $(\xi, x, y, asp\ 1)$. In ordinary mathematical terms, we have to furnish a value for ξ, a value for x, a value for y, and a proof for the statement obtained from "$x = y$" by these substitutions. A proof for the statement means, in our present convention, something of the category $is\,(\eta, a, b)$. Indeed we have something, viz. "*known*". The reader can easily verify that

0 *result* := $symm\,(\eta, a, b, known)$ $is\,(\eta, b, a)$

is an acceptable line.

The above application was given entirely in context 0, but it can be done in any block that contains η, a, b and *known*.

5.3. We are, of course, inclined to see the categories as classes, and things having that category as elements of those classes. If we want to maintain that picture, we have to say that the category "$is\,(\xi, x, y)$" consists of all proofs for $x = y$. In this picture the usual phrase "assume $x = y$" is replaced by "let p be a proof for $x = y$". Another aspect is that we have to imagine the category "$is\,(\xi, x, y)$" to be empty if the statement $x = y$ is false. The latter remark points at a difference between these assertion categories and the "ordinary" categories like "*nat*" and "*real*" in Sec. 4. In the spirit of the example of Sec. 4 it is vital to know what the expressions are, and it seems pretty useless to deal with empty categories. With the assertion categories it is different. The interesting question is whether we can find something in such a category, it doesn't matter what.

5.4. A modern mathematician has the feeling that asserting is something we do with a *proposition*. The author thinks that this is not the historic point of view. The primitive mathematical mind asserts the things it can, and is unable to discuss things it cannot assert. To put it in a nicer way, it has a kind of constructivist point of view. It requires a crooked way of thinking to build expressions that can be doubted, i.e. to build things that might or might not be asserted. A possible way to do this in PAL is to talk about the category "*bool*" consisting of all propositions, and to attach to each proposition an assertion category. We start the book like this:

0	*bool*	:=	PN	**type**
0	| *b*	:=	—	*bool*
b	| *TRUE*	:=	PN	**type**

The standard interpretation is simple. If we write in a certain context

$$\ldots \quad := \quad \ldots. \qquad TRUE(c)\ ,$$

where c is (in that context) a proposition, then the interpretation in every-day mathematical language is that we are asserting c.

5.5. In PAL we are able to write axioms and prove theorems about propositions (e.g. tautologies). In later parts of the book we will be able to use these axioms and theorems (just like the derivation of "result" in Sec. 5.2). This means that in a PAL book we are able to derive inference rules that can be applied later in that same book.

As a very primitive example we shall write the following in PAL. After introducing *bool* and *TRUE* we introduce the conjunction of two propositions. We present some axioms concerning that conjunction, and we show that from $x \wedge y$ we can derive $y \wedge x$. Finally we show how in a later piece of text the result can be used as an inference rule.

0	*bool*	:=	PN	**type**
0	*b*	:=	—	*bool*
b	*TRUE*	:=	PN	**type**
0	*x*	:=	—	*bool*
x	*y*	:=	—	*bool*
y	*and*	:=	PN	*bool*
y	*asp* 1	:=	—	*TRUE*(x)
asp 1	*asp* 2	:=	—	*TRUE*(y)
asp 2	*ax* 1	:=	PN	*TRUE*(*and*)
y	*asp* 3	:=	—	*TRUE*(*and*)
asp 3	*ax* 2	:=	PN	*TRUE*(x)
asp 3	*ax* 3	:=	PN	*TRUE*(y)
asp 3	*theorem*	:=	*ax* 1(y, x, *ax* 3, *ax* 2)	*TRUE*(*and*(y, x))
0	*u*	:=	....	*bool*
0	*v*	:=	....	*bool*
0	*known*	:=	....	*TRUE*(*and*(u, v))
0	*derived*	:=	*theorem*(u, v, *known*)	*TRUE*(*and*(v, u))

5.6. The reader will have observed from the above examples that we do not need to subdivide our text into parts like "theorem", "proof", "definition", "axiom". Every line is a result that can be used whenever we wish. It may require a large number of lines to translate the proof of a theorem into PAL. (Needless to say, we can always try to reduce the number of lines, but that makes the lines more complicated and hard to read.) Some of the lines represent definitions of notions introduced only for the sake of the proof. Other lines represent subresults, usually called lemmas. The usual idea about theorems and proofs is, at least formally, that we are not allowed to refer to results obtained inside a proof. In PAL (and in Automath), however, we are free to use every line everywhere. We never announce a theorem before the proof starts, the result is not stated before it has been derived.

6. EXTENDING PAL TO AUTOMATH

6.1. It was shown in Sec. 4 how we can deal with functional relationship in PAL. Once a function has been defined (either by PN or by definition in terms of previous notions) it can be applied. That is, a function f is introduced by saying what the value of $f(x)$ is for every x of a certain category. And if we have, at a later stage, an expression Σ having that same category, it will be possible to talk about $f(\Sigma)$. A thing that we can *not* write in PAL, however,

is "let f be any function, mapping category Σ_1 into category Σ_2". If we wish to deal with such mappings the way it is done in mathematics, we want several things:

(i) We need the facility of building the category of the mappings of Σ_1 into Σ_2.

(ii) If f is an element of that mapping category, and if x is something having category Σ_1, then we have to be able to form the image of x under f.

(iii) If a mapping of Σ_1 into Σ_2 is explicitly given in the PAL way then we have to be able to recognize that mapping as a member f of that mapping category.

(iv) If we apply (ii) to the f obtained in (iii), we can (making x a block opener) obtain a function given in the PAL way. This function should be equivalent to the one we started from in (iii).

6.2. Let us consider (iii) more closely. The "PAL way" of giving a function is the following one: We have somewhere in the book

u	x	:=	—	Σ_1	1
x	v	:=	Λ	Σ_2	2

where Λ is an expression possibly depending on x. (That is, its normal form may contain x.) But it is only fair to remark that Σ_2 may also depend on x; Σ_1, on the other hand, can *not* contain x. Let us assume that neither Σ_1 nor Σ_2 is the symbol **type**.

The mapping described here attaches to every x of type Σ_1 a value depending on x, which value has category also depending on x. We shall use the notation

$$[x : \Sigma_1]\,\Sigma_2$$

for the category of this mapping, and

$$[x : \Sigma_1]\,\Lambda$$

for the mapping itself. There is an objection against using the old identifier x for this new purpose, and therefore we replace it by a new identifier t. This t will never occur as identifier part of a line. It is called a *bound variable*, and we may assume that it will be used here, but never again.

We shall write $\Omega_x(\Sigma)\Lambda$ for the result of substitution of Σ for x in the expression Λ. (It should be remarked that Λ may contain x implicitly, which can happen if the above block contains lines between line 1 and line 2. In order to

make such implicit occurrences explicit, we have to transform Λ by application of definitions up to a point where further implicit occurrence is impossible.)

We can now phrase the rule of *functional abstraction*: In Automath we have the right to deduce from lines 1 and 2 the acceptability of the line

$$u \quad \ldots \quad := \quad [t:\Sigma_1]\,\Omega_x(t)\Lambda \qquad [t:\Sigma_1]\,\Omega_x(t)\Sigma_2 \qquad 3$$

Accordingly we have the right to consider $[t:\Sigma_1]\,\Omega_x(t)\,\Sigma_2$ as a category. So if we have (if Σ_1 and Σ_2 are expressions)

$$u \quad x \quad := \quad \text{—} \quad \Sigma_1 \qquad 4$$
$$x \quad w \quad := \quad \Sigma_2 \quad \texttt{type} \qquad 5$$

we have the right to add

$$u \quad \ldots \quad := \quad [t:\Sigma_1]\,\Omega_x(t)\Sigma_2 \qquad \texttt{type} \qquad 6$$

This makes it possible to open a new block with

$$u \quad f \quad := \quad \text{—} \quad [t:\Sigma_1]\,\Omega_x(t)\Sigma_2\ , \qquad 7$$

that is, we can start an argument with: let f be any mapping of the described kind. We also have the possibility to write line 7 with PN instead of —.

6.4. Now returning to point (ii) Sec. 6.1, we introduce the following rule. If we have a line

$$u \quad \ldots \quad := \quad \Gamma \qquad [t:\Sigma_1]\,\Omega_x(t)\Sigma_2 \qquad 8$$

and also a line

$$u \quad \ldots \quad := \quad \Delta \qquad \Sigma_1 \qquad 9$$

then we take the liberty to write

$$u \quad \ldots \quad := \quad \langle\Delta\rangle\,\Gamma \qquad \Omega_x(\Delta)\Sigma_2\ . \qquad 10$$

The interpretation is that $\langle\Delta\rangle\,\Gamma$ is the result of the substitution of Δ into Γ. We write this instead of $\Gamma(\Delta)$ since, in the case that Γ is a single identifier, the latter notation already had an entirely different meaning in PAL: it was used to change context. That is, $\Gamma(\Delta)$ is the mapping we obtain from Γ if we substitute Δ for u, and it is even questionable whether this is possible, since u need not be of category Σ_1.

6.5. In connection with this notation $\langle\ \rangle$ we take the liberty to extend the notion of definitional equality by the following pair of rules:

(i) If Σ_1, Σ_2, Σ_3 are expressions, where Σ_2 contains the bound variable t, but Σ_1 and Σ_3 do not, then we postulate the definitional equality of

$$\langle\Sigma_3\rangle\,[t:\Sigma_1]\,\Sigma_2 \quad \text{and} \quad \Omega_t(\Sigma_3)\Sigma_2\;.$$

That is, it does not make a difference whether substitution is carried out *before* or *after* functional abstraction.

(ii) If Σ_1 and Σ_2 are expressions that do not contain the bound variable x, then we postulate the definitional equality of

$$[x:\Sigma_1]\,\langle x\rangle\,\Sigma_2 \quad \text{and} \quad \Sigma_2\;.$$

The above rules (i) and (ii) explain why we prefer to write $\langle x\rangle\, f$ instead of $f\,\langle x\rangle$. By way of these rules, $\langle x\rangle\, f$ is in agreement with the convention $[t:\Sigma_1]\,\Sigma_2$ for functional abstraction, and the latter is in agreement with the general mathematical habit to write quantifiers like

$$\forall_{x\in S}\;,\quad \bigcup_{x\in S}\;,\quad \prod_{n=1}^{\infty}$$

to the left of the formulas they act on.

6.6. The description of Automath in the preceding sections was not as complete as the description of SEMIPAL and PAL in Sec. 4. For a complete and more formal definition of Automath we refer to the report mentioned in Sec. 1.2.

7. HOW TO USE AUTOMATH FOR MATHEMATICAL REASONING

7.1. If we write elementary mathematical reasoning in PAL as described in Section 5, one of the first things we can *not* do is to derive an implication. There are two things we wish to do with implication, and only one of the two can be done in PAL.

First assume we have introduced implication as a primitive notion, then it is easy to write "modus ponens" as an inference rule:

0		*bool*	:=	PN	type	1
0	|	*b*	:=	—	*bool*	2
b	|	*TRUE*	:=	PN	type	3
b	| |	*c*	:=	—	*bool*	4
c	| |	*impl*	:=	PN	*bool*	5
c	| | |	*asp* 1	:=	—	$TRUE(b)$	6
asp 1	| | | |	*asp* 2	:=	—	$TRUE(impl)$	7
asp 2	| | | |	*modpon*	:=	PN	$TRUE(c)$	8

By means of this piece of text we are able to use the inference rule

$$\frac{A,\ A \Rightarrow B}{B}$$

in all possible situations.

The second thing we want to do is this. If we have

0		*p*	:=	....	*bool*	9
0		*q*	:=	....	*bool*	10
0	|	*asp* 3	:=	—	$TRUE(p)$	11
asp 3	|	*then*	:=	....	$TRUE(q)$	12

(it might have been given in any other context instead of 0) then we want to construct something in $TRUE(impl(p,q))$. This cannot be done by means of the rules of PAL.

The problem can be solved in Automath, however. We first say that if we have a mapping from $TRUE(b)$ into $TRUE(c)$, then $impl(b,c)$ is true:

	: :					
c	| | |	*asp* 4	:=	—	$[x : TRUE(b)]\ TRUE(c)$	13
asp 4	| | | |	*axiom*	:=	PN	$TRUE(impl)$	14

Using the axiom, and functional abstraction, we can derive from lines 11, 12

0	*first*	:=	$[y : TRUE(p)]\ then(y)$	$[y : TRUE(p)]\ TRUE(q)$	15
0	*second*	:=	$axiom(p, q, first)$	$TRUE(impl(p,q))$	16

That is, we have derived an assertion of $impl(p,q)$. So we have the inference rule

$$\frac{\frac{A}{B}}{A \Rightarrow B}$$

available in all possible cases.

If we wish, we can write the application of this inference rule in one line instead of two, viz.

0	...	:=	$axiom(p, q, [y : TRUE(p)]\ then(y))$	$TRUE(impl(p, q))$	17

7.2. As a second example we introduce the all-quantifier for a predicate P on an arbitrary type ξ.

0	*bool*	:=	PN	`type`	1
0	b	:=	—	*bool*	2
b	*TRUE*	:=	PN	`type`	3
0	ξ	:=	—	`type`	4
ξ	P	:=	—	$[u : \xi]\ bool$	5
P	*all*	:=	PN	*bool*	6
P	x	:=	—	ξ	7
x	*asp* 5	:=	—	$TRUE(all)$	8
x	*ax* 1	:=	PN	$TRUE(\langle x \rangle P)$	9
P	*asp* 6	:=	—	$[v : \xi]\ TRUE(\langle v \rangle P)$	10
asp 6	*ax* 2	:=	PN	$TRUE(all)$	11

Note the close resemblance between the text of Sec. 7.1 and this one. Actually we are able to define "*impl*" in terms of "*all*": we can write instead of line 5 of Sec. 7.1

$$c \quad impl \quad := \quad all(TRUE(b), [t : TRUE(b)]\ c) \quad bool$$

If we do this after having accepted the text of 7.2, then we can replace the PN's in line 8 and line 14 of Sec. 7.1 by proofs. The reader may check that the PN in line 8 (Sec. 7.1) can be replaced by

$$ax\ 1(TRUE(b), [s : TRUE(b)]\ c,\ asp\ 1,\ asp\ 2)\ ,$$

and the one in line 14 (Sec. 7.1) by

$$ax\ 2(TRUE(b), [s : TRUE(b)]\ c,\ asp\ 4)\ .$$

7.3. Next we discuss the existence quantifier. There are various different approaches to this. The simplest one, and therefore the easiest one for application, is connected with the Hilbert operator. It says, if for any given category there exists an object for which a given property holds, then we have a way of selecting such an object as if we were in possession of a standard algorithm that selects for us.

We can write this as follows. We start again with the introduction of *bool* and *TRUE*, then we take an arbitrary category ξ and an arbitrary predicate on that category, and we introduce existence as a primitive notion. It says that "existence" is true if and only if we have something in that category ξ.

0	*bool*	:=	PN	**type**	1
0	*b*	:=	—	*bool*	2
b	*TRUE*	:=	PN	**type**	3
0	ξ	:=	—	**type**	4
ξ	*P*	:=	—	$[u:\xi]$ *bool*	5
P	*exists*	:=	PN	*bool*	6
P	*v*	:=	—	ξ	7
v	*asp* 1	:=	—	$TRUE(\langle v\rangle P)$	8
asp 1	*axiom* 1	:=	PN	$TRUE(exists)$	9
P	*asp* 2	:=	—	$TRUE(exists)$	10
asp 2	*Hilbert*	:=	PN	ξ	11
asp 2	*axiom* 3	:=	PN	$TRUE(\langle Hilbert\rangle P)$	12

In combination with other axioms this way of defining existence easily leads to non-contructive things, e.g. the axiom of choice.

A different way of introducing existence is to say that it is not true that the negation of the predicate holds for all objects in the given category. This of course requires a definition of negation, which can be done in several ways. We shall not discuss it here.

The difficulties about existence arise already at a lower level, viz. with the notion of non-emptiness of a category. In that case the following may be a useful substitute for the kind of non-emptiness related to the Hilbert operator:

0	ξ	:=	—	**type**
ξ	*NEPTY*	:=	$[c:bool]\,[u:[x:\xi]\,TRUE(c)]\,TRUE(c)$	**type**

If we have something in *NEPTY*, if c is any proposition, and if we can prove that whenever we have an x in ξ then c is true, then we have proved c. So if we have something in *NEPTY*, we have a kind of inference rule: If we want to prove a proposition c then we may act as if we know an x with category ξ.

7.4. There is no objection against higher order predicate calculus in Automath. For example, we can talk about the category R of all predicates on the category of natural numbers, say, about the category S of all predicates on R, etc.:

0	*nat*	:=	....	**type**
0	*R*	:=	$[n:nat]$ *bool*	**type**
0	*S*	:=	$[r:R]$ *bool*	**type**

7.5. Every language has its advantages and disadvantages. The disadvantages of Automath are obvious: it is tedious to have to write in full detail, carefree identification of things in different categories is forbidden (see Sec. 2.2), and

embedding of types into other types is not an automatic facility. In order to compensate for these disadvantages, the user should try to exploit the advantages the language has. One advantage is that we do not have to announce theorems and lemmas in a formal way, and therefore repetition of arguments is much easier suppressed than in ordinary mathematics. And, of course, we can invent all sorts of tricks. We present just one such trick here.

Consider an axiom like the line *TRUE* in Sec. 5.4. Once we have written it this way, we cannot get rid of it: if we want to do mathematics without it, we have to write a new book. There is a way, however, to introduce the axiom in such a way that, so to speak, it is only available to those who have authority to use it. We introduce a new primitive notion *AUTH* (for authority) and then state the axiom for those users who have something in *AUTH*:

0	*bool*	:=	PN	**type**
0	*AUTH*	:=	PN	**type**
0	| *a*	:=	—	*AUTH*
a	| | *b*	:=	—	*bool*
b	| | *TRUE**	:=	PN	**type**

If later we have c in *AUTH* and d in *bool*, we can use $TRUE^*(c,d)$. If c in *AUTH* is valid in a large part of the book, we can get rid of the awkward obligation to mention our authority, by defining (in a context where c is available)

| *e*	:=	—	*bool*
| *TRUE*	:=	$TRUE^*(c,e)$	**type**

and now we can write $TRUE(f)$ for any proposition f.

8. UNSOLVED PROBLEMS ABOUT AUTOMATH

8.1. It is very probable (but not yet proved) that the following is true. If the lines

u	...	:=	Σ_1	Λ_1
u	...	:=	Σ_2	Λ_2

occur in a book, if Σ_1 and Σ_2 are definitionally equal, then Λ_1 and Λ_2 are definitionally equal. We only say roughly what definitional equality is: Two expressions are definitionally equal if one of them can be transformed into the other by replacing an identifier in one of the expressions by the expression that defines it, and also by application of one of the operations of the lambda calculus. They are also called definitionally equal if they can be connected by a chain of pairwise definitionally equal expressions.

We do not express the notion by means of normal forms, as in 4.7, since we are not yet sure about normal forms.

8.2. Probably every expression occurring in an Automath book is definitionally equal to an expression that does not contain any $\rangle$ followed by a [. This means an expression

$$[\beta_1 : \Sigma_1] \ldots [\beta_k : \Sigma_k] \langle\Gamma_1\rangle \ldots \langle\Gamma_h\rangle \beta(\Theta_1, \ldots, \Theta_m)$$

(possibly $k = 0$, $h = 0$, or $m = 0$), where the Greek capitals again represent expressions of that form, the $\beta_1, \ldots, \beta_k$ are bound variables, and β is either a block opener or the identifier part of a line with PN.

9. PROCESSORS FOR AUTOMATH

9.1. A processor is a computer program that enables a computer to check line by line whether any given input represents a correct Automath book.

One of the things the computer gets to do is to check whether two expressions are definitionally equal. Even if the conjectures of Sec. 8 are true, it can be very impractical to use normal forms for checking that equality. It is already impractical in PAL, where there is no difficulty with the normal forms (see Sec. 4.7).

A good processor should have a good strategy for checking definitional equality. In cases where the general strategy is failing, it may pay to assist the computer by giving hints as to what to do first.

It is to be expected that very few hints will be needed in general. That is, at least as long as we do not try to condense a larger number of lines into a single one. Such a condensation is quite often possible, it saves identifiers, but makes things harder to write and harder to check. (An additional disadvantage of condensed writing is the repetition of expressions which might have been abbreviated by means of extra lines. Another aspect of the same thing is giving an argument twice where a lemma might have been more efficient.)

9.3. There are several attractive possibilities for man-machine interaction if a terminal is available for direct communication in conversational mode. (The Automath processor in operation in 1968 at the Eindhoven University of Technology, Eindhoven, did not yet provide such facilities.) For lines the machine rejects, it can produce diagnostics by means of which the operator can carry out corrections or add hints. It will be very practical for the operator to suppress the category of a line (unless the definition is — or PN), and to ask the machine what category it finds. If it does not coincide with the one the operator has in

mind, the operator can ask the machine to check definitional equality of the two expressions.

10. POSSIBILITIES FOR SUPERIMPOSED LANGUAGES

10.1. For practical purposes it will be attractive to make languages which bear the same relation to Automath as a programming language has to some particular machine language. We shall call such languages *superimposed* on Automath. They require a compiler for translation into Automath.

10.2. A very simple thing a superimposed language might do is admitting repetition of names (such as the repeated use of the letter x for many different purposes in the book). The compiler has to rename everything in order to meet the requirement that in Automath the identifier parts of the lines are distinct.

10.3. In complicated cases the superimposed language will require a fixed correct Automath book as a *basis*. If we have written a book in the superimposed language, then the compiler starts from the basis, and next it translates the given book into Automath lines which are subsequently added to the basis, and checked by the Automath processor.

10.4. In a superimposed language standard mathematical notation might be used more freely. For example, in the superimposed language one might write $p := a + b + c$. The compiler sees that a, b, c were previously introduced as *reals*, it sees that no change of context has been mentioned, it knows that "*real*" and "*plus*" are identifiers in the basis. It writes

$$p := plus(plus(a, b), c)\ real$$

and it keeps the context indicator of the previous line.

10.5. A superimposed language might be very different from Automath in its approach to things like propositions, assertions, predicates. The user of the superimposed language need not even notice that Automath has a slightly unconventional approach to these things.

10.6. It is not strictly necessary that the text presented in a superimposed language is entirely unambiguous and free of gaps. Just as the human mathematician has been trained to guess what the sentences in his textbook mean exactly, the compiler can be trained to guess the meaning of what is said in the superimposed language. It cannot be expected to do very much in this direction,

but whatever it can do, will be very helpful. Writing absolutely meticuously is very much harder than writing almost meticuously, and it will be a great gain if a machine can bridge the gap between the two.

11. AUTOMATIC THEOREM PROVING

11.1. Automath is *not* intended for automatic theorem proving. Theorem proving is a difficult and time-consuming thing for a machine. Therefore it is almost imperative to devise a special representation of mathematical thinking for any special kind of problem. Using a general purpose language like Automath would be like using a contraption that is able to catch flies as well as elephants and submarines.

11.2. There is a case for automatic proof writing in Automath if we have to produce a tedious long proof along lines that can be precisely described beforehand. Let us take an example. Assume that P is a proposition on magic squares, and that we want to prove a theorem saying that there is no 8×8 magic square that has property P. We can write a computer program for this and run it on a computer. The computer says that none exist. Now quite apart from the question whether the computer is right, we have to admit that a formal mathematical proof has not been produced. Even if we had a complete mathematical theory about the machine, the machine language, the programming language, our proof would depend on intuitive feelings that the program gives us what we want, and it would definitely depend on a particular piece of hardware.

For those who are willing to take Automath, at least temporarily, as their only final conscience of mathematical rigour, there is a way out. We can rewrite the magic square program in such a way that the search is stepwise accompanied by the production of Automath lines that give account of a detailed mathematical reasoning, ending with the conclusion that there is no 8×8 magic square with property P. This way we get a complete proof that can be checked by any mathematician. If we leave the checking to a computer, we get again into the question of whether the processor and the computer do what we expect them to do, but that is an entirely different matter.

12. EXTENSIONS OF AUTOMATH

12.1. If we feel we should have a more powerful language than Automath, this can have two reasons.

12.2. One reason is that we feel that the language is clumsy, and that we want to make it more handy, without changing the scope of what we can say. For some purposes this might be possible by extension of the language, i.e. by adding new grammar rules without cancelling the old ones. It is hardly necessary to consider such extensions for the present purpose, since it can be expected that the same goal can be reached by means of superimposed languages. We might think about facilities for easy identification of two things of different categories (see Sec. 2.2), embedding of one category into another, etc. If such matters can be handled satisfactorily, they can be handled by a superimposed language. The only reasons for doing it without such a language may be computer time and memory space.

12.3. A different reason for extension can be that we feel that Automath is not strong enough, just as we extended PAL to Automath since PAL was not strong enough for modern mathematics.

One might suspect that no single language will ever be entirely satisfactory. It is an old mathematical habit to mix language and metalanguage: we write a text in a language; we discover facts about that text; we use these facts in the subsequent text. This of course means an extention of the language. We mention an example, though not a very important one. Let q be any identifier in an Automath book, and let p be a block opener. If it happens that q does not implicitly depend on p, this is an observation *about* the book, and there seems to be no way to write it as an assertion *in* the book. It will be an extension of the language if we design some way to write this independence, a way to derive it from the book, and a way to use that written information if we need it. This kind of thing is done in ordinary mathematical language, but in Automath it is not necessary. If q does not depend on p, then we are able to define $r := q$ in a context where p is not valid, and then we need not bother about p any more.

12.4. There is a class of extensions of Automath that is very easy to describe: We start the book with a number of lines some of which have not been written according to the rules; we want to write the rest of the lines in the book according to the rules. We give an example that does not belong to Automath, but to the language we get from Automath if PN's are forbidden: Then we can write all axioms in the basis as theorems without proofs, and talk PN-free language ever after.

One might even think of an infinitely long basis. For example, one might like to have all the natural numbers as *a priori* given, and devote a line or two to each one of them.

12.5. In Automath we have the right to indulge in functional abstraction with

respect to every type. In private discussions Prof. Dana Scott said he did not like the idea of introducing "*bool*" as such a type, at least not in intuitionism. It is very easy to extend Automath by introducing a symbol **type***, and saying that if Σ has category **type***, then we do not have the right of functional abstraction with respect to Σ. It seems fair to admit the category $\Sigma_3 := [x : \Sigma_1]\,\Sigma_2$ if Σ_1 has category **type** and Σ_2 has category **type***, and to say that Σ_3 has category **type***. If we do all this, we can introduce "*bool*" as something of category **type***, and "*nat*" (the natural numbers) as something of category **type**.

12.6. In Automath we did not allow functional abstraction with respect to **type** itself. For example, if we have

0	ξ	:=	—	**type**
ξ	b	:=	PN	*bool*

then we can *not* write

$$0 \quad \ldots \quad := \quad [t : \mathtt{type}]\, b(t) \qquad [t : \mathtt{type}]\, bool\ .$$

It is difficult to see what happens if we admit this.

12.7. A possibility that seems less dangerous than the one of 12.6 is the following one: if we have

0	ξ	:=	....	**type**
0	a	:=	—	ξ
a	b	:=	PN	**type**

then we allow to write

$$0 \quad \ldots \quad := \quad [t : \xi]\, b(t) \qquad [t : \xi]\, \mathtt{type}$$

This gives more information about $[t : \xi]\, b(t)$ than just saying that it has category **type**, but on the other hand it puts an end to uniqueness of category.

Moreover, we permit lines such as

$$0 \quad a \quad := \quad — \qquad [t : \xi]\, \mathtt{type}$$

in order to introduce an arbitrary way of attaching a type to each t in ξ.

Once we have opened these possibilities, it will be pretty obvious what the further operational rules have to be.

We mention a single case where this extension of our language is needed. In connection with recursive definitions, we might wish to say: let $P_1, P_2, \ldots$ be an infinite sequence of categories. This can be done by means of a block opener with category $[n : nat]$ **type**.