UNIVERSITÉ DE MONTRÉAL — DÉPARTEMENT DE MATHÉMATIQUES

AUTOMATH
A LANGUAGE FOR MATHEMATICS

par

Nicolaas Govert de BRUIJN

Technological University, Eindhoven, Pays-Bas

1973

LES PRESSES DE L'UNIVERSITÉ DE MONTRÉAL

C.P. 6128, MONTRÉAL 101 CANADA

CONTENTS

	page
Introduction	6
Informal introduction to LSP	14
PAL - preliminary orientation	24
Definition of PAL	26
How to use PAL for mathematical reasoning	29
Metalanguage	35
The Lambda Calculus	38
Description of AUTOMATH	41
Extensions of AUT	52
Bibliography	56

Notes taken by Barry Fawcett.

Notes du cours donné par le professeur Nicolaas G. de Bruijn à la dixième session du Séminaire de mathématiques supérieures de l'Université de Montréal, tenue l'été 1971. Le Séminaire est placé sous les auspices de la Société Mathématique du Canada et du Gouvernement du Québec.

We have to start with an apology for treating AUTOMATH in a seminar devoted to combinatorics. First, AUTOMATH and related languages were devised as a serious attempt to bridge the credibility gap engendred by long and excessively detailed proofs in mathematics ; this need is felt strongly in particular in combinatorics. Secondly, the study of these languages itself has many combinatorial aspects but is also related to various other fields ; logic (in particular combinatorial logic), foundations, the philosophy of mathematics, the history of mathematics, mathematics education, the unification of mathematical disciplines and computer science.

The speaker's interest in these matters began during an attempt to clean his desk. Among a sheaf of reprints was discovered a paper which consisted of only one page, and which could not, because of its brevity, be ignored in good conscience. The topic of the paper was multiprogramming in computer systems, with an exposition of certain procedures which were claimed to make a system reach certain goals. Following a somewhat intuitive argument it was remarked, with unusual and possibly unintended candour, that "this, the author believes, completes the proof". The speaker, in attempting to prove a correct theorem, encountered horrendous details. Here was a piece of mathematics which was difficult to write up correctly and also very difficult to verify with confidence. Thus there is a need for a language which will allow machines to check whether such complicated theorems are stated and proved correctly. The attainment of such a language that will em-

brace the body of current mathematics in an easy fashion is still a distant goal. Some branches of mathematics, e.g. algebra, set theory, can be formalized easily. However, problems which are expressed in intuitive language, e.g. the celebrated problem of covering a truncated chessboard with dominoes, are more difficult to formalize, at least if we start from the original geometrical formulation. The study of these languages is a branch of combinatorial logic. As such, it is a piece of combinatorics, despite the fact that the methods appropriate for discussion of this topic are not common combinatorics. We shall be concerned with the following group of languages.

LSP

SEMIPAL

PAL*

AUTOMATH (1968)

AUT-QE (1969)

The languages, arranged in ascending order of complexity, are not programming languages. That is, they do not express sets of machine instructions. They are simply schemes which ensure that everything that is stated correctly is mathematically correct.

To begin informally, let us consider a book that has been written line by line. The usual approach is to posit a pool of assumptions about foundations, and then to test the statements of the book

---

* PAL must not be confused with the programming language which shares the same acronym.

against the eternal values of the pool. Here, however, we shall adopt the novel approach that no foundational system is given in advance, but that one is written into the text line-by-line. The problem of interpretation that arises will be dealt with later.

As for the usual confusion between language and metalanguage, let us say, in the absence of a formal definition, that a language is, roughly, a system for writing books. When we step back and talk about books that have been, or may be, written, a metalanguage must be used. Often a language is extended by incorporation of metalingual features. The jump from PAL to AUTOMATH described below is such an extension.

Mathematical languages or books may have various interpretations. For example, A.A. Morse's Theory of Sets ([22]) has theorems which can be interpretated both as logical and as set-theoretical theorems. Our aim is to define a language formally in such a way that a computer can check whether a text is written according to the rules of that language. The computer cannot take responsibility for the various interpretations of the text, but we ourselves will be concerned with interpretations for the sake of the application.

Some aspects of mathematical proofs, like hand-waving and rhetorical devices, are impossible to formalize. These techniques cannot convince a machine of the validity of a proof unless they can be expressed in a formal language. Although we should like to believe that all mathematics can be presented formally, we ought to realize that mathematics is a social affair, and related to the outside world.

Vague communication is sometimes to be preferred to formalization. Formalization is not everything, yet it is certainly something.

Historically, the idea of a fixed formal language dates from the Vienna (until 1933) Circle of Positivists. Wittgenstein said "Don't ask for the meaning, ask for the use", or to paraphrase, "ignore interpretations". Interest in formal languages increased with the advent of computer languages. We were suddenly made aware of the real requirements. Computers refused our hand-waving and eternal language-changing. Formalization beyond the stage of foundations became indispensable.

What are the implications for pedagogy ? Some say our teaching is suited for those who already know the ropes. Some confused students just copy the mysterious statements (they are often called "gifted"). Is it possible that we just lack a suitable language ? Let us go as far as possible and make something like AUTOMATH accessible to fourteen-year olds. For at present, we teach by intimidation and learn by imitation. The mysteries are repeated in a raised voice. At last the student gives in. But computers are immune to such treatment.

The student might ask why mathematics is cast into the DEFINITION-THEOREM-PROOF recipe. Is this tradition ? Or is it essential ? Clearly a language is needed to tackle this question. One positive aspect of PAL and AUTOMATH is the exposition of the structure underlying that recipe.

Our endeavour is useful for proof-checking and for avoiding the danger of incorrect usage of otherwise correct theories. The pro-

blems of teamwork, man-machine cooperation, and program checking can be brought up. Today, computer scientists need efficient techniques to check the validity of programs (i.e.,to check whether the program achieves what the programmer claims). Hopefully, a large part of this may be done by machines. As it is now, it may happen that inferior programs written by not-so-clever programmers are checked by clever experts. To avoid the waste of brainpower, it might be better that these experts write both the program and a set of hints by which a computer can check the validity of the program.

The study of AUTOMATH has led to new visions of the basis of mathematics, including an analysis of its structure and suggestions as to how it might be changed. Sometimes it may be revealed whether a process has intrinsic or only historical justification. There exists a popular belief, not shared by the speaker, that a satisfactory basis for mathematics is difficult or impossible to attain. Whitehead and Russells' Principia had a negative effect on this endeavour (because it is too complex). The issue was complicated by Cantor's Paradise (which is essentially a mixing of language and metalanguage). Cantor invented the alephs ; later mathematicians considered and adopted them as familiar constructions. In the twentieth century it is widely believed that Cantor's Paradise exists and that it is important. In the twenty-first century, mathematicians may think it better to leave Cantor's Paradise. They may get the idea that the paradise is a poor basis for mathematics, no matter how beautiful it may be.

The use of these languages may lead to new discoveries, perhaps by analogy and inspection. Historically, new languages preceded major developments in mathematics. Although AUTOMATH has not yet given rise to interesting new theorems, the future portends such developments as a general mathematical library (DIAL-A-THEOREM ?), improved publication standards and higher levels of man-machine interaction.

Certainly the entire language of mathematics will change in the future. The Greeks in antiquity studied geometrical figures. Geometrical figures, however, constitute only a rough approximation to a mathematical language. Van der Waerden ([26]) remarked that despite the obvious ingenuity of the Greeks, their notation for the integers precluded further developments (they ran out of letters). The Arabs' tremendous achievement was the introduction of letter variables. When modern algebraic notation came to Europe, Descartes and Fermat were able to "formalize" geometry in this new language. Even the development of decimal notation already facilitated the communication of a fragment of mathematics. The story of the Greeks' failure caused by lack of language might contain a moral. Leibniz's dream of a universal scientific language in which thinking is replaced by calculation is an extension of Descartes' idea.

Boole was influenced by a similar idea. He devised a language for a part of logic, but his efforts had no relation to the standard mathematics of his contemporaries. The very extensive language of Whitehead and Russell which combined logic and mathematics, and the

formidable nature of its presentation, has been mentioned previously. Peano wrote a detailed formal encyclopedia of mathematics which remains today as a period piece ; with all its details it was far from being accessible to something like mechanical checking.

In our presentation, comments may be desirable for flavour, references and physical implications, but are unnecessary for checking the text. This reflects the stimulus of computer technology. Computers are extremely stupid, and don't understand what they do not hear. But they are dependable and fast.

L.E.J. Brouwer, Kronecker and H. Weyl rejected formalism. Many paradoxes emerged from Cantor's Paradise, some involving mixing of language and metalanguage, and some, like Russell's paradox, necessitating new supports for a falling building. There is no guarantee against the appearance of new antinomies. Brouwer opted for constructivity and fought his formalistic contemporaries, but his writings give us the impression that he lacked a suitable language for his criticism. Brouwer's intuitionism seemed to cause many technicalities and therefore did not get much support. Recently, E. Bishop ([5]) has revived Brouwer's ideas. Rather than concentrating on complicated intuitionistic counter-examples, he thinks his task is simply to prove ordinary things.

Since 1960, John McCarthy, J.A. Robinson ([25]), J.B. Rosser, Hao Wang and others have studied automatic theorem proving. Their algorithms lead to long proofs which do not generally correspond to proofs

a mathematician would give. This is different with languages like AUTOMATH, which are, on the other hand, not effective for automatic theory proving. In [4] appears a mathematical language based on the programming LISP. A proof is a LISP procedure and some forms of substitution are permitted. The languages VAT'68 and VAT'70 are discussed in [1], [2] and [3]. The language AUTOMATH (or AUT) was developped by mathematicians at Eindhoven, (see : [6], [7], [11], [12], [14], [15], [20], [21] and [23]). AUT may seem to be the worst of all as far as actual writing is concerned, but it seems to be the best in terms of flexibility and general applicability.

We expect that a useful language will dependably transmit ordinary mathematics by means of formulae which are not excessively long. Reliable methods of extending languages, and of incorporating auxiliary languages are required. We expect that a machine can check a book written in the formal language and decide whether it is correct. Compare this situation with that of a machine which verifies whether a chess game has been conducted according to the rules. There is a formal language for chess which employs symbols like 1. e2 - e4, e7 - e5 2. resigns, etc., and there is a material interpretation : pieces being moved on a board. Other items must be considered, such as whether castling is still possible or whether a pawn may be captured en passant. A more complicated feature is the application of the rules about a drawn game.

What did we achieve ? An AUTOMATH checker is at present in operation at Eindhoven. An AUT text can be typed in line-by-line at a terminal : the computer signals either "WRONG", together with a diagnosis, or "CORRECT". L.S. Jutting has developped considerable experience with the proofchecker. At present, Landau's booklet on the foundations of analysis is being translated into AUT. Although the booklet is very detailed, the translator must fill several gaps. Eventually, the machine will certainly be able to read Landau's book in something like an hour.

A novel aspect of AUT is the presentation of proofs as constructions. Construction of objects and construction of proofs are treated in like manner. This sort of mixture occurs naturally in the writing of ordinary mathematical texts. For example, ruler and compass constructions in synthetic geometry usually involve the following types of construction: 1) definitions of geometrical objects in terms of others, 2) descriptions of geometrical constructions with the ruler and compass, 3) construction of propositions and predicates,and 4) the construction of proofs. If in a geometrical construction we connect two points by a straight line, we must first construct a proof that they are distinct. If we assert that a constructed point is the barycentre of a triangle, then a definition of that notion ought to precede.

Informal Introduction to LSP

LSP has an alphabet consisting of two kinds of symbols,

(1) constants a,b,c,d,...

(2) variables s,t,u,v,...

An extra symbol, PN ("primitive notion") and the signs ( ) , :=

Example of an LSP book (Interpretation : the definition of a function)

a(x,y)	:=	PN
b(t)	:=	a(t,a(t,t))
c(x,u)	:=	b(a(u,b(x)))
d(x,t)	:=	b(c(x,a(t,t)))
e(u,x,v)	:=	PN
f(x,t)	:=	d(e(x,t,x),e(x,x,b(x)))
g	:=	PN
h(x)	:=	f(g,g) .

Observe that every line has its own identifier on the left, and a (possible empty) sequence of distinct variables attached to it. On the right occurs either the symbol PN or an expression. All variables occuring in an expression on the right also occur on the left, but not necessarily conversely. Every constant on the right has occurred in a previous line. Every constant on the right is the head of an expression with as many subexpressions as the number of variables associated with that constant at the line where it was introduced. This number is called the length of the constant. In the interpretation, we say that the left hand side of a line is defined by the right-hand side (unless the right-hand side is PN).

The notion "correct LSP" book is defined recursively. First the empty book is correct. Assuming that we have a correct book B , we define an admissible expression with respect to that book as follows :

1) each variable is admissible ;

2) constants of length zero are admissible ;

3) a constant of positive length k is admissible when it is

followed by a string of k admissible expressions separated by commas, and surrounded by parentheses. In both cases, "constant" means "constant that has appeared on the left-hand side in the book".

Assuming that the book B is correct, a correct one-line extension of B is written by choosing any new constant together with any sequence of variables followed by := . On the right, we write any expression admissible with respect to the old book, provided that only variables which appear on the left are used. Or we may write PN with no condition. Let us consider an example with an interpretation from Boolean logic.

```
non(x)              :=   PN
impl(x,y)           :=   PN
notboth(x,y)        :=   impl(x,non(y))
and(x,y)            :=   non(notboth(x,y))
or(x,y)             :=   notboth(non(x),non(y))
atleastone(x,y,z)   :=   or(x,or(y,z))
allthree(x,y,z)     :=   and(x,and(y,z))
```

The notion of definitional equivalence (DEFEQUAL), is defined by means of the operator DEF which applies to admissible expressions. The head of an expression is the first symbol which appears. A constant is primitive if it is defined by PN . DEF is undefined if the head is primitive or a variable. The application of the operator DEF will be illustrated by the first example of an LSP text, line 6. To calculate

DEF(d(e(x,t,x), e(x,x,b(x)))

first take the head (= d), look up its definition,

d(x,t) = b(c(x,a(t,t))) ,

substitute

$$\begin{cases} \underline{x} = e(x,t,x) \\ \underline{t} = e(x,x,b(x)) \end{cases}$$

applied to the $\underline{x}$ and $\underline{t}$ that are underlined and not to those appearing on the right in the substitution. The result is

$$b(c(e(x,t,x),a(e(x,x,b(x)),e(x,x,b(x))))) \ .$$

Now the notion DEFEQUAL (notation $\overset{D}{=}$), with respect to a book, is defined recursively. We shall write $\Sigma_1 \overset{D}{=} \Sigma_2$ (the Σ's are metalingual symbols). We require

(1) if $\Sigma_1 =$ DEF Σ_2, then $\Sigma_1 \overset{D}{=} \Sigma_2$;

(2) if Σ and Σ' have the same head, with subexpressions $\Sigma_1, \Sigma_2, \ldots, \Sigma_r$ and $\Sigma'_1, \Sigma'_2, \ldots, \Sigma'_r$, respectively, then if $\Sigma_i \overset{D}{=} \Sigma'_i$ then $\Sigma \overset{D}{=} \Sigma'$.

(3) DEFEQUAL is reflexive, symmetric and transitive.

An expression Σ is called <u>normal</u> if all its constants are primitive. (For example, $a(t,a(t,t))$.) Now we state without proof:

<u>THEOREM 1</u>

If Σ is admissible with respect to a correct LSP book, then there is exactly one normal expression (called the <u>normal</u> form of Σ and written $NF(\Sigma)$) such that $\Sigma \overset{D}{=} NF(\Sigma)$.

Roughly speaking, we obtain the normal form if we continue to apply the operator DEF until all non-primitive constants are gone. In our example,

NF(c(x,u)) = a(a(u,a(x,a(x,x))), a(a(u,a(x,a(x,x))), a(u,a(x,a(x,x))))).

Note that this example was taken from the very beginning of our book; it can be expected that it gets much worse later on, so that normal forms have theoretical interest only.

<u>THEOREM 2</u>

$$\Sigma_1 \overset{D}{=} \Sigma_2 \quad \text{iff} \quad NF(\Sigma_1) = NF(\Sigma_2) \quad .$$

Let us remark that the notions of DEF, NF and DEFEQUAL play no role in the definition of language, nor in testing the language. A computer programme which checks LSP books is easy to write. Given a correct book B and two expressions, Σ_1 and Σ_2, THEOREM 2 produces an algorithm to determine whether they are admissible and definitionally equivalent. The head of NF(Σ) is easy to find by applying DEF. The structure of an LSP book is that of a directed graph without loops. The exact linear order of the lines is immaterial. To decide whether $\Sigma_1 \overset{D}{=} \Sigma_2$, take the expression with the younger head, apply DEF and look again. If the heads are equal, check whether the subexpressions are definitionally equivalent. CAUTION: One must check whether an expression does really depend on all its variables. Possibly some of them might be <u>inactive</u> (i.e., they do not occur in the normal form of the expression). It is not difficult to keep a list of active variables from line to line. It would be better to write texts without inactive variables; in practice they do not often occur.

SEMIPAL arises from LSP by declaring variables, context indication and abbreviations. With the new symbol: x:= ——— called a block opener, we declare the variable x . No longer are lists of variables written on the left. Declared variables form together a limited context. A set of nested blocks indicates the sequence of variables to be employed. For clear exposition, the block structure of a SEMIPAL book is indicated by vertical bars. Machines (and typists) abhor vertical lines, preferring horizontal indicator strings. The context indicator of a line is the last previously declared variable or 0 (an extra symbol of SEMIPAL) in case there are none.

Consider the example:

Indicator String	Context Indicator	
Empty	0	x := ———
x	x	y := ———
x,y	y	a := PN
x,y	y	z := ———
x,y,z	z	b := a(x,z)
x,y,z	z	c := b(z,z,y)
x,y	y	d := c(x,y,c(x,x,x))
x	x	e := d(x,x,x)
Empty	0	f := PN .

The translation to LSP is straight forward.

```
a(x,y)        :=  PN
b(x,y,z)      :=  a(x,z)
c(x,y,z)      :=  b(z,z,y)
d(x,y)        :=  c(x,y,c(x,x,x))
e(x)          :=  d(x,x,x)
f             :=  PN  .
```

In translating an LSP text into SEMIPAL it might become necessary to alter the variables first. Note that the block structure and indicator strings convey identical information; one may be retrieved from knowledge of the other. Let us remark that it is permitted to re-open an old block.

An example of an LSP book that cannot be translated immediately into SEMIPAL is easily constructed, viz.:

```
a(x,y)        :=  PN
b(x,y)        :=  a(y,x)
c(x,z,y)      :=  a(a(z,x),y) .
```

The names of the variables must be altered so that, if two strings of variables occurring to the left of := have variables in common, then these common variables should form an identical initial segment of both strings. (For example: x,y,z,w and x,y,p,q, are compatible, but not x,y,z,w and x,y,p,q,w). The condition that a variable may not be declared twice seems impractical for writing mathematics (although computers disagree). With a certain amount of care, conventions may be struck to govern the re-use of old variables. Yet it is better for theoretical purposes to avoid this.

In SEMIPAL an expression like $b(\Sigma_1,\ldots,\Sigma_k)$ can be abbreviated: if $x_1,\ldots,x_k$ is the indicator string of b; and if $\Sigma_1 = x_1$, $\Sigma_2 = x_2,\ldots,\Sigma_j = x_j$, then we may write $b(\Sigma_{j+1},\ldots,\Sigma_k)$, or just b if $j = k$. That is, if in the abbreviated form there are not enough subexpressions, they are completed at the front by adding the beginning of the indicator string of b. A direct definition of a B-acceptable abbreviated expression is possible but this would complicate somewhat the addition of a line to the book.

The notions $\overset{D}{=}$ and NF were defined by means of the unabbreviated expressions. There is an interpretation in terms of the abbreviated forms but that is hardly necessary. An example of an abbreviated book follows. It is the same example from Boolean logic given previously. Note that in this form it is not very readable.

0	x	:=	———
x	non	:=	PN
x	y	:=	———
y	impl	:=	PN
y	notboth	:=	impl(non(y))
y	and	:=	non(notboth)
y	or	:=	notboth(non,non(y))
y	z	:=	———
z	atleastone	:=	or(or(y,z))
z	allthree	:=	and(and(y,z)) .

It seems overdone to devote a special line to each "block opener". It does stress the fact that <u>all</u> identifiers are distinct. The extra lines

will have a greater significance in the language PAL. One advantage of the outlined way of context indication and abbreviation is that quite often a large number of lines in a book will depend upon a fixed number of variables. These need not be repeated in every line. The block structure allows us to introduce "local" constants which appear only in a limited context. This feature parallels the similar practice in everyday mathematical writing.

A block is not necessarily a connected piece of information. Lines which are written in an old context may be added at a later stage, old variables may be revived, and old blocks re-opened. In our examples, the blocks consisted of sets of consecutive lines. This occurred merely to sharpen the exposition.

In speaking about a SEMIPAL book we use metalingual terminology like, "something is written in context y", or, "the indicator string in context y (the empty string if y = 0)", or, "in context y the only live variables are those of the indicator string". This terminology, and its metalingual character are conspicuous. Thus the indicator string in context x is a string $x_1, x_2, \ldots, x_k$ where $x_k = x$ and x_i = indicator of x_{i+1}.

Substitution

Let $x_1, x_2, \ldots, x_k$ be distinct variables and $A_1, A_2, \ldots, A_k$, expressions. The symbol $S_{x_1 \to A_1,\ x_2 \to A_2, \ldots, x_k \to A_k}\ \Gamma$ denotes the result of substituting simultaneously

$$\left\{\begin{array}{l} x_1 \to A_1 \\ x_2 \to A_2 \\ \vdots \\ x_k \to A_k \end{array}\right.$$

in Γ (i.e. replace each x_i in Γ by the metalingual symbol Λ_i, then replace each Λ_i by the expression for which it stands).

Formally speaking, a (metalingual) notation to distinguish objects from their names is required. The customary practice of placing the name of an object in quotation marks is ill-advised, since the name ought to determine the object and not conversely. Our convention shall be to underline the object to distinguish it from its name. For example, we do not write

Montreal is very clean

"Montreal" has eight letters

but

Montreal is very clean

Montreal has eight letters.

Accordingly, our substitution recipe should read:

$$S_{x_1 \to \underline{\Lambda}_1,\ x_2 \to \underline{\Lambda}_2,\ \ldots,\ x_k \to \underline{\Lambda}_k} \quad .$$

However, we shall not apply this notation in these notes.

THEOREM 3

If $x_1, x_2, \ldots, x_k$ are distinct variables and if $\Lambda_1, \Lambda_2, \ldots, \Lambda_k$, are acceptable expressions with respect to a correct LSP book B, then

$$NF(S_{x_1 \to \Lambda_1, \ldots,\ x_k \to \Lambda_k}\ \Gamma) = S_{x_1 \to NF(\Lambda_1), \ldots, x_k \to NF(\Lambda_k)}\ NF(\Gamma) \quad .$$

The proof is omitted.

PAL - Preliminary Orientation

To capture the essence of statements like

$$\begin{cases} x \text{ is a point} \\ y \text{ is a line} \\ \text{the distance between } x \text{ and } y \text{ is a real number} \end{cases}$$

we should be able to write lines of the form

```
x    :=  ------- point
y    :=  ------- line
dist :=  ..... real    .
```

That is : we should be able to attach categories to the objects we are discussing.

PAL is an extension of SEMIPAL which allows for attatching categories. When substituting expressions, for example writing $\mathrm{dist}(\Sigma_1,\Sigma_2)$, we shall require that the expressions Σ_1 and Σ_2 be of the proper category. A category is attached to every expression. This is a restriction on the expressions we can admit. Moreover,we should like to be able to introduce new categories, possibly as a function of some variables. To expedite this, we introduce the symbol : <u>type</u>. PAL is adequate for expressing elementary geometry and first order logic (without functional abstraction). Consider two examples of a piece of a PAL book. The first, a translation of Hilbert's axioms for geometry does not get very far. The second has an interpretation as the definition of a cartesian product.

<u>Example</u>

```
point  :=  PN     type
line   :=  PN     type
x      :=  ------- point
y      :=  ------- line
```

Example

θ	:=	———	type
η	:=	———	type
prod	:=	PN	type
z	:=	———	prod
$proj_1$	:=	PN	θ
$proj_2$	:=	PN	η
x	:=	———	θ
y	:=	———	η
loc	:=	PN	prod
number	:=	PN	type
cartplane	:=	prod(number, number)	type
a	:=	Σ_1	number
b	:=	Σ_2	number
c	:=	$loc(\Sigma_1,\Sigma_2)$	cartplane .

Here again, Σ_1 and Σ_2 are metalingual symbols denoting certain expressions.

Among the expressions in PAL we make the following (metalingual) distinctions :

a 1-expression is the symbol type ,

a 2-expression is the name of a category (e.g. point, line) ;

a 3-expression is the name of an object (e.g. sum(prod(x,y)z)) .

Next we define a mapping CAT which associates with every 3-expression its category, and which associates with every 2-expression the symbol type ; CAT(type) is undefined. Thus CAT maps 3-expressions into 2-expressions, and 2-expressions into 1-expressions.

A PAL book shall have the form of a SEMIPAL book with an extra column. If B is a correct PAL book, then B* , the book obtained by omitting the last column, is a correct SEMIPAL book. In the last column the entry is either an expression or the symbol type. If B is a correct PAL book, then it remains correct if the last line is deleted. The notions D and NF will be interpreted with respect to B* .

The definition of the notion "correct PAL book" shall be arranged along similar lines as the definition of the notion "correct LSP book". Namely :

1) The empty book is correct ;

2) If B is a correct PAL book, then it is indicated how admissible B-expressions are built. Once a new expression has been constructed, its category is defined ;

3) How B may be extended by one line is described.

DEFINITION OF PAL

1) The empty book is correct ;

2) Let B be a correct book, and θ one of its variables or θ . We define a B-admissible expression Σ and its category CAT(Σ) recursively:

i) type is admissible

ii) if x occurs in the indicator string at θ , then x is admissible and CAT(x) is the entry in the category column at x (i.e. the expression Γ when x is defined in the line

x:= ———— Γ

iii) let b be a constant of B with indicator string $x_1,\ldots,x_k$ and categories $\Gamma_1,\ldots,\Gamma_k$ respectively; that is,

$$
\begin{array}{|l|lll}
 & x_1 & := \text{———} & \Gamma_1 \\
 & \vdots & & \\
 & x_k & := \text{———} & \Gamma_k \\
\ldots & b & := \Omega & \Sigma
\end{array}
$$

are (not necessarily adjacent) lines of B.

Let $\Lambda_1,\ldots,\Lambda_k$ be B-admissible expressions and all different from <u>type</u>, so that $CAT(\Lambda_1),\ldots,CAT(\Lambda_k)$ are defined. Then $b(\Lambda_1,\ldots,\Lambda_k)$ is admissible provided that

$$
\begin{aligned}
&CAT(\Lambda_1) \overset{D}{=} \Gamma_1 \\
&CAT(\Lambda_2) \overset{D}{=} S_{x_1 \to \Lambda_1}\, \Gamma_1 \\
&\quad\vdots \\
&CAT(\Lambda_k) \overset{D}{=} S_{x_1 \to \Lambda_1,\ldots,x_{k-1} \to \Lambda_{k-1}}\, \Gamma_k\ .
\end{aligned}
$$

In that case $CAT(b(\Lambda_1,\ldots,\Lambda_k))$ is defined as

$$S_{x_1 \to \Lambda_1,\ldots,x_k \to \Lambda_k}\, CAT(\Omega),$$

or, if Ω is PN then $S_{x_1 \to \Lambda_1,\ldots,x_k \to \Lambda_k}\, \Sigma$.

3) If B is a correct book and θ either 0 or a variable of B, then each of the following lines in context θ provides a correct extension of B.

$$
\begin{array}{lll}
\eta & := \text{———} & \Sigma_1 \\
\sigma & := PN & \Sigma_1 \\
\tau & := \Sigma_2 & \Sigma_3
\end{array}
$$

η, σ and τ are new identifiers, Σ_1 is <u>type</u> or a B-admissible 2-expression, Σ_2 is a B-admissible 2-expression or 3-expression and $CAT(\Sigma_2) \overset{D}{=} \Sigma_3$.

In consequence, if B is a correct PAL book and A is a B-admissible expression different from <u>type</u>, then

a) A is B*-admissible ;

b) $CAT(A)$ is B-admissible (and B*-admissible) ;

c) $NF(CAT(A)) = NF(CAT(NF(A)))$.

Now we may check the text below by means of the definition of PAL.

θ	θ	:= ———	<u>type</u>
θ	x	:= ———	θ
x	y	:= ———	θ
y	EQ	:= PN	<u>type</u>
x	I	:= PN	EQ(x,x)
y	u	:= ———	EQ(x,y)
u	z	:= ———	θ
z	v	:= ———	EQ(z,y)
v	II	:= PN	EQ(z,x)
u	III	:= II(y,I(y))	EQ(y,x)
z	w	:= ———	EQ(y,z)
w	IV	:= II(z,y,III(y,z,w),x,u)	EQ(x,z)

Let us remark that a PAL book becomes a SEMIPAL book by cancelling the last column. A SEMIPAL book becomes a PAL book by adding a first line θ := PN <u>type</u> , and then adding a last column with θ as the only entry, or simply by adding a last column with <u>type</u> as the only entry. Also, let us notice that we maintain our abbreviation facilities. That is :

Suppose a constant C is introduced in context $x_1,\ldots,x_\ell$. Suppose in some context θ we use $C(\Sigma_k,\ldots,\Sigma_\ell)$, i.e. we use C with less

than ℓ subexpressions. Then $C(\Sigma_k,\ldots,\Sigma_\ell)$ should be regarded as $C(x_1,\ldots,x_{k-1},\Sigma_k,\ldots,\Sigma_\ell)$; we of course require that $x_1,\ldots,x_{k-1}$ is a substring of the indicator-string θ .

How to use PAL for mathematical reasoning (see [6], p. 15)

So far our concern has been merely to express things by means of LSP, SEMIPAL and PAL. However, mathematicians are usually more interested in how to prove theorems, rather than expressing things. Mathematics has the same block structure as PAL, but there are two ways to open a block. One is by introducing a variable that will have meaning throughout the block; the other is by making an assumption that is valid throughout the block. The second case shall be dealt with by representing statements by categories. Constructing an object with that category means asserting the statement. This may be done by means of ——— , PN or an expression, corresponding to assertion by assumption, an axiom, or proof, respectively. Thus an assertion may be a line of the form

$$* \ldots * := \Lambda \quad \Sigma$$

Λ is called a proof for Σ , and the category Σ may be thought of as the class of all its proofs. Recall the text on page 23. As an example of a theorem and its proof, let us write from the text, in the notation of Lindenbaum, the statements :

$$\text{I} \quad x \sim x$$

$$\text{II} \quad \frac{x \sim y,\ z \sim y}{z \sim y} \quad .$$

If we wish to derive the statements,

$$\text{III} \quad \frac{x \sim y}{y \sim x}$$

and

$$\text{IV} \quad \frac{x \sim y,\ y \sim z}{x \sim z}$$

we may do so by the following argument. To prove III, first derive $y \sim y$ from I. Next use II with $z \to y$ to derive $y \sim x$. To prove IV replace $x \to z$, $z \to x$ in II to obtain $\frac{x \sim y,\ z \sim y}{x \sim z}$. Then obtain $z \sim y$ from $y \sim z$ by III (with $x \to y$, $y \to z$).

In a format suggestive of PAL we arrange the proof as:

I(x) is a (primitive) proof for $x \sim x$.

II(x,y,proof $x \sim y$, z, proof $z \sim y$) is a (primitive) proof for $x \sim x$.

III(x,y,proof $x \sim y$) := II(x,y, proof $x \sim y$, y, I(y)) is a proof for $y \sim x$.

IV(x,y, proof $x \sim y$, z, proof $y \sim z$) := II(z,y, III(y,z, proof $x \sim y$), x, proof $x \sim y$) is a proof for $x \sim z$.

Now this translates into PAL automatically. We add the type θ everything is about ; but since that is not substituted into the text, no difficulties are presented.

An <u>assertion</u> seems to be a more natural notion than that of a <u>proposition</u> (which may or may not be true). However, propositions (or <u>booleans</u>) may be introduced into PAL by admitting the category "bool", consisting of all propositions. The book may begin as follows :

```
0        bool  :  PN        type
0        b     := ———————   bool
b        TRUE  := PN        type
```

If in a certain context appears a line like :

```
...  := ......  TRUE(a)
```

where a is a boolean (in that context), then the interpretation in usual mathematical terms is that a is asserted.

Modus Ponens presented in this format (see [6], p.18) becomes

```
0                bool    := PN        type
0                b       := ———————   bool
b                TRUE    := PN        type
0                x       := ———————   bool
x                y       := ———————   bool
y                impl    := PN        bool
y                asp 4   := ———————   TRUE(x)
asp 4            asp 5   := ———————   TRUE(impl(x,y))
asp 5            modpon  := PN        TRUE(y)
```

To use such a piece of text, suppose that we have lines like :

```
ℓ1   := A     bool
ℓ2   := Γ     bool
ℓ3   := ...   TRUE(impl(A,Γ))
ℓ4   := ...   TRUE(A)
then we may add
ℓ5   := modpon(A,Γ,ℓ4 ℓ3)     TRUE(Γ)   .
```

The derivation rules, evidently, are just theorems. New rules can be devised and used. Note that the text is not subdivided into parts along the usual definition-theorem-proof model. Every line is a result that may be used whenever we wish. A theorem is never announced prior to its proof ; a result cannot be stated until it is derived.

In the next example we define something of the form : "if ..., then ..., else ...". Some assumptions are needed ; the first requirement is that axioms for equality are given before (this is an awkward feature). Also, negation should be defined beforehand.

θ	:= ———	<u>type</u>
x_1	:= ———	θ
x_2	:= ———	θ
IS	:= PN	<u>type</u>
[axioms for IS , for example, Lindenbaum's]		
condition	:= ———	<u>type</u>
value 1	:= ———	θ
value 2	:= ———	θ
ifthenelse	:= PN	θ
if	:= ———	condition
$then_1$	:= PN	IS(θ, value 1, ifthenelse)
ifhowever	:= ———	NON(condition)
$then_2$	:= PN	IS(θ, value 2, ifthenelse) .

Now suppose that, in the presence of expressions "real", "0", "1", "sum", and "greater(a,b)" etc., we wish to define the function given by the rule :

The derivation rules, evidently, are just theorems. New rules can be devised and used. Note that the text is not subdivided into parts along the usual definition-theorem-proof model. Every line is a result that may be used whenever we wish. A theorem is never announced prior to its proof ; a result cannot be stated until it is derived.

In the next example we define something of the form : "if ..., then ..., else ...". Some assumptions are needed ; the first requirement is that axioms for equality are given before (this is an awkward feature). Also, negation should be defined beforehand.

θ	:= ———	<u>type</u>
x_1	:= ———	θ
x_2	:= ———	θ
IS	:= PN	<u>type</u>
[axioms for IS , for example, Lindenbaum's]		
condition	:= ———	<u>type</u>
value 1	:= ———	θ
value 2	:= ———	θ
ifthenelse	:= PN	θ
if	:= ———	condition
$then_1$	:= PN	IS(θ, value 1, ifthenelse)
ifhowever	:= ———	NON(condition)
$then_2$	:= PN	IS(θ, value 2, ifthenelse) .

Now suppose that, in the presence of expressions "real", "0", "1", "sum", and "greater(a,b)" etc., we wish to define the function given by the rule :

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ x+1 & \text{if } x \leq 0 \end{cases} .$$

Then we may proceed :

```
| x := ———                                                  real
| f := iftheneise (real, greater(x,0), x, sum(x,1))         real
```

And we might apply the text as follows :

```
      a := ....                                             real
 weknow := ....                                             greater(a,0)
  hence := then_i(real, greater(a,0),a,sum(a,1),weknow)     IS(real,a,f(a))
```

One feels that such lines ought to be invented by machines. An auxiliary language, or a small handbook, might inform us how to construct such things without thinking.

<u>The limitations of PAL</u>

In a book with a block like :

```
| | | | | x := ———   number
| | | | | f := Σ(x)   number
```

where $\Sigma(x)$ denotes an expression containing x , a function is available, actually given by an explicit construction. But we cannot say "assume we have a block like this", or, "let f be a function mapping something to something else", or, "for every function it is true that".

Suppose a and b are booleans and that we have a block :

```
| 1 :- ------ TRUE(a)
| 2 .= ...... TRUE(b)   .
```

Then we cannot say that "a ==> b is true". That would amount to saying that, in the language, there is a block like that.

In addition, the induction axiom for the natural numbers cannot be described properly. In the 18th century implication and induction were certainly somewhat metalingual, and therefore, mysterious. In current mathematics we have incorporated them into our formalisms. Likewise for PAL, some metalingual extension is indicated.

METALANGUAGE

Change of Variables

The possibility of interchanging variables requires a metalanguage. For example, in elementary geometry, a proof might be given involving a triangle whose vertices are lettered A , B and C . Then it might be stated that the proof can be repeated with B and A interchanged. This is economical, but nonessential, use of metalanguage. (One could, after all, avoid metalanguage by repeating the proof). It should be remarked that in PAL this use of metalanguage is possible.

Models

Metalingual features present themselves in the case of models. Assume there is an axiom, in context o ,

k := PN Y

followed by a chapter K of the book, of "conclusions". And later, let us suppose something in category Σ is obtained in some context.

| | | a := A Σ

We should like to say, "Everything that was derived for k can be derived with a ". We have a model for the axiom. Indeed, this can be done by rewriting chapter K . However, we should prefer to abbreviate this. An abbreviation is possible if the PN can be replaced by a block opener.

k := ——— Y .

If however, the axiom is "covered" thus :

| x := ——— Γ
| k := PN Σ

then the abbreviation is more difficult. In some languages of AUT type (depending on whether Γ and Σ are 1- or 2-expressions) this can be accomplished. In the case of a <u>set</u> of axioms, complications abound. (It can happen that the second axiom can only be formulated after the first is assumed, etc.). Axioms can be dispensed with in an extension of AUTOMATH called AUT-SL (single-line AUTOMATH). AUT-SL is however mainly of theoretical importance.

<u>Local Axioms</u>

Let us say that, in a mathematical encyclopaedia, it is desired to place a set of axioms, for projective geometry. After these axioms are written down, they are available for use everywhere in the book. This very unaesthetic idiosyncrasy can be removed by a lock-and-key technique. We start with one harmless axiom :

projgeom := PN <u>type</u>

Thereafter, we continue :

| if := ——— projgeom (the only appearance of this type)
| followed by whatever axioms are needed.

Outside the block the axioms cannot be used as long as there is nothing in projgeom. A kind of chastity belt !

<u>Composite Notions</u>

A composite notion like "let G be a group" cannot be introduced in the same manner as

x := ——— number

One must be able to express for (G, ∘, *) "let G be a set, let ∘ be a binary operation on G , let * be a proof for the group axioms". One needs a scheme approximately like :

```
|       G         := ——— set
|  |    prod      := ——— map  G × G → G
|  |  | assump 1  := ——— ...
|  |  | assump 2  := ——— ...
```

The chastity belt technique permits a quick reference to a composite declaration. Thus :

```
|  GROUP    := PN    type
|  key      := ——— GROUP
|  G        := PN    set
|  prod     := PN    .....
|  assump   := PN    .....
   etc...
```

The extra axioms introduced by the technique are harmless ; nothing can be derived from them as long as we do not say that we assume we have a key. But on the other hand, if we do have an object which does satisfy the group axioms, we are not yet in the position where we can produce a key. Without a further extension of the language it appears that, in those circumstances, are bound to do the following

```
|       G*        := ——— set
|  |    prod*     := ——— map
|  |    :
|  |  | axiom*    := PN    GROUP
```

Subsequently, it would be necessary to write, by means of axiomatic equality that G* is equal to G(axiom*) etc... All this appears to be very clumsy. Nevertheless the lock-and-key technique is useful in many situations. AUT has no facilities for condensing composite declarations into single lines. We need auxiliary languages for this.

A thorough investigation into the metalingual requirements of mathematical languages has yet to be undertaken.

The Lambda Calculus (A. Church 1937)

It is strange that the efficient notation of Church for representing functions is not in general use. Perhaps Bourbaki is responsible. In Church's notation, the function which sends x to $x^2 + x$ is represented by

$$\lambda_x(x^2 + x) \quad .$$

The function, which for a certain parameter a, sends x to $x^2 + ax$ is denoted by $\lambda_x(x^2 + ax)$. For a more involved example, consider the mapping in Hilbert space which sends g to the map L_g, where $L_g(f) = \langle f,g\rangle$ (inner product). The map L_g sending $f \to \langle f,g\rangle$ is $\lambda_f\langle f,g\rangle$. The function described above is simply $\lambda_g\lambda_f\langle f,g\rangle$.

In analysis Freudenthal's Y-notation is convenient, mostly because it avoids the use of a letter which is nice to have available for other purposes. For example, the Fourier transform in L_2 can be denoted :

$$F = \underset{g}{Y}\ \underset{x}{Y} \int_{-\infty}^{\infty} e^{2\pi ixt}\ g(t)\ dt \ .$$

Just as with the qualifiers $\forall$, $\exists$, there is an obvious need for indicating sets or types. So, in analogy with $\forall\, x \in R \ldots$, we should rewrite the preceding definition :

$$F = \underset{g\in L_2}{Y}\ \underset{x\in R}{Y} \int_{-\infty}^{\infty} e^{2\pi ext}\ g(t)\ dt \quad .$$

In a formula like $\lambda_x(x^2 + xy + z)$, two kinds of substitution are possible, viz. for the "parameters" y, z and for the "variable" x.

The standard notation, $f(k)$, to indicate that k has replaced the variable x is ill-adapted to the language PAL and its extensions. In the first place, the parentheses (,) are used to indicate a change of context in PAL (i.e. a substitution for parameters). Furthermore, with the "quantifier" written on the left, it would be unwieldy to place the "inverse operation" (substitution for a variable) on the right. Therefore we introduce new symbols, (braces), and write $\{b\}f$ in place of $f(b)$ as used in ordinary mathematics.

Folding Rule (β-reduction) :

$$\{b\}\ \lambda_x(x^2 + x + e^x) \text{ reduces to } b^2 + b + e^b .$$

(Subsequently "reduces to" will be represented by the symbol $\geq$) . There is also α-reduction :

$$\lambda_x(x^2 + x + e^x) \geq \lambda_y(y^2 + y + e^y)$$

and η-reduction :

$$\lambda_x\{x\}\ \Lambda \geq \Lambda \text{ if } x \text{ does not occur in } \Lambda .$$

The Normal Form Problem

In an expression like :

$$\lambda_x\{\Lambda\}\ \{\sum\}\ \lambda_y\{K\}\ \lambda_u\{P\}f$$

we can attempt to simplify by folding, hoping to obtain an expression of the type : $\lambda_{x_1}\ \lambda_{x_2}\ \lambda_{x_3}\ \{\Gamma_1\}\{\Gamma_2\}f$. In this expression folding is not possible any more, supposing that it is not possible in Γ_1 and Γ_2 . We say that the expression is in normal form. But folding often makes things worse. Church gave the clever example :

$$\{\lambda_x\{x\}\ x\}\ \lambda_y\{y\}\ y$$

If we agree to denote $\lambda_x\{x\}x$ by Γ, then folding just gives $\Gamma'\Gamma$ again.

Expressions like $\{x\}x$ do not occur in mathematics. If x is a function, then its variables are of a different type. In AUTOMATH we think in terms of categories, and we extend the λ-calculus with these. In fact, AUTOMATH and related languages are just PAL augmented by such a λ-calculus. For the original version of AUTOMATH, the Normal Form Theorem, i.e. the statement that every expression reduces to an expression in normal form, has not been proved. It has been proved however for certain closely related languages.

Description of AUTOMATH

In place of $\lambda_{x\in real}$, we shall write $[x,real]$. And if, for x real, Σ has category "point", then the mapping $x \to \Sigma$ is denoted by $[x,real]\,\Sigma$ (where Σ may depend upon x) and is said to have category $[x,real]$ point. (In describing AUTOMATH we shall often place numerals above metalingual symbols to denote what sort of expression they are : thus $\overset{②}{\Lambda}$ represents a 2-expression, etc.). If we have a block :

$$\begin{array}{c|l}
\theta & x := \text{———} \quad \overset{②}{\Lambda} \qquad (\Lambda \text{ may } \underline{\text{not}} \text{ contain } x \text{ but } \Sigma \text{ and } \Gamma \text{ may}) \\
x & q := \overset{③}{\Sigma} \qquad \overset{②}{\Gamma} \\
 & \text{we allow ourselves to write} \\
\theta & p := [x,\overset{②}{\Lambda}]\,\overset{③}{\Sigma} \qquad [x,\overset{②}{\Lambda}]\,\overset{②}{\Gamma}
\end{array}$$

That is, in context θ the "expression" $[x,\overset{②}{\Lambda}]\,\overset{③}{\Sigma}$ is admissible, and its CAT is $[x,\overset{②}{\Lambda}]\,\overset{②}{\Gamma}$; the expression $[x,\Lambda]\,\Gamma$ is admissible and has category <u>type</u>. This innovation is called <u>functional abstraction</u>. For formal rea-

sons it may be preferable to replace x in the latter line by a special symbol t (a <u>bound variable</u>) ; one can imagine that it is used wherever confusion is possible, and that, roughly, two expressions are considered to be "the same" provided they can be transformed into one another by "legitimate" α-reduction.

Suppose we have a line :

$$| \; | \; | \; | \quad := [t,\Lambda]\ \Sigma \qquad [t,\Lambda]\ \Gamma$$

The block itself may be reconstructed if we stipulate that

(1) $\{x\}\ [t,\Lambda]\ \Sigma \overset{D}{=} S_{t\to x}\ \Sigma$, and

(2) if Π is acceptable, and $CAT(\Pi) \overset{D}{=} A$, then $\{\Pi\}\ [t,\Lambda]\Sigma$ is acceptable and $CAT\{\Pi\}[t,\Lambda]\Sigma = S_{t\to\Pi}\ \Gamma$.

We also admit this in a more restrictive context (as observed in PAL). The original block appeared as :

$$\begin{array}{|l} x := \text{———} \\ \quad := \{x\}\ [t,\Lambda]\ \Sigma \qquad S_{t\to x}\ \Gamma \end{array}$$

The latter expressions are definitionally equivalent to $\overset{③}{\Sigma}$ and $\overset{②}{\Gamma}$, respectively. Let us suppress the precise details of the language definition (see [7] for quite an extensive treatment) ; first, some examples.

EXAMPLE 1 (distributivity of inner product)

```
nat    := ...    type
real   := ...    type
a      := ——     real
b      := ——     real
sum    := ...    real
prod   := ...    real
EQ     := ...    type
c      := ——     real
distr  := ...    EQ(prod(sum(a,b),c),sum(prod(a,b),prod(b,c)))
```

```
m        := ———                      nat
finset   := ......                   type        (finite set of m
                                                 elements)
Y        := finset(m)                type
vector   := [t,y] real               type
f        := ———                      vector
rowsum   := ......                   real        (the proper description
                                                 of the sum of the compo-
g        := ———                      vector      nents of vector f can-
                                                 not be given here)
sumvec   := [t,y] sum({t}f,{t}g)     vector
provdec  := [t,y] prod({t}f,{t}g)    vector
EQV      := [t,y] EQ({t}f,{t}g)      type
inprod   := rowsum(prodvec)          real
h        := ———                      vector
assum    := ———                      EQV(h, sumvec)
lemma    := ......                   EQ(rowsum(h),sum(rowsum(f),rowsum(g)))
distv    := [t,y] distr({t}f,{t}g,{t}h)  EQV((f+g)h,fh+gh) (abbreviated)
theorem  := lemma(fh,gh,(f+g)h,distv)    EQ(inprod(f+g,h),inprod(f,h)+inprod(g,h))
```

EXAMPLE 2 : the induction axiom

```
nat        := PN        type
bool       := PN        type
| b        := ———       bool
| TRUE     := PN        type
1          := PN        nat
succ       := PN        [x,nat]nat  (the successor function)
| P        := ———       [x,nat]bool (a predicate)
| | assume 1  := ———     TRUE ({1}P)
| | | assume 2  := ———  [n,nat] [t,TRUE({n}P)] TRUE({{n}succ}P)
| | | inductax  := PN   [n,nat] TRUE({n}P)
```

To illustrate its use, suppose that in some context we have CAT(Q) = [x,nat]bool and that we have CAT(demo) = TRUE({1}Q) and CAT(k) = nat . We should like to assert TRUE({k}Q) , assuming that we already have :

```
| m        := ———       nat
| | IF       := ———     TRUE({m}Q)
| | THEN     := ....    TRUE({{m}succ}Q)
```

In this situation we write simply :

result := {k}inductax (Q,demo,[n,nat][t,TRUE({n}Q)] THEN(n,t)) TRUE({k}Q).

The machine finds the middle expression acceptable and gives for its CAT , {k} [n,nat] TRUE({n}Q) , which is folded into TRUE({k}Q) .

Which functional abstraction to choose ? In AUT we begin with mappings from 3-expressions to 3-expressions. Along with $[x,\Lambda]\,\overset{③②}{\Sigma}\,\overset{③}{}$ we must accept $[x,\Lambda]\,\overset{③②}{\Gamma}\,\overset{②}{}$. At this stage, little has been achieved. A notation is a-

vailable for "consider f as a function of x" , but not yet for "let f be a function". Therefore, we open the possibility to write :

$$\theta \quad ||| \quad | \; f := \overset{②}{\text{———}} \; [t,\Lambda] \; \overset{②}{\Gamma}$$

provided that the 2-expressions Λ and Γ are acceptable at θ . In this block $\{\Pi\}f$ is acceptable with $CAT\{\Pi\}f = S_{t\to\Pi}\Gamma$ whenever Γ is acceptable. And what should we write for $CAT([t,\Lambda],\Gamma)$, which should be a 1-expression ? Must we adhere to the former convention that <u>type</u> is the only 1-expression ? Or shall we say that it is $\overset{②}{[t,\Lambda]}\ \overset{②}{CAT(\Gamma)}$? In the latter case, if $CAT(\Gamma) =$ <u>type</u>, we have $CAT([t,\Lambda]\Gamma) = [t,\Lambda]$ <u>type</u> . The first choice is taken in AUT. It is natural to want "real function" and "real number" both as types.

The intermediate point of view offered by AUT-QE admits both $[t,\Lambda]$ <u>type</u> and <u>type</u> , so that if we know

1) $:= \Sigma \quad [t,\Lambda]$ <u>type</u>

we allow 2) $:= \Sigma \quad$ <u>type</u>

Note that line (1) bears more information than line (2). AUT-QE is a very handy language, but quite sophisticated ; a new complication is the nonuniqueness of CAT. That uniqueness may be preserved if we make the convention that the reduction rule $[t,\Lambda]$ <u>type</u> $\to$ <u>type</u> applies only to <u>substitution rights</u>. So that, given a block like :

Θ := ——— type

b := ..... real, when we have

... := Σ [t,Λ] type, we shall also accept

... := b(Σ) real.

By abuse of metalanguage, [t,Λ] [s,Σ] type $\subset$ [t,Λ] type $\subset$ type . Of course, we might have formulated these conditions by another word, supertype, and then agreed that in

Θ := ——— type

:= ..... , we may only substitute Σ for Θ

if CAT(Σ) = type . However, if we have

η := ——— supertype

:= ..... , we may also substitute

Γ for η if CAT(Γ) = [x,...] [y,...] type , and so forth.

This would open up a vast new area; we approach such an undertaking with trepidation. Present experience with AUT-QE suggests that shorter writing is possible than with AUT . Note that some obvious rules about DEFEQUAL must be observed, for example :

if $\Lambda_1 \overset{D}{=} \Lambda_2$ and $\Gamma_1 \overset{D}{=} \Gamma_2$, then $[t,\Lambda_1]\Gamma_1 \overset{D}{=} [t,\Lambda_2]\Gamma_2$.

Before inspecting the next piece of text, let us take note of the following possibility. When we have a block like :

```
a      := ......   bool
b      := ......   bool
if     := ———      TRUE(a)
then   := Σ        TRUE(b)
          Functional abstraction permits us to write
....   := [t,TRUE(a)]Σ  [t,TRUE(a)] TRUE(b)
```

This observation explains the derivation of the sixth line in the piece of text below, a derivation of modus ponens in AUT.

```
bool     := PN                      type
b        := ———                     bool
TRUE     := PN                      type
a        := ———                     bool
b        := ———                     bool
IMPL     := [x,TRUE(a)] TRUE(b)     type
ass1     := ———                     TRUE(a)
ass2     := ———                     IMPL
modpon   := {ass1} ass2             TRUE(b)
```

Now we investigate quantification. First we introduce the ALL symbol (over a type) ; it is slightly harder to handle over a subset of a type.

```
Θ        := ———                     type
P        := ———                     [x,Θ]bool      "a predicate"
ALL      := [u,Θ] TRUE({u}P)        type
            observe that :
a        := ———                     Θ              Roughly, if P(x)
alltrue  := ———                     ALL(Θ,P)       is true for all x ,
then     := {a} alltrue             TRUE({a}P)     then P(a) is true
            and if there is a block :
b        := ———                     Θ
then     := Σ                       TRUE({u}P)
            functional abstraction allows
....     := [u,Θ]Σ                  [u,Θ]TRUE({u}P)
....     := [u,Θ]Σ                  ALL
```

Note the analogy between the texts for IMPL and ALL.

Existence is an example of a notion that has been taught by intimidation. Actually there are several forms of existence ; we begin with a strong form (Note the slight deviation from [6], p. 73).

```
Θ            := ———— type
P            := ———— [x,Θ] bool
EXISTS       := PN   type
v            := ———— Θ                    } an axiom
ass1         := ———— TRUE({v}P)           } for any kind
then1        := PN   EXISTS               } of existence

ass2         := ———— EXISTS               } Hilbert's
epsilon      := PN   Θ                    } axiom
itsatisfies  := PN   TRUE({epsilon}P)     }
```

In a simpler form (with P identically true) we have the notion NONEMPTY.

```
Θ            := ————                      type
NONEMPTY     := PN                        type
ass3         := ————                      NONEMPTY
iota         := PN                        Θ
```

The axiom of choice is implemented by a text like :

```
Θ            := ————                      type
x            := ————                      Θ
F            := ......                    type
ass4         := ————                      [t,Θ] NONEMPTY(F(t))
then2        := [t,Θ]iota(F(t),{t}ass4)   [t,Θ] F(t)
```

There is a weaker form of existence (essentially $\exists = \neg \forall \neg$) which is still workable. Let us restrict it to nonempty ; we call the notion NEPTY. First let us remark how nonemptiness is <u>used</u> in mathematics. We have a proposition p , and a set S ; we know that S is nonempty. Then, if for all $a \in S$ the proposition p is true, then p is true. So we define nonemptiness by $\forall_{p \in bool} \; (\forall_{a \in S} \; p \Longrightarrow p)$.

Θ	:= ———	<u>type</u>
NEPTY	:= [c,bool][u,[x,Θ]TRUE(c)] TRUE(c)	<u>type</u>
	First come the lines	
a	:= ———	Θ
then3	:= [c,bool][u,[x,Θ] TRUE(c)] {a}u	NEPTY
	As for the use of NEPTY, suppose we have :	
ass5	:= ———	NEPTY
P	:= ———	bool
y	:= ———	Θ
then3	:= Σ	TRUE(P)
	then we may write	
conclusion	:= {[y,Θ]Σ} {P}ass5	TRUE(P)

Negation may be approached in the following manner :

CON(TRADICTION)	:= [x,bool] TRUE(x)	type
Θ	:= ———	type
nonempty	:= PN	bool
ass6	:= ———	TRUE(nonempty(Θ))
then4	:= PN	NEPTY(Θ)
ass7	:= ———	Θ
then5	:= PN	TRUE(nonempty(Θ))
c	:= ———	bool
NON	:= [x,TRUE(c)] CON	type
non	:= nonempty(NON(c))	bool

We leave it as an exercise to prove that :

ass8	:= ———	TRUE(c)
ass9	:= ———	TRUE(non(c))
then6	:=	CON

As a consequence, TRUE(non(c)) implies NEPTY(NON(c)) , and together with TRUE(c) , this leads to CON . The awkward jumps from Θ to TRUE(nonempty(Θ)) involve many extra lines, a very annoying feature.

Set theory may be formulated in two ways. In the first of these, an exposition of axiomatic set theory, it is permitted that sets are elements of other sets, possibly even themselves. Thus :

set	:= PN	type
a	:= ———	set
b	:= ———	set
∈	:= PN	bool
⊂	:= PN	bool
∪	:= PN	set
∩	:= PN	set

Now some axioms are added like :

if	:= ———	TRUE(∈(a,b))
c	:= ———	set
and	:= ———	TRUE(⊂(b,c))
then	:= PN	TRUE(∈(a,c))

An entirely different approach is to treat sets as <u>sets of things of a certain kind</u>. We do not need to form the union of a set of triangles with a set of real numbers. We think of types as sets ; "set" becomes a 1-expression depending on a 2-expression.

```
Θ     := ———        type
set   := PN         type

         Here ∈ may be introduced as follows

S     := ———        set(Θ)
a     := ———        Θ
∈     := PN         bool
```

In this formation $x \in x$ does not occur. The requirements of the axiom of extensionality makes us realize that it is easier ot define the notion "set" by means of a predicate; vide :

```
Θ     := ———        type
set   := [x,Θ] bool  type
S     := ———        set
a     := ———        Θ
∈     := {a}S       bool
```

Here a set is a subset of a 2-expression. Accordingly, every type is itself a set. If, furthermore, the natural numbers N are available, the language facilities of AUTOMATH permit us to construct the sets $N, N^N, N^{N^N}, \ldots$ but <u>not</u> their union. For if we have nat := PN <u>type</u> , we can form [n,N]N or N^N , which is, essentially, the set of real numbers. N^N is again a type and we may quantify over it. Cantor constructed

all these sets and took their union, which led him into Cantor's Paradise. An infinite book would be required to enable us to do that here. However, we are already in the Analysts' Paradise. No more than this is ever encountered by the analyst.

Extensions of AUT

Let Λ and Γ be 2-expressions. In AUT the expression $[x,\Lambda]\Gamma$ has CAT type. The extension of AUT called AUT-QE permits us to write CAT of the expression $[x,\Lambda]\Gamma$, as $[x,\Lambda]$ type ; the latter is a new 1-expression -a mapping type. We may reduce $[x,\Lambda]$ type to type but it is not obligatory. An expression Σ , with $CAT(\Sigma) = [x,\Lambda]$ type may be substituted for a variable η with $CAT(\eta)$ = type ; but it is forbidden to substitute an expression Σ with category type for a variable with category $[x,\Lambda]$ type . Nor can we substitute Σ with $CAT(\Sigma) = [x,A]$ type for a variable with category $[x,B]$ type , if $A \overset{D}{\neq} B$. In case $CAT(\Sigma) = O$ and $CAT(\Lambda) = [x,O]\,[\quad]\ldots[\quad]$type , we must add the right to use $\{\Sigma\}\Lambda$ and the corresponding folding rule. We do not admit quantification over anything other than 2-expressions.

In the language AUT-SL , quantification over 1-expressions is admitted. Then everything can be written at level 0 ; PN's are eliminated. A similar level-0 language employing AUT rules has been described by Nederpelt (λ-AUTOMATH) . Actually, writing at level 0 with unlimited quantification does not appear to be harmful ; even for AUT-SL the normal form

theorem can be proved. The tough question is, how far can we go with type reduction and/or substitution rights ? The elimination of PN's has the great advantage that <u>models</u> can be used : if we have a model for a PN , then all that was derived for the PN is at once available for the model. The language AUT-SL is defined by a computer programme. In fact, the same computer programme is used to check the language. Since the language is defined by the programme, the necessity of proving that the programme describes the language is sidestepped.

Let us conclude by examining a piece of text in AUT-QE. The same text in AUT appears in [11] ; the interpretation is the introduction of limits. Since <u>type</u> predicates can be used exclusively, the ALL-quantifier is unnecessary.

Θ	:= ———	<u>type</u>	
Q	:= ———	$[x,\Theta]$ <u>type</u>	(Q is a <u>type</u> predicate in Θ)
x	:= ———	Θ	
q	:= $\{x\}$ Q	<u>type</u>	

Note that if $CAT(a) = \Theta$, then $q(a) \overset{D}{=} \{a\}\ Q$

ALL	:= $[u,\Theta]$ $\{u\}$ Q	<u>type</u>	(see : introduction of ALL)

By η-reduction, $[x,\Theta]\ \{x\}\ Q \overset{D}{=} Q$, so that there is something in Q iff there is something in ALL. The type predicate is "the same" as the ALL quantifier. For the present text it is irrelevant what notion of existence is chosen.

Θ	:= ———	<u>type</u>	
Q	:= ———	[x,Θ] <u>type</u>	
EXISTS	:= PN	<u>type</u>	(not strictly necessary -see previous discussion of existence)
a	:= ———	Θ	
ass	:= ———	{a} Q	
axiom	:= PN	EXISTS	

The auxiliary notions are identical in appearance to their AUTOMATH versions (see [11] p. 9).

real	:= PN	<u>type</u>	
r_1	:= ———	real	
r_2	:= ———	real	
dist	:= PN	real	
LESS	:= PN	<u>type</u>	
null	:= PN	real	
nat	:= PN	<u>type</u>	
1	:= PN	nat	
if	:= ———	LESS(null,r_2)	
lemma	:= PN	LESS(dist(r_1,r_1),r_2)	
k_1	:= ———	nat	
k_2	:= ———	nat	
LESSNAT	:= PN	<u>type</u>	
Sequence	:= [x,nat]real	<u>type</u>	

If a is a sequence and l is real, the LIM will assert that $\lim_{n\to\infty} a_n = l$. CONV asserts that such a limit exists.

a	:= ———	sequence
l	:= ———	real
δ	:= ———	real
t	:= ———	LESS(null,δ)
p	:= [n_0,nat][n,nat][u,LESSNAT(n_0,n)]LESS(dist({n}a,l),δ)	[n_0,nat]<u>type</u>
LIM	:= [δ,real][t,LESS(null,δ)]EXISTS(nat,p(δ,t))	<u>type</u>
CONV	:= EXISTS(real,[l,real] LIM(l))	<u>type</u>

The theorem that the sequence with constant value c converges to the limit c follows. Compare the proof written in AUT ([11] p.10) with the proof in AUT-QE. The logical structure is the same as it is for any limit theorem ; technically it is quite simple.

c	:= ———	real
p	:= [x,nat]c	sequence
$\overline{\delta}$	:= ———	real
ass	:= ———	LESS(null,$\overline{\delta}$)
Q	:= P(p,c,$\overline{\delta}$,ass)	[n_0,nat] <u>type</u>
now	:= [n,nat] [u,LESSNAT(1,n)] lemma({n}p,$\overline{\delta}$,ass)	{1} Q
thus	:= axiom(nat,Q,1,now)	EXIST(nat,Q)
THM1	:= [$\overline{\delta}$,real][t,LESS(null,$\overline{\delta}$)] thus($\overline{\delta}$,t)	LIM(p,c)
THM2	:= axiom(real,[s,real] LIM(p,s),c,THM1)	CONV(p,c)

The "technical" part of the proof is simply the segment

lemma ({n}p,$\overline{\delta}$,ass)

appearing in line 6. The rest is the "logical" part of the proof. AUT-QE is somewhat easier to write than AUT, judging by this example; yet AUT-QE is a stronger language. Hopefully, this one example may serve for many. Those interested in pursuing the matter may consult the bibliography.

BIBLIOGRAPHY

[1] ARNOLD, A., Les mathématiques à la portée de l'ordinateur, Initiation aux nouveautés de la science 22, Dunod, Paris (1970).

[2] ARNOLD, A., Le langage mathématique formel VAT' 70, Publications du Laboratoire de Calcul de la Faculté des Sciences de l'Université de Lille, n° 21 (1970).

[3] ARNOLD, A., Formalisation des démonstrations mathématiques, Thèse, Lille (1968).

[4] ABRAHAMS, P., Machine Verification of Mathematical Proof, Math. Algorithms, Vol. 1 (1966), vol. 2 (1967), vol. 3 (1968).

[5] BISHOP, F., Foundations of Constructive Analysis. McGraw-Hill, (1967).

[6] de BRUIJN, N.G., The mathematical language AUTOMATH, Springer Lecture Notes in Mathematics NR 125 (1970).

[7] de BRUIJN, N.G., AUTOMATH, a language for mathematics. Report 68-WSK-05. The Technological University Eindhoven (THE).

[8] de BRUIJN, N.G., Verificatie van Wiskundige Bewijzendoor een Computer. Colloquiumvoordracht, THE.

[9] de BRUIJN, N.G., SEMIPAL 2, an extension of the mathematical notational language, SEMIPAL, Notitie Nr 43, THE (1969).

[10] de BRUIJN, N.G., Generating functions for context-free languages given in Backus normal form. Notitie Nr 7, THE (1969).

[11] de BRUIJN, N.G., Vérification des textes mathématiques par un ordinateur. Conférence à Lille (1969).

[12] de BRUIJN, N.G., Machinale verificatie van redeneringen, Voordracht Kon.Ned.Akad.v.Wetensch, (1969).

[13] de BRUIJN, N.G., A processor for PAL. Notitie 30, THE (1970).

[14] de BRUIJN, N.G., The syntax of PAL and AUTOMATH. Notitie 32, THE (1970).

[15] de BRUIJN, N.G., On the use of bound variables in AUTOMATH. Notitie 9, THE (1970).

[16] de BRUIJN, N.G., Coding system for AUT-QE. Notitie 11, THE (1970).

[17] de BRUIJN, N.G., Formulas with indications for establishing definitional equivalence. Notitie 15, THE (1970).

[18] de BRUIJN, N.G., AUT-SL, a single line version of AUTOMATH. Notitie 22, THE (1971).

[19] JUTTING, L.S., Example of a text written in AUT-QE. Report, THE (1970).

[20] JUTTING, L.S., Definition of the language AUTOMATH. Report, THE (1970).

[21] JUTTING, L.S., Example of a text written in the formal language AUTOMATH. Report, THE (1970).

[22] MORSE, A.A., A theory of sets. Academic Press (1965).

[23] NEDERPELT, R.P., AUTOMATH, a language for checking mathematics with a computer. Report, THE (1970).

[24] PIETRZYKOWSKI, T., A language for computer assisted theorem proving. Department of Applied Analysis and Computer Science Research Report C5RR 2009. Faculty of Mathematics, University of Waterloo (1969).

[25] ROBINSON, J.A., Theorem Proving on the Computer. Journal ACM, vol. 10 (1963), p. 163.

[26] VAN DER WAERDEN, B.L., Science Awakening, Noordhoff, Groningen (1961).