UNIVERSITÉ PARIS XI

U.E.R. MATHÉMATIQUE

91405 ORSAY FRANCE

100-7420

APLASM 73

SYMPOSIUM D'ORSAY SUR LA MANIPULATION DES SYMBOLES ET L'UTILISATION D'APL

VOLUME 1

LE PROJET AUTOMATH

ÉDITÉ PAR P. BRAFFORT ET G. KIREMITDJIAN

LE SYMPOSIUM APLASM 73

Les 20 et 21 décembre I973, le Département de Mathématiques de l'Université Paris-Sud (Laboratoire Al Khowarizmi) organisait à Orsay un Symposium International consacré aux problèmes de la manipulation des symboles en mathématique pure et à l'utilisation du système APL.

Plus de cinquante participants venus de huit pays différents furent acceuillis par G. POITOU et participèrent aux sessions présidées par M. DEMAZURE, H.HAEGI, G. MARTIN, J. DELBREIL.

Une introduction générale au projet LIMA et aux problèmes généraux abordés au cours du Symposium est publiée séparement (note ECSTASM N°1).

Nous publions, avec le concours de l'IRIA (*) les communications présentées pendant ces deux journées en les regroupant en trois volumes qui correspondent aux trois pôles d'intérêts principaux.

Certaines communications n'étaient pas disponibles pour publications,par contre nous avons ajouté plusieurs textes correspondant à des travaux effectués postérieurement au Symposium et qui permettent de parfaire l'homogénéité de l'ensemble.

P.B , M.D.

(*) Contrat SESORI 73 021

CONTENTS

VOLUME ONE : THE AUTOMATH PROJECT

VOLUME TWO : THE LIMA PROJECT

VOLUME THREE : RESEARCH ON APL

I.0 INTRODUCTION

DU PREMIER CHAPITRE

par P. BRAFFORT.

Le projet AUTOMATH est né de la constatation, par un "mathématicien pratiquant", des possibilités de l'automatisation et, dès lors, de la nécessité d'une micro analyse du langage mathématique.

Ce qui fait la valeur exemplaire d'AUTOMATH, c'est que ce projet correspond ainsi à un besoin réel, et n'est pas un exercice gratuit de logicien ou d'informaticien.

L'équipe d'Eindhoven , conduite par le professeur de BRUIJN a bientôt attirer l'attention de logiciens éminents en même temps qu'elle découvrait, pour son propre compte, l'importance de certains problèmes formels (notamment ceux de l'abstraction fonctionnelle).

Le caractère très général des objectifs que le projet s'est fixé correspont bien aux ambitions de l'équipe d'orsay pour son projet LIMA , et il était donc naturel qu'AUTOMATH soit présent au symposium APLASM.

On trouvera donc dans les articles qui suivant une présentation complète d'AUTOMATH : généralités, description formelle, exemples, et algorithme programmé en ALGOL.

Nous y avons joint le travail d'un des membres de notre équipe, qui a réalisé l'algorithme de vérification en PAL (un sous-ensemble d'AUTOMATH), en utilisant lelangage APL.

Ce "LIMA-PAL" nous permet ainsi de réaliser une transition naturelle entre AUTOMATH et LIMA.

THE AUTOMATH mathematics Checking Project

par

N.G. DE BRUIJN*

Cet article présente le projet AUTOMATH du point de vue historique. L'auteur insiste sur les motivations qui l'ont conduit à définir et réaliser un système universel de vérification automatique des textes mathématiques.

* Département of Mathematics
Technological University

Eindhoven, the Netherlands.

This lecture will describe the AUTOMATH project, but will be pretty vague about the nature of AUTOMATH language itself. We refer to [1] and [2] for details about the definition of the language; here we shall mainly concentrate on motivation.

One source of confusion should be taken away at the start: AUTOMATH is a mathematical language and not a programming language. Nevertheless the two kinds of languages have much in common, and can certainly profit from each other's ideas.

The AUTOMATH project was conceived in 1966. The idea was to develop a system of writing entire mathematical theories in such a precise fashion that verification of the correctness can be carried out automatically, yet keeping, step by step, contact with ordinary mathematical presentation. A similar idea possibly existed in the mind of Leibniz but did not develop at that time since there was neither interest nor experience in formal linguistics.

The idea is to make a language such that everything we write in it is interpretable as correct mathematics, as long as our writing is syntactically correct (including correct references to things that have been said before). This may include the writing of a vast mathematical encyclopaedia, to which everybody (either a human or a machine) may contribute what he likes, and any contribution that has been accepted syntactically can be safely used by others. The idea of a kind of formalized encyclopaedia was already conceived and partly carried out by Peano around 1900, but that was still far from what we might call automatically readable.

The task of checking syntactic correctness can be left to a computer. Since the checking only concerns the question whether the text has been written according to the rules, we have to admit that the task of the checking is as human as the task of the writing. Yet, the idea of a computer is in the background in order to set the standards: what a computer cannot do, cannot be called automatic. Moreover, computers have some practical advantages over humans. They take all details seriously and never get bored. The human writer is inclined to change details now and then, and to believe this has no consequences elsewhere; the computer is merciless in this respect.

The speed of the computer is hardly a problem, since we do not expect it to do much more than the human writer can do. The problems there are, concern storage organization in today's computer systems. The mathematician has a subdivided memory: fast and slow parts of the brain, the sheet he is working on, his own recent notes, the books on his desk, the institute library, and finally other libraries he has to depend on if his institute library fails. Similarly the computer system's memory contains flip-flops, core memory, drum, disks, tape, etc. Both in the human and in the machine case, the user has problems to decide what to store where. In the case of the computer, it is quite possible that technological improvements of fast memory will put an end to these storage difficulties in the future.

One of the aims of the AUTOMATH project was, right at the start, to make something of a universal nature. This is a disadvantage over systems that try to tackle small portions of mathematics only, like propositional logic, predicate logic, etc. The need for universality had the effect that no claims could be made in the direction of theorem proving. That subject is so difficult that it can have success only in situations where problems and methods belong to a very limited area, and where language and syntactic analysis have been tailored exactly to the expected situation.

AUTOMATH is a language in which we can write books, consisting of sequences of lines. The syntactic correctness of a line depends on the previous lines. For the time being, we are mainly interested in books that follow ordinary mathematical presentation almost line by line, and do not express thoughts the human mathematician would not have.

We have to realize that no language can embrace all mathematical activity. Language and notation may have an influence on the formation of ideas, but to require that the formation of ideas should always take place in a rigid language would mean killing mathematics. In particular, there is not much chance of putting geometrical or physical intuition in an operational formal framework. On the pure linguistic side, it seems hard to replace illuminating, natural language by something more formal. The psychological function of mathematical understanding is usually more (but sometimes less) than checking correctness: it can be a feeling of peace of mind that sees a mathematical situation in harmony with situations that have become familiar already. Part of that kind of thinking is supposed to be subconscious.

Even if we do not require complete formalization, but just require dependable mathematics at every step, we would kill parts of mathematics, at least in the stage of early development. Important parts of mathematics have been explored on the basis of some fundamental errors, or at least very serious gaps. Without knowing what beatiful things there were at the other side, one would never have had the energy (or the methodology) to repair the error or to bridge the gap. In some cases it has been very lucky for mathematics that one did not have the intellectual facilities to discover that there was an error or a gap at all, until after one had extensive experience with the material beyond.

Let us try to describe the production of completely formalized mathematics as a kind of assembly line. If we think of an AUTOMATH book as a final goal, we have the following phases:

(i) mathematical ideas,
(ii) formal definitions and proofs,
(iii) very precise detailed presentation of these,
(iv) a book in a intermediate language,
(v) an AUTOMATH book.

We have inserted (iv) since AUTOMATH is not so easy to write, because of its universality. Most mathematical material concerns only a small part of mathematics, with well-established traditions about short notations and short ways of saying things. Therefore we have included books of type (iv), in what we may call a problem-oriented language.

What kind of personell do we need on the assembly line? In order to produce (i) we need the Great Mathematician. (Here we do not mean a special class of mathematician: every mathematician can be great now and then). In order to get from (i) to (ii) we need the Good Mathematician, who masters the field and its techniques.

The phases (i) and (ii) have, of course, nothing to do with AUTOMATH; it is the field of standard mathematical practice.

In order to pass the partly finished product from (ii) to (iii) we need a Competent Mathematician. He still has to know the subject, at least he should be able to master the shorthand traditions in the subject.

The transitions from (iii) to (iv), from (iv) to (v), and the final checking of (v), can be left to cheap labour. Much of this, certainly the checking of (v), can be left to very cheap labour in the form of a computer.

There are many things that a universal language like AUTOMATH might achieve. Several of these are, by themselves, not sufficient as a motivation for the AUTOMATH project, but their totality seems important enough for going into some effort. Let us classify the objectives into two groups: checking and understanding .

When saying "checking", the first thing that may come into our mind is the checking of long tedious proofs, where the chain is as weak as its weakest link, and where, quite often, the dependability of the proof is not supported by intuition or experimental evidence. In particular one might expect this situation in complicated combinatorial problems where a large number of cases and subcases have to be checked. Under this heading we also find problems concerning the semantics of computer programs. The number of elementary steps to be taken, and the amount of administration to be carried out, may be so large that human methods become very unreliable. It is in this field that we have to think also about the problems of team-work and of man-machine cooperation. Both require a very rigid communication system. It seems to be worthwhile to work in this field, since tremendous sums of money are spent on computer software, and it is of quite some interest to know what is reliable and what is not.

Let us now look at objectives that fall under the heading "understanding". First we remark that the mere fact of having a fixed well-defined language for mathematics is an advantage all by itself. It enables us to subdivide mathematical discussion into (i) saying things in the language, (ii) discussing how things are said in the language, and (iii) connecting things said in the language with things in another world, like standard mathematics, physical reality, etc. We might refer to (ii) as to "metalanguage" and to (iii) as "interpretation".

Most mathematicians do not have a clear idea of the foundation of their own mathematics. This may partly be the fault of the logicians who, finding so many interesting technical problems in their field, neglected their original

mission, to build a basis for others. Many mathematicians have a vague idea that predicate logic plus set theory form a complete basis for their own activity, but if they look into these fields they see to their surprise that logic and set theory consist of mathematical activity too! Instead of finding a foundation of the mathematical pattern of axioms-definitions-inference rules-proofs-theorems, they seem to find the same pattern again, all over the place. What is lacking, is a good language. Actually, in AUTOMATH these things become quite clear. The language contains hardly anything that can be called logic, and once we have the language and say things correctly (in the syntactical sense) the question of what are axioms, inference rules, definitions, assumptions, theorems, etc. is just metalingual and interpretational. It has not the slightest influence on the results of an AUTOMATH book whether we call a thing a definition or a theorem or anything else; it is just correct as it stands.

Another objective in the direction of "understanding" is analysis of complexity. Some things are more difficult than others, and a complete formal presentation is able to show this. It is possible to classify pieces of mathematics as to their "depth". The mathematics of the 19-th century was certainly deeper than that of the 18-th century. In a somewhat stylized way one can say this: in the 18-th century one could talk about functions one had explicitly constructed, but one could not say "let f be a function", since the word "function" was metalinguistic. In the same stylized fashion one might say that 18-th century mathematics can be expressed by means of PAL, which is the sublanguage of AUTOMATH we get by leaving the lambda calculus out.

In this connection it may be remarked that AUTOMATH violates the historical order. Already in PAL, things like "proofs" are treated in the same way as things like "numbers", whereas even in the second part of the 20-th century most mathematicians feel that a "proof" is a metalinguistic notion and that a "number" is an "object". The ideas about what is an object and what is not, are usually vague. The difference between objects and non-objects is apparently parallel to the distinction between language and metalanguage; one feels that an object is something we can denote by a symbol. Many people believe that it is better to talk about sets than about predicates. Rather than saying that x satisfies the predicate P, they form the set of all things satisfying that predicate, and then say that x belongs to that set.

Usually this is caused by fear for predicates, which are not believed to be objects.

Coming back to "understanding": it has often been said that mathematics is taught by intimidation and learned by imitation. The only way to find out how much truth is in this, is to codify everything in a very rigid language.

Under the heading "understanding" one may finally put the influence that every new notation (provided it has some power) has on the development of mathematics whether one asked for such an influence or not.

Apart from "checking" and "understanding" there are some advantages in the fact that machines can process the mathematics we produce. For example, we can imagine we give a book on analytic number theory to a machine, saying: "I am interested in the Prime Number Theorem only. Print everything that is needed for this theorem and omit everything else" (Some people say that E. Landau was such a machine; he wrote his books that way). Or we can say: "Print Theorem 325, and all definitions needed for reading what it says, starting from scratch". In this case all proofs will be omitted too.

In order to show a glimpse of how mathematical reasoning is expressed in an AUTOMATH book, we have to explain a little about the language. First we note that books are organized into nested "blocks" of lines. The first line of a block has a special form. Its interpretation is that we introduce either a variable that can be used inside the block, or an assumption valid throughout the block.

The lines all have this form:
"In the context A the name B is defined as C and is of type D ".
Here B is a new identifier, not used in previous lines. C and D are expressions in terms of old identifiers, with the use of a few connectives like brackets, parentheses, commas, etc. Some lines (the block openers) have just a bar (——) instead of C (interpretation: a variable is introduced by giving it a name and saying what type it has). In each line, the A is a string of previously introduced block opening identifiers. The A-parts of the lines serve to indicate the block structure of the book, indicating for each line to which blocks it belongs.

Sometimes the C is not an expression, but the special symbol: "PN". The lines where the C-part is PN, serve to introduce primitive notions, which are not defined. These things just get a name and a type, but no definition, yet they

can be used from then on. A PN-line is not a block opener, it just occurs somewhere inside a block.

We have to mention the possibility that the D-part of a line is not an expression, but just the symbol "type". Such lines introduce a new type, either by definition, or as a primitive, or as a variable.

This describes very roughly the structure of the language PAL, mentioned earlier in this lecture. The languages of the AUTOMATH family arise from PAL if we add some kind of typed lambda calculus. We shall not discuss this here.

Let us say a few things about interpretation. First, the context indication (the A-part) is a thing that is usually not explicitly stated in mathematics. Parts of it can be derived from things like subdivision into chapters and sections, other parts can be traced by careful reading of the previous text. The B-part has the usual interpretation of the name given to a new object we form or assume. The interpretation of the C and D parts is as described by saying that B is defined by C and is of type D. Let us introduce the symbol $\mathbf{E}$ for this typing: C $\mathbf{E}$ D. In natural language we say things like "3 is a number", but since the word "is" is used for many different things, we prefer "3 $\mathbf{E}$ number".

Some of the types we shall be using, have set-like interpretations. Instead of "3 $\mathbf{E}$ number" one might think of $3 \in S$, where S is the set of all numbers, but we should be careful not to confuse $\mathbf{E}$ and $\in$. In AUTOMATH, the type of a thing C (i.e. the D with C $\mathbf{E}$ D) is uniquely determined, and can be evaluated by means of an algorithm. With $3 \in S$ this is not so since S can be any set containing 3.

Apart from the types with set-like interpretation we can have others. The most important ones are the propositional types. In line with this kind of interpretation, the D-part corresponds to a proposition, and the C-part to its proof. We operate on proofs: if they depend on variables we can substitute expressions for these variables, in the same way as this is done in the case of objects depending on variables. This has the effect that a modified proof (modified by substitution) is accepted as a proof for the correspondingly modified proposition. Note that the B-part of the line is a name for the proof C, and not for the proposition D. The whole line can be called a theorem; later applications of that theorem are made by means of references to B. Note that the majority of the theorem lines will be just stepping stones leading finally to one important theorem line, which

a mathematician would call a theorem; he would not bother to call the other lines even lemmas.

There are also block openers with propositional interpretation. These seem to say: "let x be a proof for the proposition D". That is, these lines introduce assumptions, valid throughout the block. And there can be lines where the C-part is PN. These serve to introduce the truth of the proposition D as an axiom. Thus we have taken care of the three types of propositional lines: theorems, assumptions and axioms.

We are able to create new types if we wish, and we can also select interpretations. For instance, if we want to make a mathematical theory of plane geometric construction with ruler and compass, we need not go to the trouble of coding constructions as sets (according to the dogmatic idea that everything is a set; for criticism see[3]), but we can introduce a type "construction" directly.

We mention another case. For every set Ω, we introduce a type "program(Ω)". If we have C $\in$ program(Ω), than the interpretation is that C is a program acting on the state space Ω. By means of PN-lines we introduce primitive programs and primitive ways to construct bigger programs from smaller components. In other words, we describe the syntax of a programming language in the same book where we have the logic and the mathematics (there is no essential difference between the latter two). Next we can develop, in the same book, axioms about the semantics of the programming language primitives. And, still in the same book, we can derive logical theorems (derived inference rules), mathematical theorems, semantic theorems, special programs, and semantic results on those programs. The various parts can be interwoven. For example, there can be a mathematical treatment of the g.c.d in a number-theoretical setting, a description of a computer program for finding the g.c.d., and a proof that the execution of the computer program terminates and produces the value of the number-theoretical function g.c.d. (For explicit and extensive proposals for semantical treatment of ALGOL-like languages, see[4]). It would not do any harm to write syntax and semantics of two different programs in one book, and to prove, in that book, that program P_1 in language Q_1 has the same semantic effects as program P_2 in language Q_2. Proofs of this kind can be long, tedious and yet important, and may be typical cases where automatic verification is adequate.

When relating a book like this to the outside world, there is quite an amount of interpretation. As long as we have no further formalism to handle interpretation, we have to "convince" ourselves that the primitives (whether

logical, mathematical, syntactical or semantical) express what they are supposed to mean in the outside world. And we have to "convince" ourselves that the interpretations of the primitives generate interpretations of further material, and that interpretations of the final results can be obtained without bothering about the interpretation of the intermediate pieces of the book, lying between primitives and final results. And we believe that the final interpretations are mathematically correct.

This situation with computer language interpretation is more complex than with standard mathematics, but not essentially different from it. Interpretation always has to remain on a rather intuitive basis, as long as the "outside world" has not been completely formalized.

We end this paper with a short description of the AUTOMATH Project Group at the Department of Mathematics of the Technological University, Eindhoven, The Netherlands. The group has been growing slowly since 1967; early 1974 it consists of 4 full-time mathematicians (taken in the sense that includes both logicians and computer scientists), three part-time mathematicians (including the author of this paper, who leads the project), a programmer and a part-time punch-typist. We mention some of the things that have been done thus far.

(i) Language checkers have been produced, and are now available in conversational mode within the framework of a time-sharing system. Text can be fed line-by-line into the machine, which responds within at most a few seconds. If the checker refuses to accept the line, it gives complete diagnostics, which usually enables the man in charge to improve the text (possibly after consulting, over the telephone, the mathematician who produced the text). Until September 1973, the computer was the Electrologica X8, after that a Burroughs 6700. In both cases the available multiprogramming systems required the use of ALGOL 60 as the programming language.

(ii) Theoretical work on the languages of the AUTOMATH family centered around problems of normalization, strong normalization and the Church-Rosser theorem. Almost all goals have been achieved. For a detailed report one of the languages (AUT-SL) we refer to [8]. We note that there is some overlap with work of others ([5], [6], [7]) who started to interpret logic in terms of the typed lambda calculus independently, roughly at the time of the start of the AUTOMATH project.

(iii) As a kind of test-case, the work was undertaken to translate a very meticulous mathematical text into AUTOMATH. The choice fell on E.Landau's"Grundlagen der Analysis". The translation, carried out by L.S. van Benthem Jutting, is about half completed. It has not been tried to rearrange the text in order to make the translation into AUTOMATH easier, but Landau's text was followed as precisely as possible (thus getting all disadvantages and none of the advantages). It is hoped that the experience obtained will be of great help in deciding what intermediate auxiliary language should be taken for more general use. Several possibilities are being explored at the moment.

The AUTOMATH project depends very substantially on financial support by the Netherlands organization for the advancement of pure research (Z.W.O.).

References

[1] de Bruijn, N.G., "The mathematical language AUTOMATH, its usage, and some of its extensions", Symposium on Automatic Demonstration (Versailles, December 1968), Lecture notes in Mathematics, Vol.125, pp. 29-61, Springer Verlag (1970).

[2] de Bruijn, N.G., "AUTOMATH, a language for mathematics", Notes (prepared by B. Fawcett) of a series of lectures in the Séminaire de Mathématiques Supérieures, Université de Montreal, 1971.

[3] de Bruijn, N.G., "Set theory with type restrictions", International Colloquium on Infinite and Finite Sets, Keszthely, Hungary, 1973.

[4] de Bruijn, N.G., "A system for handling syntax and semantics of computer programs in terms of the mathematical language AUTOMATH", Report, Department of Mathematics, Technological University, Eindhoven.

[5] Girard, J.Y., "Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types", Proc. 2nd Scandinavian Logic Symp. (editor Fenstad), North-Holland Publishing Company, Amsterdam, 1970.

[6] Howard, W.A., "The formulae-as-types notion of construction", mimeographed, 1969.

[7] Martin-Löf, P., "An intuitionistic theory of types", unpublished, 1972.

[8] Nederpelt, R.P., "Strong normalization in a typed lambda calculus with lambda structured types", Doctoral Thesis, Technological University, Eindhoven, 1973.

Department of Mathematics
Technological University
Eindhoven, The Netherlands

Les trois articles qui suivent présentent différents développements du projet AUTOMATH.

Celui de D.T. Van Daalen descrit les aspects théoriques du langage . Il prépare celui de I. Zandleven qui présente un programme de vérification d'AUTOMATH.

Ensuite B. Jutting montre sur un exemple, comment passer d'un texte mathématique au livre AUTOMATH correspondant.

Ainsi tout le processus de vérification est décrit, et le lecteur peut se faire une idée assez précise du projet.

Acknowledgements

The authors of the next three papers in this volume wish to express their common gratitude:

- towards their colleague in the AUTOMATH project Jeff Zucker, who so patiently helped them to write correct English and who also contributed a lot of most valuable mathematical and expository comments and suggestions.
- towards Miss Marèse van den Hurk and Mrs. Yvonne Kornaat who so carefully and cheerfully typed the manuscripts.

Diederik van Daalen
Bert Jutting
Ids Zandleven

Adress of the authors:
AUTOMATH project
Department of Mathematics
Technological University
P.O. Box 513
Eindhoven, The Netherlands.

A description of AUTOMATH and some aspects of its language theory

by

D.T. van Daalen*)

0. Summary

This note presents a self-contained introduction into AUTOMATH, a formal definition and an overview of the language theory. Thus it can serve as an introduction to the papers of L.S. Jutting [7] and I. Zandleven [11] in this volume. Among the various AUTOMATH languages this paper concentrates on the original version AUT-68 (because of its relative simplicity) and one extension AUT-QE (in which most texts have been written thus far).
The contents are:

1. Introductory remarks.
2. Informal description of AUT-68.
3. Mathematics in AUTOMATH: propositions as types.
4. Extension of AUT-68 to AUT-QE.
5. A formal definition of AUT-QE.
6. Some remarks on language theory.

For a description of the AUTOMATH project and for its motivation we refer to Prof. de Bruijn's paper also in this volume [4].

*) The author is employed in the AUTOMATH project and is supported by the Netherlands Organization for the Advancement of Pure Science (Z.W.O.).

1. Introductory remarks

1.1. According to the claims for the *formal system* AUTOMATH one should be able to formalize many mathematical fields in it in such a precise and complete fashion that machine verification becomes possible. The flexibility required to meet the indicated universality is provided by having a rather meagre *basic system*. The AUTOMATH user himself has to add appropriate *primitive notions* to the basic system in order to introduce the concepts and axioms specific to the part of mathematics he likes to consider. In this respect, the basic system may be compared with some usual system of logic (e.g. first order predicate calculus) to which one adds mathematical axioms in order to form mathematical theories.

1.2. In spite of this analogy however the basic system itself does not contain any logic in the usual sense. Basic for the system are the concepts of *type* and *function* (instead of, e.g., the concept of set or of natural number), which are formalized by a certain *typed λ-calculus*.
When representing mathematics in AUTOMATH one has to deal with the question of *coding*: How to formalize general mathematical concepts in the form of *types* and *functions* (see section 2.2). Clearly an appropriate formalization will incorporate as much as possible of the basic type-and-function framework. Section 3 discusses this coding problem and in particular proposes a suitable way of representing propositions, predicates and proofs (a *functional interpretation* of logic).

1.3. In order to satisfy the claim of automatic verification of correctness the system certainly has to be decidable (and even *feasibly decidable* on now-existing computing machines). Since many common mathematical theories produce undecidable sets of theorems we must conclude that we cannot expect the computer to do all our work. Indeed theorems have to be given *together with their proofs* in order to allow verification.
Thus the correctness produced by the machine verification covers the arguments leading from axioms to conclusions only. The AUTOMATH user himself is responsible for his choice of primitive notions and all the coding (and decoding) involved.

2. Informal description of AUTOMATH

2.1. Introduction

Here we treat the original version of AUTOMATH, now named AUT-68. We chose this system as an example because of its relative simplicity. The discussion will be informal and intuitive and in fact restricted to the object-and-type fragment of the language (thus leaving the proof-and-proposition fragment to section 3).

2.2. Intuitive framework

(This section may be skipped by formalists).

The mathematical entities discussed in the language fall into two sorts: *objects* and *types*. The types may be considered as classes or sets of a certain kind, which may have objects as their elements. All types are supposed to be disjoint, for each object belongs to just one type. This *uniqueness of types* permits one to speak about *the* type of an object.

The typestructure is built up by starting from *ground types* and forming *function types* from these. Each mathematician may choose the ground types himself (as primitive notions), e.g. the type of natural numbers.

An example of a function type is the type $\alpha \to \beta$ (where α and β are types) of the functions from α to β. More generally, the function types are formed by taking *products*, as follows: The language allows one to express dependence of types on objects (of some given type). That is, one can describe certain families of types β_x indexed by the objects x of a given type α. Now every function type is formed as the *generalized Cartesian product* of such β_x, usually denoted $\prod_{x\in\alpha} .\beta_x$, and containing as objects just these functions that associate to any object x of type α an object of type β_x. The type $\alpha \to \beta$ is the special case where all β_x are a fixed type β.

2.3. Expressions, degrees and formulas; correctness

The language as such only expresses the constructions of types and objects and the typing relations between objects and types.

The *expressions* of the language have *degree* 1, 2 or 3. Types and objects are denoted by expressions of degree 2 and 3 respectively (for short 2-expressions, 3-expressions). For convenience we introduce the 1-expression type to provide a type for the types. Further 1-expressions will be introduced in sections 3 and 4.

The symbol E expresses the typing relation: ... has type ... . So if A denotes an object then we have the E-formulas A E α and α E type. The 2-expressions and 3-expressions are built up from *variables* and *constant-expressions* by means of:

i) the substitution mechanism (section 2.5)

ii) functional abstraction and application (sections 2.8 and 2.10).

The constant-expressions have the form $c(x_1,\ldots,x_k)$ where $x_1,\ldots,x_k$ are variables and c is either a *primitive* constant introduced as a primitive notion (sections 2.6) or a *defined* constant (section 2.7).
Expressions and formulas are *correct* if they are constructed according to the rules of the language, which are informally discussed in the sequel.

2.4. Variables and contexts

A mathematical statement generally presupposes certain assumptions on the variables used. For example: "let x be a natural and y a real number". In AUTOMATH, in accordance with this usage, each variable of degree 3 (*object-variable*) ranges over a certain type, called the type of the variable. The 2-variables (*type-variables*) are supposed to range through the types and have type as their type.
Expressions and formulas containing *free* object- or type-variables, say $x_1,\ldots,x_k$, can only be *correct* relative to a certain *context*: I.e. a finite sequence of E-formulas $x_1 \mathrel{E} \alpha_1,\ldots,x_k \mathrel{E} \alpha_k$, called *assumptions*, in which the free variables have to be explicitly introduced with their types.
Some of the types α_i may depend on the variables given earlier in the sequence. For instance, α_3 may contain both x_1 and x_2 as free variables. It is understood that all α_i are correct expressions themselves: α_1 relative to the *empty* context, α_2 relative to $x_1 \mathrel{E} \alpha_1$, etc.

2.5. Substitution mechanism

Let us, in informal discussion, exhibit the possible dependence of an expression Σ on variables $x_1,\ldots,x_k$ by writing $\Sigma[\![x_1,\ldots,x_k]\!]$ *for* Σ. Then we write $\Sigma[\![A_1,\ldots,A_k]\!]$ for the result of *simultaneously substituting* A_i for x_i (for $i = 1,\ldots,k$) in Σ.

Suppose that under assumptions $x_1 \underline{E} \alpha_1, \ldots, x_k \underline{E} \alpha_k$ we have a correct $\underline{E}$-formula $A[x_1,\ldots,x_k] \underline{E} \alpha[x_1,\ldots,x_k]$. Then the *substitution mechanism* yields the substitution *instance* $A[A_1,\ldots,A_k] \underline{E} \alpha[A_1,\ldots,A_k]$ for any sequence $A_1,\ldots,A_k$ of suitable candidates for $x_1,\ldots,x_k$. I.e. these $A_1,\ldots,A_k$ have to be of the appropriate types where, however, in view of the possible dependence of types on variables, the substitution has to take place in the types too. So we require

$$A_1 \underline{E} \alpha_1,\ A_2 \underline{E} \alpha_2[A_1],\ldots,A_k \underline{E} \alpha_k[A_1,\ldots,A_{k-1}] .$$

2.6. Primitive notions

As mentioned before, one has to add primitive notions to the basic system in order to introduce the specific concepts of the piece of mathematics one wants to study.

For example, in order to write about the natural numbers, one might introduce the primitive *type-constant nat* and the *object-constant* 1 by axiomatically stating:

$$\begin{aligned} &\text{nat } \underline{E}\ \underline{\text{type}} \\ &\quad 1 \ \underline{E}\ \text{nat} \end{aligned}$$

In general, primitive notions are introduced by stating an axiomatic $\underline{E}$-formula $p(x_1,\ldots,x_k) \underline{E} \alpha[x_1,\ldots,x_k]$ under certain assumptions $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$. Here either α is $\underline{\text{type}}$ (and p is a type-constant) or in the current context we have $\alpha \underline{E}\ \underline{\text{type}}$ already (p being an object-constant).

All correct substitution instances $p(A_1,\ldots,A_k)$ of such a constant-expression $p(x_1,\ldots,x_k)$ are then produced by the substitution mechanism, described above.

For example, the concept of *successor* in the natural number system can be introduced under the assumption $x \underline{E}$ nat by stating: successor(x) $\underline{E}$ nat.

Using the substitution mechanism we get

$$\begin{aligned} &\text{successor}(1) \qquad\qquad \underline{E}\ \text{nat} \\ &\text{successor}(\text{successor}(1)) \ \underline{E}\ \text{nat, etc .} \end{aligned}$$

Notice that primitive constant-expression may not only contain object-variables (like the x in successor(x)) but also type-variables.

2.7. Abbreviations

In mathematics one often introduces abbreviations, i.e. new names for possibly long and complicated expressions. In AUTOMATH this abbreviation facility is also present; indeed, it will appear that by the particular format of the language every *derived* statement gives rise to the introduction of a new defined constant. Although this kind of explicit definition is often considered theoretically uninteresting, we feel that it is essential in practice for the actual formalization and verification of complicated theories.

Just like primitive notions, abbreviations are introduced under certain assumptions and so may contain free variables in general. Thus new constant-expressions $d(x_1,\ldots,x_k)$ are introduced, abbreviating expressions D which are correct in the current context. Clearly the type of $d(x_1,\ldots,x_k)$ must be the same as that of D.

Example: 2,3,... can be introduced by

2 := successor(1)

3 := successor(2), etc .

Further, the notion of "successor of successor" might be abbreviated by stating (under assumption x $\underline{E}$ nat) that

plustwo(x) := successor(successor(x)) .

Again, all correct substitution instances with their types are produced by the substitution mechanism.

2.8. Functional abstraction: λ-calculus

We have mentioned functional abstraction and application as further tools for constructing expressions. By these devices a form of typed λ-calculus is incorporated into the basic system. In λ-calculus, intuitively speaking, λx.B denotes the function which to any object x associates the object B. Or (exhibiting the dependence on x) λx.B[x] is the map which, with any A, associates B[A].

In AUTOMATH (where all functions have a *domain*) such explicitly given functions are denoted by *abstraction expressions* [x,α]B, where B may contain x as a free variable; α is the type of x and the domain of the function. In case B is a 3-expression, [x,α]B attaches objects to the objects of type and is called an *object-valued function*. If B is a 2-expression, [x,α]B

attaches types to the objects of type α and is called a *type-valued function*. In AUT-68 no abstraction expressions of degree 1 are formed (in contrast with AUT-QE).

Notice that possible free *occurrences* of x in B are *bound* by the abstractor [x,α] and are not free in [x,α]B any more. An important restriction on abstracting is that such a bound variable must be a 3-variable. Thus we only *quantify* (cf. section 3.4) over (the objects of) a given type and quantification over type is not possible.

2.9. Type of abstraction expressions

Suppose that under the assumption x E α we have B E β. If β is not a 1-expression then we may form both the abstraction expressions [x,α]B and [x,α]β. According to section 2.8 [x,α]B denotes an object-valued function and [x,α]β denotes a type-valued function.

The latter abstraction expression [x,α]β[x] however is also used with a different meaning in Automath, that is, to denote the *corresponding function type* $\prod_{x E \alpha} .\beta[x]$ (which is the type of [x,α]B[x] by section 2.2).

So we obtain [x,α]B E [x,α]β and [x,α]β E type.

Example: the successor *function* can be introduced (in the empty context) by

succfun := [x,nat]successor(x) E [x,nat]nat .

The double use of 2-expressions mentioned above does not cause ambiguity, because it is always clear whether an expression acts as a function or as a type in a formula. In fact in AUT-68 abstraction expressions of degree 2 are exclusively used with the second meaning, i.e. as function types.

2.10. Functional application

In full (i.e. type-free) λ-calculus any expression - as a function - may be applied to any expression - even itself - as an argument.

In AUTOMATH, as a *typed* λ-calculus, all functions have *domains* and any form of self-application is ruled out by the *application restrictions*: The *application expression* <A>B (denoting the result of applying B as a function to A as an argument) is correct only if:

i) B is a function and so has a domain, say α.

ii) A is an object of type α.

The notation <A>B, with the argument in front, is somewhat unusual; it is convenient however since abstractions are written in front too.

2.11. Type of application expressions

Assume that B $\underline{E}$ [x,α]β. Here [x,α]β[x] is a 2-expression acting as a type and so denotes $\Pi_{x\underline{E}\alpha}.\beta[x]$. Hence B must be considered as a function with domain α. Now if A $\underline{E}$ α we are allowed to form the application expression <A>B having β[A] as its type.

Note that B need not be of the form [x,α]C itself. It may, e.g., be a single object variable or object constant with type [x,α]β.

Example: As an alternative expression for the number 3 we might introduce

3alt := <2>succfun $\underline{E}$ nat .

2.12. Equality

We will define a relation of *definitional equality* among the correct expressions, appropriate to the interpretation of expressions suggested above. The relation is denoted ... = ... and generated by:

i) *abbreviational* or δ-*equality*, $=_\delta$

ii) λ-*equality*.

The latter is generated in turn by β-*equality*, $=_\beta$, and η-equality, $=_\eta$. Usually in λ-calculus the λ-equality also explicitly embodies α-*equality* (renaming of bound variables). In this note however we take the point of view of simply ignoring the names of the bound variables. So α-equal expressions are identified and are a fortiori definitionally equal by the reflexivity of the = -relation (cf. also section 5.3.2).

2.12.1. δ-equality

Assume the defined constant d has been introduced in suitable context by

$$d(x_1,\ldots,x_k) := D[x_1,\ldots,x_k] .$$

Then $d(x_1,\ldots,x_k)$ abbreviates D and we write $d(x_1,\ldots,x_k) =_\delta D$. And further for the substitution instances:

$$d(A_1,\ldots,A_k) =_\delta D[A_1,\ldots,A_k] .$$

2.12.2. β-equality

Assume <A>[x,α]B[[x]] is a correct expression (so A $\underline{E}$ α). Now β-equality exploits the interpretation of [x,α]B as a function with domain α and simply amounts to evaluating the result of the application:

$$\langle A\rangle[x,\alpha]B =_{\beta} B[[A]] .$$

2.12.3. η-equality

In mathematics one usually considers functions as *extensional* objects, in the sense that functions with the same domain and which are pointwise equal are identified. In AUTOMATH this extensional equality is *partly* covered by the η-equality: *If* x *does not occur free in* B *then* [x,α]<x>B $=_{\eta}$ B (for correct expressions only). This is intuitively sound only if domain B = α, which indeed is the case by the correctness of [x,α]<x>B.

2.12.4. Definitional equality

Now definitional equality = is defined to be the *equivalence relation* on the correct expressions, generated by $=_{\delta}$, $=_{\beta}$, $=_{\eta}$ and by *monotonicity*: *If* A = A' *and* B' *is produced from* B *by replacing one specific occurrence or* A *in* B *by* (*an occurrence of*) A' *then* B = B'.

Or, using *suggestive dots* for the *unchanged* part of the expression B: *If* A = A' *then* ... A ... = ... A' ... .

Example of the monotonicity rule: If A = A' then <C><A>D = <C><A'>D (if both expressions are correct).

2.13. The format: books and lines

2.13.1. Actual AUTOMATH texts are written in the form of books. A *book* consists of a finite sequence of *lines*. Each line must be placed in a certain *context* (the context of the line) and introduces a new identifier of a certain type. All lines consist of four consecutive parts, separated by suitable marks or spaces:

i) *context part*, indicating the context of the line. In general the context part consists of the *context indicator*, i.e. the last variable of the current context. From this the complete context can easily be recovered. If the context of the line is $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$, the sequence of variables $x_1,\ldots,x_k$ is called the *indicator string* of the line. The *empty* context can be indicated by an empty context part.

ii) *identifier part*, consisting of the new *identifier*.

iii) *middle part*, containing the symbol EB (cf. 2.13.2), the symbol PN (cf. 2.13.3) or the *definition* of the new identifier (cf. 2.13.4).

iv) *category part*, containing the type of the new identifier.

Assume an AUTOMATH book is given, in which the variable x_k has been introduced with type α_k in the context $x_1 \underline{E} \alpha_1,\ldots,x_{k-1} \underline{E} \alpha_{k-1}$. Then we may add lines with context indicator x_k, so having $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$ as their context. Below we discuss the three different kinds of lines.

2.13.2. The *block opening lines* have middle part EB (for *empty block opener*) or, in alternative notation, a bar ———. An EB-line introduces a new *variable* and thus allows extension of the current context by one assumption.
Example: $x_k * y := \underline{EB}\ \underline{E}\ \alpha$ ("let y be of type α") introduces a new variable y of type α. Lines having y as their context part - which may appear later in the book - then have $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$, $y \underline{E} \alpha$ as their context.

2.13.3. The *primitive notion lines* have middle part PN and introduce the primitive notions. For example:

$$x_k * p := \underline{PN}\ \underline{E}\ \alpha$$

introduces the primitive constant expression $p(x_1,\ldots,x_k)$ and contains the *axiomatic* E-statement $p(x_1,\ldots,x_k) \underline{E} \alpha$.

2.13.4. The *abbreviation lines* look like:

$$x_k * d := D\ \underline{E}\ \alpha\ ,$$

where the middle part D is the definition of d, i.e. the expression to be abbreviated. This line contains, relative to the preceding book and the current context, both the derived E-statement $D \underline{E} \alpha$ and the defining axiom for the new defined constant d:

$$d(x_1,\ldots,x_k) := D\ .$$

2.14. Correctness of lines; validity

A line is *correct* if both the middle part (if not EB or PN) and the category part are correct expressions with respect to the preceding book and the current context, and the category part is the type of the middle part (if not EB or PN). For the correctness of the expressions, all identifiers used have to be *valid*. Constants are valid in a book from the line on in which they are introduced. Free variables are valid in a line if they occur in its context. We speak about the *block* of lines in which a free variable is valid (whence *block opener*).

2.15. Shorthand facility

Assume that a primitive or defined constant c was introduced in a certain context $x_1 \underline{E} \alpha_1,\ldots,x_k \underline{E} \alpha_k$. Then if later in the book c occurs with fewer than k arguments, the argument list is completed by adding a suitable initial segment of the original indicator string (cf. 2.13.1ii)) $x_1,\ldots,x_k$. In other words the expression $c(A_{i+1},\ldots,A_k)$ is shorthand for $c(x_1,\ldots,x_i,A_{i+1},\ldots,A_k)$ and the single constant c is shorthand for $c(x_1,\ldots,x_k)$. Clearly the completing variables have to be valid, that is, the initial segments of the original and the current context have to coincide. The shorthand facility accords with usual mathematical practice where free variables are often considered as fixed throughout an argument and are not mentioned explicitly.

2.16. Paragraph system

For each variable and constant it must be possible to retrace from which line it originates. This condition is clearly satisfied when all names are unique. A more liberal method of naming however is allowed by the socalled *paragraph system*, for a description of which we refer to Zandleven [11, section 11]. Both shorthand facility and paragraph system do not really concern the language definition but are present for convenience only.

2.17. Example

In the following AUT-68 booklet the examples of the preceding sections are now written in the proper format.

```
  * nat        := PN                    type
  * 1          := PN                    nat
  * x          := ——                    nat
x * successor  := PN                    nat
  * 2          := successor(1)          nat
  * 3          := successor(2)          nat
x * plustwo    := successor(successor)  nat
  * succfun    := [x,nat]successor(x)   [x,nat)nat
  * 3alt       := <2>succfun            nat
```

Here the middle part of plustwo uses the shorthand facility. It is left to the reader to establish 3 = 3alt.

3. Mathematics in AUTOMATH: Propositions as types

3.1. Functional interpretation of logic

Up till now we have described AUTOMATH as a calculus of objects and their types only. A major part of mathematics however consists of making statements and reasoning with them, i.e. deals with logic.

Now there are different ways of coding some logic into the objects-and-types framework. Here we only mention a socalled *functional interpretation* of logic, which gives rise to the *propositions-as-types* notion. This idea of interpreting logic was developed independently by de Bruijn and certain others, of whom we mention Howard [6], Prawitz [10], Girard [5] and Martin-Löf [8].

3.2. Propositions as types

So far we have introduced type as the only 1-expression. We had Σ E type and Γ E Σ for the types Σ and the objects Γ of type Σ respectively. Now we introduce another 1-expression, the basic symbol prop. Originally in AUT-68 no distinction was made between type and prop. The latter 1-expression acts just like type and was introduced later to allow difference of treatment between types which are to be considered as propositions and types which are just types of objects.

If Σ E prop we consider Σ as a proposition. If further Γ E Σ, we consider Γ as some construction establishing the truth of Σ (a "proof" of Σ). Thus the formula Γ E Σ is conceived as *asserting* the proposition Σ.

3.3. Interpreting implication

Let α E prop and β E prop. Now we may say we have a "proof" of the implication $\alpha \to \beta$ if from an assumption of the truth of α we can argue and conclude the truth of β. That is, if for any construction establishing the truth of α we can produce a construction for the truth of β or, equivalently, if we have a map from "proofs" of α to "proofs" of β.

Now in AUTOMATH terminology: we say we "prove" $\alpha \to \beta$ if for any x E α we can produce some B E β. I.e. if we have some Σ in the function type $[x,\alpha]\beta$. So we let $[x,\alpha]\beta$ denote the implication $\alpha \to \beta$ and have $[x,\alpha]\beta$ E prop. This corresponds to the second interpretation of abstraction expressions in section 2.9.

Now by this interpretation we obtain the *modus ponens* (from α and $\alpha \to \beta$ infer β) by simple functional *application*. For let $A \underline{E} \alpha$ and $\Sigma \underline{E} [x,\alpha]\beta$ (A and Σ thus being "proofs" of α and $\alpha \to \beta$ respectively). Then by the application rule we construct $\langle A \rangle \Sigma$ establishing the truth of β.

3.4. Universal quantification; negation

In exactly the same manner a function interpretation of *universal* statements can be given. Namely if $\alpha \underline{E}$ type and for $x \underline{E} \alpha$ we have $\beta \underline{E}$ prop then we identify the function type $[x,\alpha]\beta$ with the universal statement $\forall_{x \underline{E} \alpha} \beta$. Here functional application corresponds to the "*instantiation*" *rule* in logic.
Thus by this interpretation of logic in AUTOMATH one gets the $(\forall,\to)$-fragment of first order predicate logic for free. However in AUTOMATH only positive statements are made and statements like: "Σ is not of type Γ" cannot be expressed. In order to interpret negation we introduce as a primitive notion the proposition *con* (for "contradiction") together with some suitable axiom (primitive notion). Here are different possibilities, e.g. the intuitionistic *absurdity rule* (for any proposition α, from con infer α) or the classical *double negation law*. Then an AUTOMATH theory (i.e. book) is *consistent* if, in the empty context, it does not produce some $\Sigma \underline{E}$ con.
For $\alpha \underline{E}$ prop we define *non*(α) as $a \to$ *con* or, in AUTOMATH notation, $[x,\alpha]$*con*.
Now the double negation law can be stated by introducing the primitive notion *dnl* as follows: *If* $\alpha \underline{E}$ prop, $x \underline{E}$ *non*(*non*(α)) *then* *dnl*(α,x) $\underline{E}$ α.
By also choosing suitable definitions for the other connectives $(\wedge,\vee)$ and the existential quantifier we can smoothly obtain full classical first order predicate calculus.

3.5. Assumptions, axioms, theorems

In AUTOMATH-books the $\underline{E}$-formula $\Gamma \underline{E} \Sigma$ for proposition Σ can occur in the usual three kinds of lines again:

i) $\underline{EB}$-lines: $\sigma * x := \underline{EB}\ \underline{E}\ \Sigma$.
These must be interpreted as *assumptions*: "let Σ hold" or "let x be a proof of Σ". Now in a line where x is valid we may refer to x whenever we want to use the assumed truth of Σ.

ii) $\underline{PN}$-lines: $\sigma * p := \underline{PN}\ \underline{E}\ \Sigma$.
These serve as axioms, or rather as axiom *schemes* (by the dependence on the variables contained in the context σ).

iii) abbreviation lines: $\sigma * d := \Gamma \underline{E} \Sigma$ must be considered as derived statements, i.e. theorems, lemmas etc. Here the middle part Γ "proves" the proposition Σ from the assumptions in the context σ.

3.6. Book-equality

The definitional equality (cf. section 2.12) of AUTOMATH only covers a small part of the usual mathematical equality. Further a statement of definitional equality cannot be handled as an actual proposition; e.g. it cannot be negated or even assumed (as in: let A = B). As the AUTOMATH-counter part of the usual mathematical ... equals ..., the *book-equality* IS(α,A,B) - where A and B are objects of type α - can be introduced by suitable primitive notions, some of which are shown in the example below.

* α	:= ———	type
α * x	:= ———	α
x * y	:= ———	α
y * IS	:= PN	prop
x * REFL	:= PN	IS(x,x)
y * i	:= ———	IS(x,y)
i * SYM	:= PN	IS(y,x)
	etc.	

and also:

α * β	:= ———	type
β * f	:= ———	[x,α]β
f * x	:= ———	α
x * y	:= ———	α
y * i	:= ———	IS(x,y)
i * ISAX1	:= PN	IS(β,<x>f,<y>f)

By the axiom of reflexivity (REFL) above, definitional equality implies book-equality: if A $\underline{E}$ α, B $\underline{E}$ α, A = B then REFL(α,A) $\underline{E}$ IS(α,A,B).

4. Extension of AUT-68 to AUT-QE

4.1. Function-like expressions

Expressions Σ such that Σ E $[x,\alpha]\beta$ or $\Sigma = [x,\alpha]\beta$ are called *function-like* expressions. Whereas in AUT-68 function-like 3-expressions may have any form, e.g. they can be variables or primitive constant expressions, the only function-like 2-expressions are (possibly abbreviated) abstraction expressions. This is because function-like 1-expressions are absent in AUT-68.
Thus we can discuss explicitly constructed families of types β_x where x ranges over some type α (namely by forming the abstraction expression $[x,\alpha]\beta[\![x]\!]$) but we cannot discuss *arbitrary* families of types indexed by x E α. Indeed, we cannot introduce a family of types as a primitive notion or as a variable.

4.2. Supertypes or quasi-expressions

In AUT-QE on the other hand such arbitrary type-valued functions are admitted however, by extending the class of 1-expressions. The new 1-expressions, *quasi-expressions* (whence AUT-*QE*) or *supertypes*, have the form $[x_1,\alpha_1] \ldots [x_k,\alpha_k]$ type or $[x_1,\alpha_1] \ldots [x_k,\alpha_k]$ prop, where $\alpha_1,\ldots,\alpha_k$ are 2-expressions, i.e. propositions or types.
For example, an arbitrary type-valued function on α can be introduced by an EB-line:

$$\sigma * f := \text{——} [x,\alpha]\text{type} .$$

If for α we take the type of natural numbers, then f is an arbitrary *sequence of types*.

4.3. The use of AUT-QE

Similarly we have arbitrary *prop-valued functions* in AUT-QE. These are especially useful in our interpretation of logic, for a prop-valued function with domain α is nothing but a *predicate* over α. For example, by an EB-line

$$\sigma * R = \text{——} [x,\text{nat}][y,\text{nat}]\text{prop}$$

an arbitrary binary predicate (rather: relation) on the natural numbers is introduced. The presence of predicate and relation variables in AUT-QE allows us to write *axiom schemes* with such variables, e.g. to introduce a further equality axiom (cf. section 3.6) we can write:

α * P := ——— [x,α]prop

P * x := ——— α

x * y := ——— α

y * i := ——— IS(x,y)

i * y := ——— <x>P

j * ISAX2 := PN <y>P

We emphasize however that abstraction over such 2-variables (e.g. type-variables, prop-variables, predicate-variables) in AUT-QE is still forbidden, so both AUT-68 and AUT-QE may still be called *first-order systems*.

4.4. Type-inclusion and prop-inclusion

Just as in AUT-68 the function-like 2-expression f (cf. section 4.2) also codes its corresponding function space, i.e. the type of those g with domain α such that for A E α we have <A>g E <A>f. As prop behaves just like type, the predicate P (cf. section 4.3) also denotes the proposition $\forall_{x \underline{E} \alpha}.P(x)$.

As a consequence, we allow the transition from Σ E [x,α]type to Σ E type. This transition or, in general, from

$$\Sigma \ \underline{E} \ [x_1,\alpha_1] \ \ldots \ [x_k,\alpha_k][y_1,\beta_1] \ \ldots \ [y_m,\beta_m]\underline{type}$$

to

$$\Sigma \ \underline{E} \ [x_1,\alpha_1] \ \ldots \ [x_k,\alpha_k]\underline{type}$$

is called *type-inclusion*. The similar transition with prop instead of type is called *prop-inclusion*. By this *type-inclusion* and *prop-inclusion* AUT-QE contains AUT-68 as a proper *subsystem*. Notice that for 2-expressions uniqueness of types - if A E α, A E β then $\alpha = \beta$ - is lost.

4.5. Let us finish with a table in which some AUTOMATH notions are listed with their possible meanings in the propositions-as-types interpretation.

AUTOMATH-notions	object-and-type interpretation	proof-and-proposition interpretation
2-expressions	types	propositions
3-expressions	objects	proofs
... E ...	... has type ...	... proves ...
function-like 2-expressions	type-valued functions	predicates
	function types	implications
		universal statements
EB-lines	variable introductions	assumptions
PN-lines	primitive object introductions	axioms
abbreviation lines	definitions or abbreviations	theorems

5. A formal definition of AUT-QE

5.1. The language, to be defined formally now, is the one accepted by the current checker (cf. [11]) except for two points:

i) Paragraph facilities are not present here so all constant names have to be distinct (cf. section 2.16).

ii) There is no shorthand facility (i.e. all expressions are written out in full (cf. section 2.15).

The actual formalism has been chosen in this way in order to keep as close as possible to the preceding informal book-and-line description. A definition along more usual *natural deduction* lines may possibly be more elegant. For technical reasons we preferred to avoid redundancy almost completely in our definition. As a consequence of this, some useful extra rules follow as *derived rules* in the section on language theory.

5.2. Our aim is to define formally what correct AUT-QE books are.
The description consists of:

i) Preliminaries, mainly devoted to the context free part of the language (section 5.4).

ii) *Simultaneous* definition of correctness of books, contexts, lines, expressions, E-formulas and = - formulas (section 5.5).

The = - formulas only serve as a help in our definition; they do not appear in the book. The kernel of ii) is the definition of correctness of expressions and formulas relative to a certain book and context. Here the book serves to determine the set of primitive notions and abbreviations, and the context serves to determine the set of valid free variables.
Most concepts are introduced by *ordinary inductive definitions*. These consist of a finite set of rules of the form: "if ... then ...". Here only such conclusions may be drawn which follow from a finite number of applications of the rules.

5.3. Notational conventions

5.3.1. An extensive use is made of *syntactic variables* throughout the definition. Often certain assumptions on these variables are implicit by their specific choice, e.g. σ and ξ always run over contexts. Syntactic variables may always be indexed or primed.

5.3.2. As for substitution and α-*conversion* (renaming of bound variables) we adopt the following point of view: expressions with bound variables are considered as named versions - named to facilitate reading - of some actually *namefree* skeleton (cf. [3]). Thus we identify α-equal expressions and assume that α-conversion is applied whenever necessary to avoid *clash of variables*. We use ... ≡ ... to denote *syntactic identity* (symbol-for-symbol equality) modulo α-equality. E.g. [x,Σ] ...x...x... ≡ [y,Σ] ...y...y... .

5.3.3. Correctness of expressions A and formulas φ relative to a book $\mathcal{B}$ and a context σ are abbreviated by $\mathcal{B}$; σ ⊢ A and $\mathcal{B}$; σ ⊢ φ respectively. Sometimes we write ⊢ A or σ ⊢ A for $\mathcal{B}$; σ ⊢ A and ⊢ φ or σ ⊢ φ for $\mathcal{B}$; σ ⊢ φ when there is no particular need to emphasize the current book or context. The notions $\vdash^{(i)}$A and $\vdash^{(i)}$A E B are used to express that A is an i-expression and ⊢ A : (respectively ⊢ A E B).

5.4. Preliminaries

5.4.1. Alphabet

1) As *variables* and *constants* we allow any *alphanumeric string*. Such a string is considered atomic and is thus counted as one single symbol. Syntactic variables for variables are x,y,z,... Among the constants (syntactic variable c) we distinguish *primitive* (syntactic variables p,q) and *defined* or *abbreviational* constants (syntactic variable d).
2) Improper symbols
   i) Some *brackets* and *braces*: [,], (,), < , >.
   ii) Some *separation marks*: !, *, ⊢, E, :=, =, *semicolon* and *comma*.
   iii) Some *reserved symbols*: EB, PN.

5.4.2. Expressions (syntactic variables A,B,C,D,...,Σ,Δ,Γ,...)

i) *Variables*: x
ii) *Abstraction expressions*: [x,Σ]Δ
iii) *Applications expressions*: <Σ>Δ
iv) *Constant-expression instances*: $c(\Sigma_1,\ldots,\Sigma_k)$
v) *Basic constants*: type, prop.

As special syntactic variables for 2-*expressions* we take α,β,... .

5.4.3. Formulas (syntactic variable φ)

i) $\underline{E}$-*formulas*: $\Sigma\ \underline{E}\ \Delta$

ii) $=$-*formulas*: $\Sigma = \Delta$.

5.4.4. Additional concepts

1) *Contexts* (syntactic variables σ,ξ): *Any finite (possibly empty) sequence of* $\underline{E}$-*formulas* $x_i\ \underline{E}\ \Sigma_i$, *separated by commas, where all* x_i *are different.*

2) Lines (syntactic variable λ)

i) $\underline{EB}$-*lines* : $\sigma * x := \underline{EB}\ \underline{E}\ \Sigma$

ii) $\underline{PN}$-*lines* : $\sigma * p := \underline{PN}\ \underline{E}\ \Sigma$

iii) *Abbreviation lines*: $\sigma * d := \Delta\ \underline{E}\ \Sigma$

3) *Books* (syntactic variable B): *Any finite (possibly empty) sequence of lines, separated from one another by exclamation signs* (!).

5.4.5. Free variables

We define the *free variable* set $FV(\Sigma)$ of expressions Σ by induction on the structure of Σ (cf. section 5.4.2):

i) $FV(x) = \{x\}$

ii) $FV([x,\Gamma]\Delta) = FV(\Gamma) \cup (FV(\Delta)\backslash\{x\})$

iii) $FV(<\Gamma>\Delta) = FV(\Gamma) \cup FV(\Delta)$

iv) $FV(c(\Sigma_1,\ldots,\Sigma_k)) = \bigcup_{i=1,\ldots,k} FV(\Sigma_i)$

v) $FV(\underline{prop}) = FV(\underline{type}) = \emptyset$.

5.4.6. Substitution

1) The result of *simultaneous substitution* of $A_1,\ldots,A_k$ for the free variables $x_1,\ldots,x_k$ in an expression Σ is denoted by $[\![x_1,\ldots,x_k/A_1,\ldots,A_k]\!]\Sigma$ and locally abbreviated by Σ^*:

i) $x_i^* \equiv A_i$

ii) $y^* \equiv y$ if y not among $x_1,\ldots,x_k$

iii) $([y,\Sigma_1]\Sigma_2)^* = [y,\Sigma_1^*]\Sigma_2^*$ if y not among $x_1,\ldots,x_k$ and $(x_i \in FV(B) \Rightarrow y \notin FV(A_i))$ for $i = 1,\ldots,k$ (otherwise rename y in $[y,\Sigma_1]\Sigma_2$).

7. References

[1] De Bruijn, N.G.; *The mathematical language AUTOMATH, its usage and some of its extensions.* Symposium on Automatic Demonstration (Versailles December 1968), Lecture Notes in Mathematics, Vol. 125, pp. 29-61, Springer-Verlag, Berlin, 1970.

[2] De Bruijn, N.G.; *Automath, a language for mathematics;* notes (prepared by B. Fawcett) of a series of lectures in the Séminaire de Mathématiques Supérieures, Université de Montréal, 1971.

[3] De Bruijn, N.G.; *Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem,* Indag. Math., 34, No. 5, 1972.

[4] De Bruijn, N.G.; *The AUTOMATH Mathematics Checking Project,* this volume.

[5] Girard, J.Y.; *Interpretation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur,* Doctoral dissertation, Université Paris VII, 1972.

[6] Howard, W.A.; *The formulae-as-types notion of construction,* unpublished 1969.

[7] Jutting, L.S. van Benthem; *The development of a text in AUT-QE,* this volume.

[8] Martin-Löf, P.; *An intuionistic theory of types,* unpublished 1972.

[9] Nederpelt, R.P.; *Strong normalization in a typed lambda-calculus with lambda-structured types,* Doctoral dissertation, Technological University, Eindhoven, 1972.

[10] Prawitz, D.; *Ideas and results in proof theory,* in: Proc. 2nd. Scandinavian Logic Symp., North-Holland Publ. Comp., Amsterdam, 1971.

[11] Zandleven, I.; *Verifying program for AUTOMATH,* this volume.

A Verifying Program for AUTOMATH

by I. Zandleven *)

0. Summary

This paper describes the AUTOMATH verifier which is currently being operated at the Technological University at Eindhoven.
The description is given in terms of a number of procedures, written in an ALGOL-like language.
The contents are:

1. General remarks.
2. The description language.
3. The translator.
4. Some basic notions and procedures.
5. Substitution.
6. Reductions.
7. CAT and DOM.
8. Definitional equality.
9. Correctness of expressions.
10. Correctness of lines.
11. A paragraph system.
12. Final remarks.

For the theoretical background we refer to the papers of Prof. de Bruijn D. van Daalen and R. Nederpelt: [1], [2], [3] and [6].

*) The author is employed in the AUTOMATH project and is supported by the Netherlands Organization for the Advancement of Pure Science (Z.W.O.).

1. General remarks

1.1. The aim of this paper is to give a rough description of how the AUT-68 and AUT-QE verifier is constructed and how it works. Most of the procedures are much simplified for the sake of clarity and so as not to bother the reader with topics like memory organization, error messages etc.

1.2. The whole verifier is embedded in a conversational system (operating via a terminal) in order to control the amount of work the program might do in certain cases (mostly when an error in the AUTOMATH text has been made). The parts of the procedure texts, whose execution is (partly) controlled by human intervention, are placed between the brackets ?(and)? .
Furthermore there is the opportunity to the user to debug the text on-line.

1.3. Notations

1.3.1. Expressions are denoted by A,B,...,A1,A2,...etc.

1.3.2. Syntactical identity is denoted by $\equiv$

1.3.3. Bound variables in abstraction expressions are denoted by x,y,...; thus e.g. [x,A]B.

1.3.4. Expressionstrings are denoted by Σ, Γ,...

1.3.5. An expression, occurring in an expressionstring Σ is denoted by Σ with a subscript; thus $\Sigma \equiv (\Sigma_1,\ldots,\Sigma_n)$, where Σ_i are the expressions occurring in Σ $(i=1,\ldots,n)$.

1.3.6. Each non-empty string Σ can be divided into two parts:

Σ^+ := the last expression of Σ

Σ^- := the rest of Σ (which may be empty).

Example:
If Σ= A,B,C(D,E), F(G,H) then
Σ^+ = F(G,H), Σ^- = A,B,C,(D,E).

1.3.7. The composition of a string is denoted by the parenthesis ((and)) e.g.: $\Sigma \equiv ((\Sigma^+, \Sigma^-))$.

1.3.8. An indicator string [3,§2.13] is denoted by I, and a context [3,§2.2] by α.

1.3.9. Sometimes, in theoretical discussions, the notation of D. van Daalen is used [3,§5.3].

2. The description language

2.1. The language used for the description of the verifying procedures, is based upon ALGOL '60.

2.2. Several types (in the sense of ALGOL '60) are added, e.g. expression, defined name, etc.

2.3. A construction case....of begin.... end is added, to avoid repeated if.. then.. else-constructions. The values of the case selector are placed before the entries, as labels.

Examples:

The statement

```
if color= red then paint (river valley)
if color= white then paint (Christmas) else
if color= blue then paint (moon) else paint (nothing),
```

may now be written as:

```
case color of
begin
    red: paint (river valley);
    white: paint (Christmas);
    blue: paint (moon);
    otherwise: paint (nothing);
end;
```

Another possibility is:

```
paint (case color of
       begin
          red: river valley;
          white: Christmas;
          blue: moon;
          otherwise: nothing;
       end);
```

So the case-construction may be used for both statement selection and assignment selection.

2.4. Some non-ALGOL symbols are used, e.g. ⊢, Ø,..., and sometimes procedure identifiers are defined as infix, e.g. d OLDER THAN b would be written OLDERTHAN(d,b) in correct ALGOL.

2.5. Each procedure, whose identifier is written in capitals or non-ALGOL symbols, is explained.

2.6. No use is made of the parameter device: value. If an actual parameter has to be evaluated, this is done once only at the beginning of the procedure. All further calls are calls by reference to a program variable.

3. The translator

Before AUTOMATH texts are presented with the verifier, they are passed through a translator. One may consider this translator as a pre-processor, checking the context-free part of the AUTOMATH syntax (parentheses, commas etc.), coding the identifier-paragraph identification (see §11), completing the expressions written in shorthand, etc.

4. Some basic notions and procedures

4.1. Shapes

Most of the procedures must be able to distinguish the different characteristic forms in which expressions appear.

For this purpose we introduce the notion *shape*, which represents the outhermost characteristic form of an expression.
E.g. the expression:

<A(B)>C([x,D]E)

has the "application shape", symbolically denoted bv an application expression such as <P>Q or $<E_1>E_2$.

4.1.1. The shapes, and their symbolism, which are used, are:

shape	symbolism
type	type
prop	prop
variable	variable
bound variable	boundvar
constant shape	d(Σ)
application shape	<A>B
abstraction shape	[x,A]B

4.1.2. When using this symbolism for the shapes, we will permit ourselves to use the sub-elements of it, as expressions on which to operate (without explicit declaration of and assigment to the program variables). So we may write, for example:
if shape (E)= [x,A]B then domain:=A else...

4.2. Primitive procedures
Often, during the verificaton process of a book B we need the indicator string, the middle expression or the category expression of a certain line of B. Each line in the book B is uniquely indicated by the name introduced in the identifier part of that line (possibly with a paragraph reference, see §11). These names will belong to the ALGOL-type definedname.
Because an indicator string may be considered as a string of expressions, we may introduce the

4.2.1. expressionstring procedure INDSTR (d);
definedname d;
comment INDSTR becomes the indicator string of the line in which d is defined;

For the middle and category expression procedures:

4.2.2. expression procedure MIDDLE (d);
definedname d;
comment MIDDLE becomes the middle expression of the line in which d is defined. Of course this procedure is only allowed for those d which represent an abbreviation.

4.2.3. expression procedure CATEGORY (d);
definedname d;
comment CATEGORY becomes the category expression of the line in which d is defined (both for EB lines, PN lines and abbreviations);
The bodies of these procedures cannot be explained without going into details of memory organization, a subject which is beyond the scope of this note.

4.2.4. Another primitive procedure, OLDER THAN , will be explained in §8.2 .

5. Substitution

5.1. We have introduced two different shapes (and codings) for variables to be able to distinguish properly between all the variables occurring in an expression.
By "shape=variable" we code the variables which occur in indicator strings (these variables are sometimes called *parameters*).
By "shape=boundvar" we code the variables which occur in abstractors.
Futhermore, in one AUTOMATH book, all binding variables (i.e. variables occurring as x in [x,A]...) get different code-numbers. So the substitution becomes a simple replacement operation.
Now there is only one possible way to get a so-called *clash of variables*, namely in the following example.
Suppose we have an expression like

[x,A](...,<B(x)>[y,C][x,A]D(y),...).

If we want to reduce the expression between the dots : (by β-reduction), we will obtain the expression

[x,A]D(B(x))

and we see that the x in D(B(x)) is bound by the wrong abstractor now.
It is claimed by the author that by this coding system no clash (conflict, confusion) of variables arises during the verification process of AUTOMATH.

5.2. Substitution for free variables

At first we define a procedure SUBST, which will replace free variables (shape=variable) by expressions, as follows:
Let v be a string of free variables (mutually distinct),
let Γ be an equally long string of expressions,
let E be an expression.
The procedure SUBST constructs a new expression by replacing in E all v_i by the corresponding Γ_i.
The procedure (function) identifier SUBST will become the resulting expression: $[\![v_1,\ldots,v_n/\Gamma_1,\ldots,\Gamma_n]\!] E$ (see [3] for this notation).
We call this procedure by e.g.

SUBST(v,Γ,E)

The string analogue of SUBST(v,Γ,E) , STRINGSUBST(v,Γ,Σ) means: replace, in all Σ_j, all v_i by the corresponding Γ_i .

5.2.1. Procedure text

5.2.1.1. expression procedure SUBST (v,Γ, E);

```
expression E; expressionstring v,Γ;
comment shape (vi) must be variable;
SUBST:=
case shape (E) of
begin
   variable  : if ∃i0:=i (vi≡E) then Γi0 else E ;
   d(Σ)      : d(STRINGSUBST (v,Γ,E)) ;
   <A>B      : <SUBST(v,Γ,A)>SUBST(v,Γ,B) ;
   [x,A]B    : [x,SUBST (v,Γ,A)]SUBST(v,Γ,B) ;
   otherwise : E ;
end;
```

5.2.1.2. expressionstring procedure STRINGSUBST(v,Γ,Σ);

```
expressionstring v,Γ,Σ;
comment shape (vi) must be : variable;
   STRINGSUBST is the string-analogue of SUBST;
STRINGSUBST:= if Σ ≡ ∅ then ∅
              else (( STRINGSUBST(v,Γ,Σ⁻), SUBST(v,Γ,Σ⁺) ))
```

5.3. Substitution for bound variables (shape=boundvar)

This is like the substitution for free variables (apart from the fact that only one boundvar at a time is substituted for). Therefore we will only give the procedure heading.

5.3.1. expression procedure BOUNDSUBST(x,A,E),
boundvar x; expression A,E;
comment A is either an expression or another boundvar to substitute for x in E_j .

6. Reductions

The reductions involved in the verification of correctness of =- formulas (cf. §8) are α-, β-,η- and δ-reduction. See also [3, §2.12 and §6.2].

6.1. α- reduction

To perform an α-reduction one can easily use the procedure BOUNDSUBST. For an expression [x,A]B, where x is to be replaced by y (say), we have simply to construct

[y,A]BOUNDSUBST(x,y,B)

(y must be new of course).

6.2. β-reduction

The β-reductor is written in the form: $A \gtrsim_\beta B$, where $\gtrsim_\beta$ represents a boolean procedure with two parameters, A and B.

A typical use of this procedure is e.g.

if $E_1 \gtrsim_\beta E_2$ then A:=E_2 else...

If a β-reduction is applicable to A (so $A \equiv <A_1>[x,A_2]A_3$) then B becomes $[x/A_1]A_3$, and the procedure identifier gets the value true.

If A has the form $<A_1>A_2$ where A_2 does not have an abstraction shape, so that no direct β-reduction is possible, then the procedure tries to reduce A_2 with β-and/or δ-reduction so as to obtain the form $[x,A_3]A_4$. At that point the actual β-reduction can be carried out.

6.2.1. Procedure text

6.2.1.1.
```
boolean procedure   A≥β B,
expression   A,B;
comment   if A is reducible by β-reduction, then B becomes the β-reduct
          of A;
begin
   if shape(A)=<P>Q then
   begin boolean possible;
      possible:= shape(Q)= [x,R]T;
      if not possible then
      begin boolean continue; continue:= true;
         while continue do
         ?(begin case shape(Q) of
                begin
                   <R>S: continue:= Q≥βU
                   d(Σ): continue:= Q≥δU
                   otherwise: continue:= false;
                end;
             if continue then
             begin Q:=U; possible:=shape(Q)= [x,R]T;
                continue:=not possible;
             end;
         end)?;
      end;
      if possible then
      begin B:=BOUNDSUBST(x,P,T); ≥β:=true;
      end
      else ≥β:=false;
   end
   else ≥β:=false ;
end;
```

6.3. η-reduction

The whole procedure runs under control of the boolean "etareduction allowed", which may be set or reset by the user. When reset (etareduction allowed=false), the verificator can only use α-, β- and δ-reduction. Interestingly enough, in the AUTOMATH texts, checked so far, η-reduction has almost never been been used.

The η-reductor is written in the same form as the β-reductor: $A \geq_{\eta} B$.

We have for A the following cases.

i) $A \equiv [x,P]\langle Q\rangle R$.

a) If $Q \not\geq x$ then the procedure first tries to reduce $\langle Q\rangle R$

b) If $Q \geq x$, but x occurs in R then the procedure first tries to remove x in R by reducing R.

c) If $Q \geq x$ and x does not occur in R, then the η-reduct (B) becomes R and $\underset{\eta}{\geq}$ gets the value true.

ii) $A \equiv [x,P]Q$, $Q \equiv d(\Sigma)$ or $Q \equiv [y,R]S$

Now the procedure first tries to reduce Q, and afterwards tests if an η-reduction is possible.

In either case if no η-reduction is possible, the procedure identifier $\underset{\eta}{\geq}$ gets the value false.

There appear two procedures in $\underset{\eta}{\geq}$, which must still be explained.

Firstly there is the procedure $\overset{D}{=}$ to declare as boolean procedure $E_1 \overset{D}{=} E_2$; where E_1 and E_2 are expressions.

This procedure investigates whether E_1 and E_2 are definitionally equal, and is described in §8.

Secondly there is the procedure OCCURS IN, which searches an expression for occurrences of a specific bound variable. This procedure is defined as follows.

6.3.1. Procedure text for OCCURS IN.

6.3.1.1.
```
boolean procedure x OCCURS IN E;
boundvar x; expression E;
OCCURS IN:=
case shape(E) of
begin
   boundvar  :x≡E;
   d(Σ)      :∃i x OCCURS IN Σi
   <A>B      :x OCCURS IN A or x OCCURS IN B;
   [y,A]B    :x OCCURS IN A or x OCCURS IN B;
   otherwise :false;
end;
```

6.3.2.1. Procedure text for the η-reductor

6.3.2.1.1.
```
boolean procedure   A≥ηB;
expression   A,B;
comment   if A is reducible by η-reduction then B becomes the η-reduct
          of A;
if etareduction allowed then
begin
   if shape(A)=[x,P]Q then
   case shape(Q) of
   begin
      <R>T: if x=DR then
                if not x OCCURS IN T then
                begin ≥η:=true; B:=T
                end
                else
                ?(if T≥T1 and not x OCCURS IN T1
                  then begin ≥η:=true; B:=T
                       end
                  else if Q≥βQ1 then ≥η:=[x,P]Q1≥ηB
                       else ≥η:=false)?
            else if Q≥βQ1 then ≥η:=[x,P]Q1≥ηB
                 else ≥η:=false;
      d(Σ): if Q≥δQ1 then ≥η:=[x,P]Q1≥ηB
            else ≥η:=false;
      [x,R]T: if Q≥ηQ1 then ≥η:=[x,P]Q1≥ηB
              else ≥η:=false;
      otherwise: ≥η:=false;
   end
   else≥≥η:=false;
end;
else ≥η:=false;
```

6.3.2.2. The part between ?(and)? has not yet been implemented.
Although such cases are easily constructed (e.g. [x,X]<x>f(x,y), where f(x,y)≥δy), in practice this has never occurred up to now.

6.4. δ-reduction

The δ-reductor is written in the same way as the β-and η-reductor, and tries to perform a single δ-reduction on the presented expression. If the presented expression has shape d(Σ), the procedure takes the middle expression of the line where d is defined (=MIDDLE(d)) and replaces the free variables in it (i.e. the elements of INDSTR(d))by the expressions of Σ.

6.4.1. Procedure text

6.4.1.1.
```
boolean procedure  A≥δB;
expression  A,B;
comment  if A reducible by δ-reduction then B becomes the δ-reduct of A.
begin
   if shape(A)=d(Σ) then
      if d represents an abbreviation then
      begin ≥δ:=true; B:=SUBST(INDSTR(d), Σ, MIDDLE(d));
      end
      else ≥δ:=false
   else ≥δ:=false;
end;
```

7. CAT and DOM

As pointed out in [3, §6.4], we need two functions, CAT and DOM, to compute mechanically the category (type) and the domain of an expression respectively.

7.1. The "*mechanical type*" *function* CAT is defined by induction on the length of the expressions as follows.

Let $\mathcal{B}$ be a correct book and σ a correct context

i) If $\sigma \equiv x_1 \underline{E} \alpha_1, \ldots, x_n \underline{E} \alpha_n$ then $CAT(x_i) := \alpha_i$

ii) If d is an abbreviation constant, defined in a line of $\mathcal{B}$ by

d:=A E B, with indicator string I,

then $CAT(d(\Sigma)) := [\![I/\Sigma]\!]B$

iii) $CAT(\langle A\rangle B) :=$ if $CAT(B) \equiv [x,P]Q$

then $[x/A]Q$

else $\langle A\rangle CAT(B)$

iv) CAT([x,A]B):=[x,A]CAT(B)

CAT is not defined for variables with shape=boundvar (sec §5.1), because in the verification process there is no need for it. (§9.5)

Futher CAT is not defined for λ-expressions, of course.

It is easy to see that, if the argument for CAT is a correct expression, the outcome will again be correct.

7.2. The procedure text of CAT reflects the given definition completely.

7.2.1.
```
expression procedure  CAT(E);
Expression E;
CAT:=
case shape(E) of
begin
   variable  : CATEGORY(E);
   d(Σ)      : SUBST(INDSTR(d), Σ, CATEGORY(d));
   <A>B      : if shape(CAT(B))=[x,P]Q then BOUNDSUBST(x,A,Q)
               else <A>CAT(B)
   [x,A]B    : [x,A]CAT(B);
   otherwise :undefined;
end;
```

7.3. The *"mechanical domain" function* DOM

This procedure has to yield (where possible), for a given expression A, an expression α, such that $\vdash A \underline{E} [x,\alpha]\beta$ *or* $\vdash A=[x,\alpha]\beta$.

For expressions A of the form [x,B]C, the computing of the domain is trivial: DOM(A)≡B.

If A is a variable, we may compute the domain of the category of A.

More difficult is the case where A has the shape

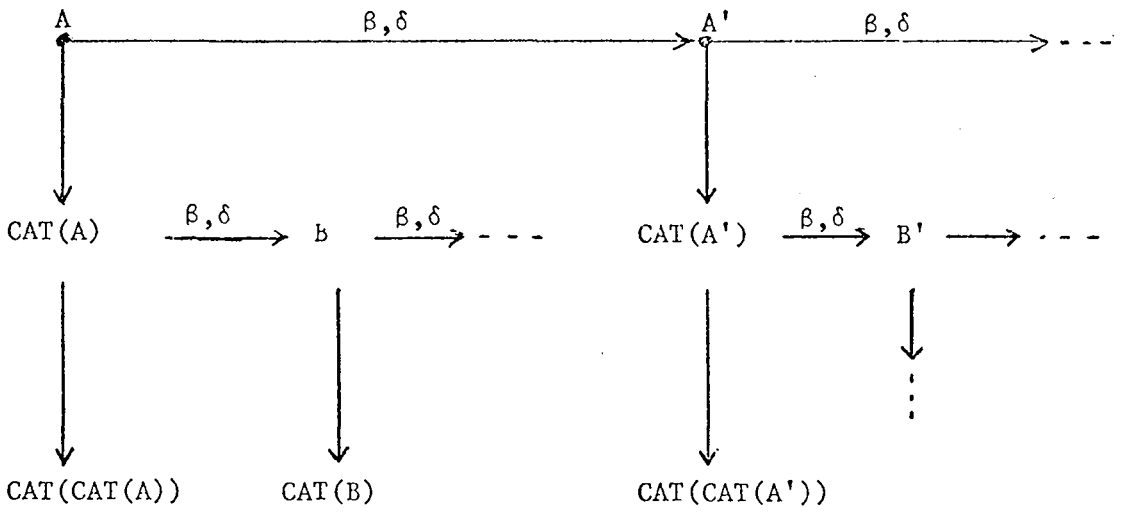
d(Σ) or the shape <B>C.

If we try to reduce A, we may end up with a PN (e.g.: d(Σ)≥f(Γ),f:=PN).

On the other hand, if we take the category of A by computing CAT(A), we may obtain type or $[x_1,\alpha_1]\ldots[x_n,\alpha_n]$type.

To deal with this problem we use the following strategy. At first CAT(A) is computed, and presented to DOM (N.B. This is a recursive call, so possibly CAT(CAT(A)) is computed). If DOM(CAT(A)) does not yield a domain at all, then a δ- or β-reduction on A is carried out (if possible), and the reduct is again presented to DOM.

Since only 1,2 and 3-expressions are investigated, the whole process can be given by the following tree figure:



7.3.1. Procedure text

7.3.1.1.
```
expression procedure  DOM(A);
expression  A;
case shape(A) of
begin
   [x,B]C   : DOM:=B;
   variable  : DOM:=DOM(CATEGORY(A));
   d(Σ),<B>C : begin D:=DOM(CAT(A));
                  if undefined (D) then
                     if A ≥δ A1 then DOM:=DOM(A1)
                     else if A ≥β A1 then DOM:=DOM(A1)
                          else DOM:=undefined
                  else DOM:=D
               end;
   otherwise : undefined;
end;
```

8. Definitional equality

To verify the correctness of a given =-formula we will use the Church-Rosser theorem:

if A=B then A≥C≤B for some C

(see also [3, §6.3.1]).

This definition is the guide for the procedure $\overset{D}{=}$ which we will introduce here.

8.1. Description of $\overset{D}{=}$

The type of the procedure is boolean, and the identifier will be written in infix notation, viz. $A \overset{D}{=} B$ (in the same way as for $\underset{\eta}{\geq}$, $\underset{\delta}{\geq}$ etc.).

Roughly speaking, in order to check $A \overset{D}{=} B$, the procedure tries to reduce A and/or B until either the two expressions are identical or the decision $A \overset{D}{\neq} B$ can be made.

It is not always necessary for both complete expressions to be present during the whole reduction process. If, for example, $A \equiv d(\Sigma_1, \Sigma_2, \Sigma_3)$ and $B \equiv d(\Sigma_1, \Sigma_4, \Sigma_3)$ then the procedure needs only parts of both expressions, namely Σ_2 and Σ_4, and will check $\Sigma_2 \overset{D}{=} \Sigma_4$.

So, in general, the procedure uses recursive calls, applied to sub-expressions, following the monotonicity rules described in [3, §5.5.6].

Recursive calls are also used for the reduction sequences. Firstly the procedure tries, if necesary, to reduce one of the expressions A and B. Which is reduced is a matter of strategy.

If one of the two expressions is reduced, one could continue the equality-check by using an *iterative* or a *recursive* method. A *recursive* method is chosen in order to make the algorithm more readable.

Example:

If $A \equiv d(\Sigma)$ and B=<P>Q then the procedure first tries to reduce B by β-reduction. If this succeeds, and the outcome is B_1, then the definitional equality of A and B follows from that of A and B_1.

Otherwise the procedure tries to reduce A to A_1 (say) and checks $A_1 \overset{D}{=} B$.

If this also fails, then the procedure identifier $\overset{D}{=}$ gets the value <u>false</u>.

8.2. Type inclusion

If we want to verify A <u>E</u> B, we check ⊢A and ⊢B, compute CAT(A) and check $CAT(A) \overset{D}{=} B$; so CAT(A) is the first parameter and B is the second parameter of the procedure call.

In order to accept type inclusion as well, we add a slight extension to $\overset{D}{=}$, namely:

$$[x,A]\underline{type} \overset{D}{=} \underline{type}$$

will be accepted as correct, but *not*

$$\underline{type} \overset{D}{=} [x,A]\underline{type}$$

The same holds for prop. So the procedure is no longer symetrical for l-expressions.

(Notice that calls are sometimes made with reversed order of the arguments of $\overset{D}{=}$, but as one can see in the procedure text these cases can never refer to l-expressions).

Now the definition of $\overset{D}{=}$ is exactly the same as that of $\subseteq$ [3, §6.].

8.3. OLDER THAN

The procedure $\overset{D}{=}$ needs, in one special case, namely $d(\Sigma)\overset{D}{=}b(\Gamma)$ and $d \neq b$, the boolean procedure OLDER THAN, to decide which of d and b must be reduced. It seems a good strategy to start off by reducing the younger of the two, i.e. the constant which was the more recently, for in this way we have a chance of reducing it to the other.

8.3.1. boolean procedure d OLDER THAN b;
definedname d,b;
comment OLDER THAN:= the line in which b is defined, appears later in the book than the line in which d is defined;

8.4. Procedure text of $\overset{D}{=}$

8.4.1.
```
boolean procedure   E1 =D= E2;
expression   E1, E2;
=D=:=
case   (shape (E1), shape(E2))of
begin
   (type, type)              : true;
   (type, otherwise)         : false;
   (prop, prop)              : true;
   (prop, otherwise)         : false;
   (variable, variable)      : E1≡E2
   (variable, d(Σ))          : if E2>E22 then E1 =D= E22 else false;
   (variable, <A>B)          : if E2 >δ E22 then E1 =D= E22 else false;
   (variable, [x,A]B)        : if E2 >β E22 then E1 =D= E22 else false;
   (variable, otherwise)     : false;η
   (boundvar, boundvar)      : E1≡E2;
   (boundvar, otherwise)     : consider (variable, shape(E2))
   (d(Σ), b(Γ))              : if d≡b then
                                  if Σ =SD= Γ then true
                                  else ?(if E1>E11 then E11 =D= E2 else false)?
                               else if d OLDER δ THAN b then
                                       if E2>E22 then E1 =D= E22 else false
                                    else if E1 >δ E11 then E11 =D= E2 else false;
   (d(Σ), <A>B)              : if E2>E22 then E1 δ=D= E22 else
                               if E1 >β E11 then E11 =D= E2 else false
   (d(Σ), [x,A]B)            : if E2 >δ E22 then E1 =D= E22 else
                               if E1 >η E11 then E11 =D= E2 else false;
   (d(Σ), otherwise)         : consider δ reverse (i.e.(shape(E2), shape(E1)))
   (<A>B, <C>D)              : if A =D= C and B =D= D then true
                               else ?(if E1 >β E11 then E11 =D= E2 else
                                      if E2>E22 then E1 =D= E22 else false)?;
   (<A>B, [x,C]D)            : if E1>E11 then E11 =D= E2 else
                               if E2 >β E22 then E1 =D= E22 else false;
   (<A>B, otherwise )        : consider reverse;η
   ([x,A] B, type)           : B =D= E2;
   ([X,A] B, prop)           : B =D= E2;
   ([x,A ]B, [y,C] D)        : if A =D= C then B =D= BOUNDSUBST(y,x,D) else false;
   ([x,A ]B, otherwise)      : consider reverse;
end;
```

8.4.2. boolean procedure $\Sigma_1 \overset{SD}{=\!=} \Sigma_2$;
expressionstring Σ_1, Σ_2;
comment $\overset{SD}{=\!=}$ is the string analogue of $\overset{D}{=}$;
$\overset{SD}{=\!=}$:=if $\Sigma_1 \equiv \emptyset$ then $\Sigma_2 \equiv \emptyset$
else $\Sigma_1^- \overset{SD}{=\!=} \Sigma_2^-$ and $\Sigma_1^+ \overset{D}{=} \Sigma_2^+$;

9. Correctness of expressions. ($\vdash$)

9.1. Correctness of an expression is checked by the boolean procedure "$\vdash$", operating on an expression (say E) and the incicator string (say I) belonging to E. A procedure call is written like I $\vdash$ E.
Mentioning I is necessary, on account of the free variables in E which must all appear in I.
Two non-trivial cases arise:

1) if shape(E)=<A>B, then the "applicability" (let us say) of B to A has to be checked.
This is done by looking at :CAT(A)$\overset{D}{=}$DOM(B). (see also [3, §6.4.2.3])
2) if shape(E)=d(Σ) then
firstly : all Σ_i must be correct,
secondly : all Σ_i must have the correct categories.

In the case 2 there is a difficulty:
Let us consider the following book:

```
∅ * α :=EB ; type.
α * a :=EB ; α
a * f :=PN ; type
∅ * β :=EB ; type
β * b :=EB ; β
b * g :=f(β,b);type.
```

Now: (β,b) $\vdash$f(β,b), nevertheless the string of types expected by f is not definitially equal to the string of given types:

type, α$\overset{SD}{\neq}$type, β

We may conclude that after checking the definitional equality of the first two categories, we have to replace, in the category string of (α,a), the variable α by β
This replacement (substitution) is, in a more general way, done by the procedure CORRECTCATS.(see also [3, §2.5. and 5.4.6.])

9.2. boolean procedure CORRECTCATS(Σ,I);
expressionstring Σ,I;
CORRECTCATS:=
if Σ≡∅ then I≡∅ else
CORRECTCATS(Σ⁻,I⁻) and
CAT(Σ⁺)$\overset{D}{=}$SUBST(I⁻,Σ⁻,CAT(I⁺));

9.3. boolean procedure I ⊢E;
expressionstring I; expression E;
⊢ :=
case shape(E) of
begin
type : true;
prop : true;
variable : $\exists_i(I_i \equiv E)$;
d(Σ) : I $\vdash_s$ Σ and CORRECTCATS(Σ,INDSTR(d));
<A>B : I ⊢A and I ⊢B and CAT(A)$\overset{D}{=}$DOM(B);
[x,A]B : I ⊢A and ((I,x)) ⊢B; (see 9.5.)
otherwise : false;
end;

9.4. boolean procedure I $\vdash_s$Σ;
expressionstring I,Σ;
comment $\vdash_s$ is the string analogue of ⊢;
$\vdash_s$:=
if Σ≡∅ then true
else I $\vdash_s$ Σ⁻ and I ⊢ Σ⁺;

9.5. A comment on I ⊢ [x,P]Q
In this case, the checker, after checking I ⊢ P, adds a "waste-line" to the book, of the form:

I * waste :=EB ;P.

If we denote this new book by B', then the checker checks the statement

B', ((I,waste)) ⊢ [x/waste]Q.

For this reason the correctness of a bound variable will never be asked for, and its CAT or DOM will never be computed.
Only in $\overset{D}{=}$ can the shape boundvar occur.

10. The correctness of lines

The checking for correctness of an AUTOMATH line is now easy to describe in terms of already defined procedures:

10.1.
```
boolean procedure   CORRECT(LINE);
AUTOMATH line value   LINE;
CORRECT   :=
case form of the line is of
begin
   I * N := EB ; E.   :I  - E ;
   I * N := PN ; E1   :I  - E ;
   I * N := E1 ; E2   :I  - E1 and I  - E2 and CAT(E1) =D= E2;
   otherwise: false;
end;
```

11. A paragraph system

As already mentioned in [3 section 2.16,] the syntactical definition of AUT-68 (and AUT-QE) forces us to write mutually exclusive names (identifiers) for both variables and constants. This, of course, is very annoying to the writer of AUTOMATH. Therefore we have introduced a paragraph system. Each AUTOMATH text may be divided into sections, called paragraphs. A paragraph starts with:

+ paragraph name.

and ends with:

- paragraph name.

In a paragraph one may write AUTOMATH lines and other paragraphs (sub-paragraphs). Finally the whole book is contained in one big paragraph, so all paragraphs occur nested. Behind the identifier of a given constant one may write a so-called paragraph reference, to indicate in which paragraph this identifier has been defined. An identifier b with paragraph reference to (say) paragraph P_n is written in the form: b"P_1-P_2-...-P_n", where P_2 is a sub-paragraph of P_1, P_3 is a sub-paragraph of P_2,..., and P_n is the paragraph in which b is actually defined.
An identifier, not followed by a paragraph reference, refers to a constant or variable defined in the same paragraph, or, if not found there, in the paragraph, which contains that one, ans so on.

Example: (a :=... denotes a definition of a
...(a)...denotes a reference to a)

line nr	book	reference to line nr:
	+ A.	
1	p:=...	
	+ B.	
	+ C.	
2	...(p"A")...	1
3	...(p"B")...	
		no good reference (p has not been defined in B).
4	p:=...	
5	...(p)...	4
6	...(p"A-B-C")...	4
7	...(p"A")...	1
	- C.	
8	...(p)...	1
	- B.	
	- A.	

12. Final remarks

12.1. We repeat that the procedures given here form only an outline of the actual verifier. Many more parameters are passed through the procedures to avoid duplication, to control critical passages, to permit communication with the user and so on.

12.2. With regard to efficiency, improvements may be possible. For example, parts of the strategy, implemented in $\stackrel{D}{=}$, are more or less arbitrary, although suggested by reflexion and practical work. Experience and research may lead to better strategies.
Also the use of the features of [4] may lead to a more efficient verifier.

12.3. We are pleased to say, in any event, that the verifier has been working satisfactorily up to now.

12.4. An example of a text checked with the described verifier is found in [5]

Reference list

[1] De Bruijn, N.G., *The Mathematical language AUTOMATH, its usage and some of its extensions;* Symposium on Automatic Demonstration (Versailles, December 1968), Lecture Notes in Mathematics, Vol. 125, p, 29-61, Springer Verlag, Berlin 1970.

[2] De Bruijn, N.G., *AUTOMATH, a language for mathematics;* notes (prepared by B. Fawcett) of a series of lectures in the Séminaire de Mathématiques Superieures, Université de Montréal, 1971.

[3] van Daalen, D T., *A description of AUTOMATH and some aspects of its languagetheory,* this volume.

[4] De Bruijn, N.G., *Lambda Calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem,* Indag. Math., 34, No. 5, 1972.

[5] Jutting, L.S. v. Benthem, *The development of a text in AUT-QE,* this volume.

[6] Nederpelt, R P., *Strong normalisation in a typed lambda-calculus with lambda-structured types,* Doctoral dissertation, Technological University Eindhoven, 1972.

The development of a text in AUT-QE

by L.S. van Benthem Jutting.

Introduction.

As the subject of the AUTOMATH text which will serve as an example of how the language may be used, we choose the following theorem. Let N denote the set of naturals {1,2,3,...} and let [1,n] denote {x∈N | x≤n}.

Theorem. If f is an injection of [1,n] into itself then f is a surjection.

In the following pages a formal proof of this theorem in AUTOMATH will be developed in four successive stages A,B,C,D.

The proof A is a proof which would normally be given in conversation. It is very informal and contains many gaps.

Proof B could be a page in a rather pedantic textbook. In contrast to what is usually done, we are taking the functions in this proof to be total functions on N, because this simplifies the treatment in AUTOMATH.

In proof C the arguments are given in more detail. Here the vertical bars in the margin indicate the context as far as assumptions are concerned. Also certain lemmas concerning order in the naturals which are used in the course of the proof are noted explicitly (by Roman numerals). Logical concepts and rules however are used without notice.

After the completion of proof C, in a section headed "Naturals", the concept of order in the naturals is defined and the lemmas which were assumed in proof C are proved.

Proof D is a text in AUT-QE which has been checked by the program described by I. Zandleven [2]. Here sections "Logic" and "Naturals", containing the relevant concepts, rules and lemmas, precede the main proof. For an introduction to the language AUT-QE and the interpretation of logic in this language we refer to D. van Daalen [1]. The glossary at the and of this proof will serve to facilitate its decoding by a (human) reader.

A Let $[1,n]$ denote the set of naturals $\{1,2,\ldots,n\}$

Theorem. If f maps $[1,n]$ one to one into itself then f maps $[1,n]$ onto itself.

Proof. Induction.

For n=1 the issue is clear.

Now suppose the theorem has been proved up to k.

And suppose f maps $[1,k+1]$ one to one into itself.

Now if $f(k+1)=k+1$ then the induction hypothesis applied to the restriction of f to $[1,k]$ proves the theorem.

Therefore suppose $f(k+1)\leq k$. Define for $x\leq k$

$$g(x):=\begin{cases} f(k+1) & \text{if } f(x)=k+1 \\ f(x) & \text{if } f(x)\leq k \end{cases}$$

(see fig. 1)

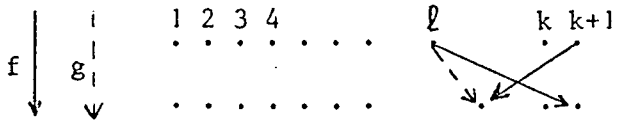


fig. 1.

$g(x)=f(x)$ for $x\neq \ell$, $x\leq k$ (where $f(\ell)=k+1$).

Then the induction hypothesis applied to g proves the theorem.

B Let f be a function from N to N.

f is called bounded by n if $f(x)\leq n$ for all x.

f is called injective on $[1,n]$ if for all $x\leq n$ and all $y\leq n$ $f(x)=f(y)$ implies $x=y$.

f is called surjective on $[1,n]$ if for all $x\leq n$ there is a $y\leq n$ such that $f(y)=x$.

Theorem. If f is bounded by n and injective on $[1,n]$ then f is surjective on $[1,n]$.

Proof. Induction with respect to n.

1 If $f(x)\leq 1$ for all x then $f(1)\leq 1$ hence $f(1)=1$.
It follows that f is surjective on $[1,1]$.

2 Suppose the theorem holds for n.
Let f be bounded by n' and injective on $[1,n']$.
We discern two cases:
a) $f(n')=n'$.
We define g by $g(x):=\begin{cases} f(x) & \text{if } f(x)\leq n \\ n & \text{if } f(x)=n'. \end{cases}$
It is clear that g is bounded by n.
Moreover g is injective on $[1,n]$. In fact, if $x\leq n$ then $x\neq n'$, hence $f(x)\neq f(n')=n'$. It follows that $g(x)=f(x)$ for $x\leq n$. Hence, if $x\leq n$ and $y\leq n$ and $g(x)=g(y)$, then $f(x)=f(y)$ and therefore $x=y$.
Now if $x\leq n$ then,by induction hypothesis, there exists a $y\leq n$ such that $g(y)=x$ and, as $g(y)=f(y)$ for $y\leq n$, also $f(y)=x$.
If $x=n'$ then $f(n')=x$.
From this it follows that f is surjective on $[1,n']$.
b) $f(n')\leq n$.
We define h by $h(x):=\begin{cases} f(x) & \text{if } f(x)\leq n \\ f(n') & \text{if } f(x)=n'. \end{cases}$
Again it is clear that h is bounded by n.
Also, if $x\leq n$ then $x\neq n'$, hence $f(x)\neq f(n')$.
Therefore, if $x\leq n$ then $h(x)=f(n')$ implies $f(x)=n'$
and $h(x)\neq f(n')$ implies $h(x)=f(x)$.
We now will prove that h is injective on $[1,n]$. In fact, suppose $x\leq n$, $y\leq n$ and $h(x)=h(y)$. Now if $h(x)=f(n')$ then $h(y)=f(n')$, hence $f(x)=n'=f(y)$ and therefore $x=y$.
And if $h(x)\neq f(n')$ then $h(y)\neq f(n')$ hence $f(x)=h(x)=h(y)=f(y)$ and therefore again $x=y$.
By the induction hypothesis h is surjective on $[1,n]$.
Now let $x\leq n$ and $x\neq f(n')$. Then $h(y)=x$ for some $y\leq n$ and, as $h(y)\neq f(n')$, also $f(y)=h(y)=x$. Hence, if $x\leq n$,then there is a $y\leq n'$ such that $f(y)=x$.
And if $x=n'$ then, as for some $y\leq n$ $h(y)=f(n')$, it follows that for this y $f(y)=n'=x$.
It follows that in this case too f is surjective on $[1,n']$.

C f bounded by n:= $\forall x\,[f(x)\leq n]$.

f injective on [1,n]:= $\forall x\leq n\ \forall y\leq n\,[f(x)=f(y) \Rightarrow x=y]$.

x n-image under f:= $\exists y\leq n\,[f(y)=x]$.

f surjective on [1,n]:= $\forall x\leq n\,[x$ n-image$]$.

Theorem. $\forall f:N\to N\,[f$ bounded by n $\Rightarrow$
(f injective on [1,n] $\Rightarrow$ f surjective on [1,n])].

Proof.

Let f: N→N.

b1⌋ Let f bounded by 1.

Let k≤1.

Then k=1 and (by b1⌋) f(1)=1. I $\dfrac{x\leq 1}{x=1}$

Therefore f(1)=k.

As 1≤1 we have II $x\leq x$

∃y≤1 [f(y)=k] i.e. k 1-image under f.

Hence ∀x≤1 [x 1-image] i.e. f surjective on [1,1].

Therefore Theorem holds for n=1.

ind. hyp.⌋ Suppose Theorem holds for n.

Let f: N→N.

b2⌋ Let f bounded by n'.

i⌋ Let f be injective on [1,n'].

Suppose f(n')=n'.

Define for k∈N

a(k):= if f(k)=n' then n else f(k).

If f(k)=n' then a(k)=n≤n. II

If f(k)≠n' then (by b2⌋) f(k)≤n, III $\dfrac{x\leq n' \quad x\neq n'}{x\leq n}$

hence a(k)=f(k)≤n.

1⌋ It follows that in any case a(k)≤n.

Let k≤n.

Suppose f(k)=n',

then f(k)=f(n'),

hence (by i⌋) k=n' which is impossible. IV $\dfrac{x<n}{x\neq n'}$

It follows that f(k)≠n'.

Now let also ℓ≤n.

And suppose a(k)=a(ℓ).

Then (as f(k)≠n' and f(ℓ)≠n') f(k)=a(k)=a(ℓ)=f(ℓ).

2⌋ Hence (by i) k=ℓ.

Define $g:=\lambda x.a(x)$.

1⌋ proves that g is bounded by n.

2⌋ proves that g is injective on $[1,n]$.

Let $k\leq n'$.

Suppose $k=n'$.

Then $f(n')=k$.

As $n'\leq n'$ we have II

$\exists y\leq n'\ [f(y)=k]$ i.e. k n'-image under f.

Now suppose $k\leq n$.

Then, by ind. hyp.⌋ applied to g and k, $\exists y\leq n\ [g(y)=k]$.

Let $\ell\leq n$.

Let $g(\ell)=k$.

Then (as $f(\ell)\neq n'$) $k=g(\ell)=a(\ell)=f(\ell)$.

As $\ell\leq n'$ we have V $\dfrac{x\leq n}{x\leq n'}$,

$\exists y\leq n'\ [f(y)=k]$ i.e. k n'-image under f.

Hence in both cases VI $\dfrac{x\leq n'}{x=n' \vee x\leq n}$

k n'-image under f.

Hence $\forall k\leq n'$ [k n'-image] i.e. f surjective on $[1,n']$.

Now suppose $f(n')\leq n$.

Define for $k\in N$

$b(k):=$ if $f(k)=n'$ then $f(n')$ else $f(k)$.

If $f(k)=n'$ then $b(k)=f(n')\leq n$.

If $f(k)\neq n'$ then (by b2⌋) $f(k)\leq n$, III

hence $b(k)=f(k)\leq n$.

3⌋It follows that in anycase $b(k)\leq n$.

Let $k\leq n$.

Suppose $b(k)=f(n')$.

Now if $f(k)\neq n'$ then $f(n')=b(k)=f(k)$,

hence (by i⌋) $k=n'$, which is impossible. IV

It follows that $f(k)=n'$.

Suppose on the other hand $b(k)\neq f(n')$,

then clearly $b(k)=f(k)$.

Now let also $\ell\leq n$.

And suppose $b(k)=b(\ell)$.

Now if $b(k)=f(n')$

then also $b(\ell)=f(n')$,

hence $f(k)=n'=f(\ell)$.

And if $b(k)\neq f(n')$

then also $b(\ell)\neq f(n')$

and therefore $f(k)=b(k)=b(\ell)=f(\ell)$.

It follows that in anycase $f(k)=f(\ell)$.

4] Hence (by i])$k=\ell$.

Define $h:=\lambda x.b(x)$.

3] proves that h is bounded by n.

4] proves that h is injective on $[1,n]$.

Let $k\leq n'$.

Suppose $k=n'$.

Then, by ind. hyp.] applied to h and $f(n')$, $\exists y\leq n\ [h(y)=f(n')]$.

Let $\ell\leq n$.

Let $h(\ell)=f(n')$.

Then, as $b(\ell)=h(\ell)=f(n')$,

it follows that $f(\ell)=n'=k$.

As $\ell\leq n'$ we have V

$\exists y\leq n'\ [f(y)=k]$ i.e. k n'-image under f.

Now suppose $k\leq n$.

Let $k=f(n')$.

Then, as $n'\leq n'$, we have II

$\exists y\leq n'\ [f(y)=k]$ i.e. k n'-image under f.

Now let $k\neq f(n')$.

Then,by ind. hyp.] applied to h and k, $\exists y\leq n\ [h(y)=k]$.

Let $\ell\leq n$.

Let $h(\ell)=k$.

Then $b(\ell)=h(\ell)=k\neq f(n')$,

hence $f(\ell)=b(\ell)=h(\ell)=k$.

Again, as $\ell\leq n'$ we have V

$\exists y\leq n'\ [f(y)=k]$ i.e. k n'-image under f.

It follows that in both cases k n'-image under f. VI

Hence in all cases k n'-image under f. VI

Therefore $\forall k\leq n$ [k n'-image] i.e. f surjective on $[1,n']$.

We now have proved that in any case f is surjective on $[1,n']$.

Therefore Theorem holds for n'.

Now by induction we infer that Theorem holds for any n.

Naturals.

ax1⌋ $1 \in N$.

For $k \in N$.

ax2⌋ $k' \in N$.

Let $k = \ell$

then $k' = 2$;

ax3⌋ $k' \neq 1$.

Let $k' = \ell'$ then

ax4⌋ $k = \ell$.

S denotes a set of naturals.

S progressive := $\forall x \in S\ [x' \in S]$.

Let S be progressive.

Let $1 \in S$ then

ax5⌋ $k \in S$.

P denotes a predicate over N.

Let P(1).

Let $\forall x \in N\ [P(x) \Rightarrow P(x')]$.

$S := \{x \mid P(x)\}$.

Then $1 \in S$.

Let $k \in S$.

Then P(k), hence P(k')

and therefore $k' \in S$.

Hence $\forall k \in S\ [k' \in S]$ i.e. S is progressive.

Therefeore (by ax5⌋) $k \in S$, and hence

induction (for predicates) P(k).

$k \leq \ell := \forall S \in P(n)\ [S \text{ progressive} \Rightarrow (k \in S \Rightarrow \ell \in S)]$.

refl $\leq$ ⌋ $k \leq k$. II

Let $k \leq \ell$.

Let $\ell \leq m$.

Let S be progressive.

Let $k \in S$

then $\ell \in S$

hence $m \in S$. It follows that

tr$\leq$⌋ $k \leq m$.

th1⌋ $k \leq k'$.

Let $k \leq \ell$, then (by tr≤and th1)
cor1⌋ $k \leq \ell'$. V

Let $k' \leq \ell'$.
Let S be progressive.
So:= $\{x' \mid x \in S\}$
Let $m \in So$
then $\exists x \in S\ [x'=m]$.
Let $n \in S$.
Let $n'=m$
then (as S is progressive) $n' \in S$
and $n''=m'$.
It follows that $\exists x \in S\ [x'=m']$.
Hence $m' \in So$.
Therefore $\forall x \in So\ [x' \in So]$ i.e. So progressive.
Let $k \in S$
then $k' \in So$,
therefore (as $k' \leq \ell'$) $\ell' \in So$,
hence $\exists x \in S\ [x'=\ell']$.
Let $m \in S$.
Let $m'=\ell'$
then (by ax4⌋) $m=\ell$.
It follows that $\ell \in S$ and hence
th2⌋ $k \leq \ell$.

Let $k \leq \ell$.
Let $k \neq \ell$.
Let S be progressive.
Let $k' \in S$.
So:= $\{x \mid x=k \vee x \in S\}$
Let $m \in So$,
then $m=k \vee m \in S$.
Now if $m=k$ then $m'=k' \in S$,
And if $m \in S$ then (as S is progressive) $m' \in S$.
Hence in any case $m' \in S$,
therefore $m' \in So$.

It follows that $\forall x \in So\ [x' \in So]$ i.e. So progressive.
Now $k \in So$
hence $\ell \in So$
and (as $k \neq \ell$) $\ell \in S$. We conclude
th3⌋ $k' \leq \ell$.

| Let $k \leq \ell'$.
| | Let $k \neq \ell'$, then (by th2⌋ and th3⌋)
| | cor3a⌋ $k \leq \ell$. III

| Let $k \leq \ell'$, then (by cor3a⌋)
| cor3b⌋ $k = \ell' \vee k \leq \ell$. VI

So:= $\{x' \mid x \in N\}$.
Now $\exists x \in N\ [x' = k']$,
hence $k' \in So$.
| Suppose $k' \leq 1$,
| then (as So is progressive) $1 \in So$.
| This contradicts ax3⌋ . We conclude
th4⌋ $\sim (k' \leq 1)$.

| Let $k \leq 1$, then (by th3⌋ and th4⌋)
| cor4⌋ $k = 1$. I

For $k \in N$
$P(k) := \sim(k' \leq k)$.
Then (by th4⌋) $P(1)$.
| And suppose $P(k)$
| then (by th2⌋) $P(k')$. We conclude by induction⌋
th5⌋ $\sim(k' \leq k)$.

| Let $k \leq \ell$, then (by th5⌋)
| cor5⌋ $k \neq \ell'$. IV

```
+LOGIC.

    * A             := EB                                  ; PROP.

A   * B             := EB                                  ; PROP.

B   * IMP           := [T,A]B                              ; PROP.          (1)

    * CON           := PN                                  ; PROP.          (2)

A   * NOT           := IMP(A,CON)                          ; PROP.          (3)

B   * I             := EB                                  ; IMP(A,B).

I   * N             := EB                                  ; NOT(B).

N   * CONTRAPOS     := [T,A]<<T>I>N                        ; NOT(A).        (4)

A   * AØ            := EB                                  ; A.

AØ  * TH1           := [T,NOT(A)]<AØ>T                     ; NOT(NOT(A)).   (5)

A   * N             := EB                                  ; NOT(NOT(A)).

N   * DBLNEGLAW     := PN                                  ; A.             (6)

B   * I             := EB                                  ; IMP(A,B).

I   * J             := EB                                  ; IMP(NOT(A),B).

J   * ANYCASE       := DBLNEGLAW(B,[T,                                      (7)
                       NOT(B)]<<CONTRAPOS(A,B,I,
                       T)>J>T)                              ; B.

B   * N             := EB                                  ; NOT(A).

N   * TH2           := [T,A]DBLNEGLAW(B,[U,                                 (8)
                       NOT(B)]<T>N)                         ; IMP(A,B).

B   * AØ            := EB                                  ; A.

AØ  * N             := EB                                  ; NOT(B).

N   * TH3           := [T,IMP(A,B)]<<AØ>T>N                ; NOT(IMP(A,B)). (9)

B   * N             := EB                                  ; NOT(IMP(A,B)).

N   * TH4           := DBLNEGLAW(A,[T,                                      (10)
                       NOT(A)]<TH2(A,B,T)>N)                ; A.

N   * TH5           := [T,B]<[U,A]T>N                      ; NOT(B).        (11)
```

```
B   * OR          := IMP(NOT(A),B)                          ; PROP.          (12)

B   * AØ          := EB                                     ; A.

AØ  * ORI1        := TH2(NOT(A),B,TH1(A,AØ))                ; OR(A,B).       (13)

B   * BØ          := EB                                     ; B.

BØ  * ORI2        := [T,NOT(A)]BØ                           ; OR(A,B).       (14)

B   * 0           := EB                                     ; OR(A,B).

C   * N           := EB                                     ; NOT(A).

N   * NOTCASE1    := <N>0                                   ; B.             (15)

0   * N           := EB                                     ; NOT(B).

N   * NOTCASE2    := DBLNEGLAW(A,                                            (16)
                     CONTRAPOS(NOT(A),B,0,N))               ; A.

B   * C           := EB                                     ; PROP.

C   * 0           := EB                                     ; OR(A,B).

0   * I           := EB                                     ; IMP(A,C).

I   * J           := EB                                     ; IMP(B,C).

J   * ORE         := ANYCASE(A,C,I,[T,                                       (17)
                     NOT(A)]<<T>0>J)                        ; C.

B   * AND         := NOT(IMP(A,NOT(B)))                     ; PROP.          (18)

B   * AØ          := EB                                     ; A.

AØ  * BØ          := EB                                     ; B.

BØ  * ANDI        := TH3(A,NOT(B),AØ,TH1(B,BØ))             ; AND(A,B).      (19)

B   * A1          := EB                                     ; AND(A,B).

A1  * ANDE1       := TH4(A,NOT(B),A1)                       ; A.             (20)

A1  * ANDE2       := DBLNEGLAW(B,TH5(A,NOT(B),                               (21)
                     A1))                                   ; B.
```

```
    * NAT          := PN                                  ; TYPE.          (22)
    * P            := EB                                  ; [X,NAT]PROP.
P   * ALL          := P                                   ; PROP.          (23)
P   * SOME         := NOT(ALL([X,NAT]NOT(<X>P)))          ; PROP.          (24)
P   * K            := EB                                  ; NAT.
K   * KP           := EB                                  ; <K>P.
KP  * SOMEI        :- [T,[X,                                               (25)
                       NAT]NOT(<X>P)]<KP><K>T             ; SOME(P).
P   * A            := EB                                  ; PROP.
A   * S            := EB                                  ; SOME(P).
S   * AØ           := EB                                  ; [X,NAT][T,<X>P]A.

+1.

AØ  * N            := EB                                  ; NOT(A).
N   * K            := EB                                  ; NAT.
K   * T1           := CONTRAPOS(<K>P,A,<K>AØ,N)           ; NOT(<K>P).
N   * T2           := <[X,NAT]T1(X)>S                     ; CON.

-1.

AØ  * SOMEE        := DBLNEGLAW(A,[T,                                      (26)
                       NOT(A)]T2"-1"(T))                  ; A.
```

```
    * K          := EB                                  ; NAT.

K   * L          := EB                                  ; NAT.

L   * IS         := PN                                  ; PROP.          (27)

K   * REFLEQ     := PN                                  ; IS(K,K).       (28)

L   * I          := EB                                  ; IS(K,L).

I   * P          := EB                                  ; [X,NAT]PROP.

P   * KP         := EB                                  ; <K>P.

KP  * EQPRED1    := PN                                  ; <L>P.          (29)

I   * SYMEQ      := EQPRED1([X,NAT]IS(X,K),             (30)
                    REFLEQ(K))                          ; IS(L,K).

P   * LP         := EB                                  ; <L>P.

LP  * EQPRED2    := EQPRED1(L,K,SYMEQ(K,L,I),           (31)
                    P,LP)                               ; <K>P.

L   * M          := EB                                  ; NAT.

M   * I          := EB                                  ; IS(K,L).

I   * J          := EB                                  ; IS(L,M).

J   * TREQ       := EQPRED1(L,M,J,[X,NAT]IS(K,          (32)
                    X),I)                               ; IS(K,M).

M   * I          := EB                                  ; IS(K,M).

I   * J          := EB                                  ; IS(L,M).

J   * CONVEQ     := TREQ(K,M,L,I,SYMEQ(L,M,J))          ; IS(K,L).       (33)

M   * I          := EB                                  ; IS(M,K).

I   * J          := EB                                  ; IS(M,L).

J   * DIVEQ      := TREQ(K,M,L,SYMEQ(M,K,I),J)          ; IS(K,L).       (34)

M   * N          := EB                                  ; NAT.

N   * I          := EB                                  ; IS(K,L).

I   * J          := EB                                  ; IS(L,M).

J   * IØ         := EB                                  ; IS(M,N).

IØ  * TR3EQ      := TREQ(K,M,N,TREQ(K,L,M,I,            (35)
                    J),IØ)                              ; IS(K,N).
```

```
    * P            := EB                              ; [X,NAT]PROP.

P   * NOTTWO       := [X,NAT][Y,NAT][T,<X>P][U,
                      <Y>P]IS(X,Y)                    ; PROP.          (36)

P   * ONE          := AND(SOME(P),NOTTWO(P))          ; PROP.          (37)

P   * 0            := EB                              ; ONE.

0   * INDIVIDUAL   := PN                              ; NAT.           (38)

0   * AXINDIVIDUAL:= PN                               ; <INDIVIDUAL>P. (39)

    * A            := EB                              ; PROP.

A   * K            := EB                              ; NAT.

K   * L            := EB                              ; NAT.

+3.

L   * N            := EB                              ; NAT.

N   * PROP1        := IMP(A,IS(N,K))                  ; PROP.

N   * PROP2        := IMP(NOT(A),IS(N,L))             ; PROP.

N   * PROP3        := AND(PROP1,PROP2)                ; PROP.

L   * AØ           := EB                              ; A.

AØ  * T1           := ANDI(PROP1(K),PROP2(K),[T,
                      A]REFLEQ(K),TH2(NOT(A),
                      IS(K,L),TH1(A,AØ)))             ; PROP3(K).

AØ  * T2           := SOMEI([X,NAT]PROP3(X),K,
                      T1)                             ; SOME([X,NAT]PROP3(X)).

L   * A1           := EB                              ; NOT(A).

A1  * T3           := ANDI(PROP1(L),PROP2(L),
                      TH2(A,IS(L,K),A1),[T,
                      NOT(A)]REFLEQ(L))               ; PROP3(L).

A1  * T4           := SOMEI([X,NAT]PROP3(X),L,
                      T3)                             ; SOME([X,NAT]PROP3(X)).

L   * EXISTENCE    := ANYCASE(A,SOME([X,
                      NAT]PROP3(X)),[T,A]T2(T),
                      [T,NOT(A)]T4(T))                ; SOME([X,NAT]PROP3(X)).
```

```
L   * M          := EB                                  ; NAT.

M   * P          := EB                                  ; PROP3(M).

P   * AØ         := EB                                  ; A.

AØ  * T5         := <AØ>ANDE1(PROP1(M),
                    PROP2(M),P)                         ; IS(M,K).

P   * A1         := EB                                  ; NOT(A).

A1  * T6         := <A1>ANDE2(PROP1(M),
                    PROP2(M),P)                         ; IS(M,L).

M   * N          := EB                                  ; NAT.

N   * P          := EB                                  ; PROP3(M).

P   * Q          := EB                                  ; PROP3(N).

Q   * AØ         := EB                                  ; A.

AØ  * T7         := CONVEQ(M,N,K,T5(M,P,AØ),
                    T5(N,Q,AØ))                         ; IS(M,N).

Q   * A1         := EB                                  ; NOT(A).

A1  * T8         := CONVEQ(M,N,L,T6(M,P,A1),
                    T6(N,Q,A1))                         ; IS(M,N).

Q   * UNICITY    := ANYCASE(A,IS(M,N),[T,
                    A]T7(T),[T,NOT(A)]T8(T))            ; IS(M,N).

L   * T9         := ANDI(SOME([X,
                    NAT]PROP3(X)),NOTTWO([X,
                    NAT]PROP3(X)),EXISTENCE,
                    [X,NAT][Y,NAT][T,
                    PROP3(X)][U,
                    PROP3(Y)]UNICITY(X,Y,T,U))          ; ONE([X,NAT]PROP3(X)).

L   * NØ         := INDIVIDUAL([X,
                    NAT]PROP3(X),T9)                    ; NAT.

L   * T1Ø        := AXINDIVIDUAL([X,
                    NAT]PROP3(X),T9)                    ; PROP3(NØ).

-3.

L   * IFTHENELSE := NØ"-3"                              ; NAT.          (40)

L   * AØ         := EB                                  ; A.

AØ  * THEN       := T5"-3"(NØ"-3",T1Ø"-3",AØ)           ; IS(IFTHENELSE,K).   (41)

L   * A1         := EB                                  ; NOT(A).

A1  * ELSE       := T6"-3"(NØ"-3",T1Ø"-3",A1)           ; IS(IFTHENELSE,L).   (42)
```

	* SET	:= PN	; TYPE.	(43)
	* K	:= EB	; NAT.	
K	* S	:= EB	; SET.	
S	* IN	:= PN	; PROP.	(44)
	* P	:= EB	; [X,NAT]PROP.	
P	* SETOF	:= PN	; SET.	(45)
P	* K	:= EB	; NAT.	
K	* KP	:= EB	; <K>P.	
KP	* INI	:= PN	; IN(K,SETOF(P)).	(46)
K	* I	:= EB	; IN(K,SETOF(P)).	
I	* INE	:= PN	; <K>P.	(47)

```
+NATURALS.

    * 1             := PN                                  ; NAT.

    * K             := EB                                  ; NAT.

K   * SUC           := PN                                  ; NAT.          (48)

K   * L             := EB                                  ; NAT.

L   * I             := EB                                  ; IS(K,L).

I   * AX2           := EQPRED1(K,L,I,[X,                                   (49)
                       NAT]IS(SUC(K),SUC(X)),
                       REFLEQ(SUC(K)))
                                                           ; IS(SUC(K),SUC(L)).

K   * AX3           := PN                                  ; NOT(IS(SUC(K),1)).  (50)

L   * I             := EB                                  ; IS(SUC(K),SUC(L)).

I   * AX4           := PN                                  ; IS(K,L).      (51)

    * S             := EB                                  ; SET.

S   * PROGRESSIVE   := ALL([X,NAT]IMP(IN(X,S),                             (52)
                       IN(SUC(X),S)))
                                                           ; PROP.

S   * P             := EB                                  ; PROGRESSIVE(S).

P   * I             := EB                                  ; IN(1,S).

I   * K             := EB                                  ; NAT.

K   * AX5           := PN                                  ; IN(K,S).      (53)

    * P             := EB                                  ; [X,NAT]PROP.

P   * 1P            := EB                                  ; <1>P.

1P  * A             := EB                                  ; ALL([X,NAT]IMP(<X>P,
                                                             <SUC(X)>P)).

A   * K             := EB                                  ; NAT.

+Ø.

A   * SØ            := SETOF(P)                            ; SET.

A   * T1            := INI(P,1,1P)                         ; IN(1,SØ).

K   * I             := EB                                  ; IN(K,SØ).

I   * T2            := INI(P,SUC(K),<INE(P,K,
                       I)><K>A)                            ; IN(SUC(K),SØ).

-Ø.

K   * INDUCTION     := INE(P,K,AX5(SØ"-Ø",[X,                              (54)
                       NAT][T,IN(X,
                       SØ"-Ø")]T2"-Ø"(X,T),
                       T1"-Ø",K))                          ; <K>P.
```

```
    * K          := EB                                   ; NAT.

K   * L          := EB                                   ; NAT.

L   * LE         := [S,SET][T,                                          (55)
                    PROGRESSIVE(S)]IMP(IN(K,
                    S),IN(L,S))                          ; PROP.

K   * REFLLE     := [S,SET][T,                                          (56)
                    PROGRESSIVE(S)][U,IN(K,
                    S)]U                                 ; LE(K,K).

L   * M          := EB                                   ; NAT.

M   * L1         := EB                                   ; LE(K,L).

L1  * L2         := EB                                   ; LE(L,M).

+*Ø.

L2  * S          := EB                                   ; SET.

S   * P          := EB                                   ; PROGRESSIVE(S).

P   * I          := EB                                   ; IN(K,S).

I   * T3         := <I><P><S>L1                          ; IN(L,S).

I   * T4         := <T3><P><S>L2                         ; IN(M,S).

-Ø.

L2  * TRLE       := [S,SET][T,                                          (57)
                    PROGRESSIVE(S)][U,IN(K,
                    S)]T4"-Ø"(S,T,U)                     ; LE(K,M).

K   * TH1        := [S,SET][T,                                          (58)
                    PROGRESSIVE(S)]<K>T                  ; LE(K,SUC(K)).

L   * L1         := EB                                   ; LE(K,L).

L1  * COR1       := TRLE(K,L,SUC(L),L1,TH1(L))           ; LE(K,SUC(L)).  (59)
```

```
L   * L1          := EB                                  ; LE(SUC(K),SUC(L)).

+2.

L1  * S           := EB                                  ; SET.

S   * P           := EB                                  ; PROGRESSIVE(S).

P   * M           := EB                                  ; NAT.

M   * N           := EB                                  ; NAT.

N   * PROP1       := AND(IN(N,S),IS(SUC(N),M))           ; PROP.

P   * SØ          := SETOF([X,NAT]SOME([Y,
                     NAT]PROP1(X,Y)))                    ; SET.

P   * M           := EB                                  ; NAT.

M   * I           := EB                                  ; IN(M,SØ).

I   * T1          := INE([X,NAT]SOME([Y,
                     NAT]PROP1(X,Y)),M,I)                ; SOME([X,NAT]PROP1(M,
                                                           X)).

I   * N           := EB                                  ; NAT.

N   * Q           := EB                                  ; PROP1(M,N).

Q   * T2          := <ANDE1(IN(N,S),IS(SUC(N),
                     M),Q)><N>P                          ; IN(SUC(N),S).

Q   * T3          := AX2(SUC(N),M,ANDE2(IN(N,
                     S),IS(SUC(N),M),Q))                 ; IS(SUC(SUC(N)),
                                                           SUC(M)).

Q   * T4          := ANDI(IN(SUC(N),S),
                     IS(SUC(SUC(N)),SUC(M)),T2,
                     T3)                                 ; PROP1(SUC(M),SUC(N)).

Q   * T5          := SOMEI([X,NAT]PROP1(SUC(M),
                     X),SUC(N),T4)                       ; SOME([X,
                                                           NAT]PROP1(SUC(M),X)).

I   * T6          := SOMEE([X,NAT]PROP1(M,X),
                     SOME([X,NAT]PROP1(SUC(M),
                     X)),T1,[X,NAT][T,PROP1(M,
                     X)]T5(X,T))                         ; SOME([X,
                                                           NAT]PROP1(SUC(M),X)).

I   * T7          := INI([X,NAT]SOME([Y,
                     NAT]PROP1(X,Y)),SUC(M),T6)          ; IN(SUC(M),SØ).
```

```
P   * I          := EB                                  ; IN(K,S).

I   * T8         := ANDI(IN(K,S),IS(SUC(K),
                    SUC(K)),I,REFLEQ(SUC(K)))           ; PROP1(SUC(K),K).

I   * T9         := SOMEI([X,NAT]PROP1(SUC(K),
                    X),K,T8)                            ; SOME([X,
                                                          NAT]PROP1(SUC(K),X)).

I   * T10        := INI([X,NAT]SOME([Y,
                    NAT]PROP1(X,Y)),SUC(K),T9)          ; IN(SUC(K),SØ).

I   * T11        := <TIØ><[X,NAT][T,IN(X,
                    SØ)]T7(X,T)><SØ>L1                  ; IN(SUC(L),SØ).

I   * T12        := T1(SUC(L),T11)                      ; SOME([X,
                                                          NAT]PROP1(SUC(L),X)).

I   * M          := EB                                  ; NAT.

M   * Q          := EB                                  ; PROP1(SUC(L),M).

Q   * T13        := AX4(M,L,ANDE2(IN(M,S),
                    IS(SUC(M),SUC(L)),Q))               ; IS(M,L).

Q   * T14        := EQPRED1(M,L,T13,[X,
                    NAT]IN(X,S),ANDE1(IN(M,S),
                    IS(SUC(M),SUC(L)),Q))               ; IN(L,S).

I   * T15        := SOMEE([X,NAT]PROP1(SUC(L),
                    X),IN(L,S),T12,[X,NAT][T,
                    PROP1(SUC(L),X)]T14(X,T))           ; IN(L,S).

-2.

L1  * TH2        := [S,SET][T,                                          (60)
                    PROGRESSIVE(S)][U,IN(K,
                    S)]T15"-2"(S,T,U)                   ; LE(K,L).
```

```
L   * L1          := EB                                  ; LE(K,L).

L1  * N           := EB                                  ; NOT(IS(K,L)).

+3.

N   * S           := EB                                  ; SET.

S   * P           := EB                                  ; PROGRESSIVE(S).

P   * I           := EB                                  ; IN(SUC(K),S).

I   * SØ          := SETOF([X,NAT]OR(IS(K,X),
                     IN(X,S)))                           ; SET.

I   * M           := EB                                  ; NAT.

M   * J           := EB                                  ; IN(M,SØ).

J   * T1          := INE([X,NAT]OR(IS(K,X),
                     IN(X,S)),M,J)                       ; OR(IS(K,M),IN(M,S)).

J   * IØ          := EB                                  ; IS(K,M).

IØ  * T2          := EQPRED1(K,M,IØ,[X,
                     NAT]IN(SUC(X),S),I)                 ; IN(SUC(M),S).

J   * I1          := EB                                  ; IN(M,S).

I1  * T3          := <I1><M>P                            ; IN(SUC(M),S).

J   * T4          := ORE(IS(K,M),IN(M,S),
                     IN(SUC(M),S),T1,[T,IS(K,
                     M)]T2(T),[T,IN(M,S)]T3(T))          ; IN(SUC(M),S).

J   * T5          := INI([X,NAT]OR(IS(K,X),
                     IN(X,S)),SUC(M),ORI2(IS(K,
                     SUC(M)),IN(SUC(M),S),T4))           ; IN(SUC(M),SØ).

I   * T6          := INI([X,NAT]OR(IS(K,X),
                     IN(X,S)),K,ORI1(IS(K,K),
                     IN(K,S),REFLEQ(K)))                 ; IN(K,SØ).

I   * T7          := <T6><[X,NAT][T,IN(X,
                     SØ)]T5(X,T)><SØ>L1                  ; IN(L,SØ).

I   * T8          := NOTCASE1(IS(K,L),IN(L,S),
                     T1(L,T7),N)                         ; IN(L,S).

-3.

N   * TH3         := [S,SET][T,                                        (61)
                     PROGRESSIVE(S)][U,
                     IN(SUC(K),S)]T8"-3"(S,T,U)          ; LE(SUC(K),L).
```

```
L    * L1          := EB                                  ; LE(K,SUC(L)).

L1   * N           := EB                                  ; NOT(IS(K,SUC(L))).

N    * COR3A       := TH2(K,L,TH3(K,SUC(L),L1,
                     N))                                  ; LE(K,L).          (62)

L1   * COR3B       := [T,NOT(IS(K,
                     SUC(L)))]COR3A(T)                    ; OR(IS(K,SUC(L)),LE(K,
                                                            L)).              (63)

+4.

     * SØ          := SETOF([X,NAT]SOME([Y,
                     NAT]IS(SUC(Y),X)))                   ; SET.

K    * T1          := SOMEI([X,NAT]IS(SUC(X),
                     SUC(K)),K,REFLEQ(SUC(K)))            ; SOME([X,
                                                            NAT]IS(SUC(X),
                                                            SUC(K))).

K    * T2          := INI([X,NAT]SOME([Y,
                     NAT]IS(SUC(Y),X)),SUC(K),
                     T1)                                  ; IN(SUC(K),SØ).

K    * L1          := EB                                  ; LE(SUC(K),1).

L1   * T3          := <T2><[X,NAT][T,IN(X,
                     SØ)]T2(X)><SØ>L1                     ; IN(1,SØ).

L1   * T4          := INE([X,NAT]SOME([Y,
                     NAT]IS(SUC(Y),X)),1,T3)              ; SOME([X,
                                                            NAT]IS(SUC(X),1)).

L1   * T5          := <[X,NAT]AX3(X)>T4                   ; CON.

-4.

K    * TH4         := [T,LE(SUC(K),1)]T5"-4"(T)           ; NOT(LE(SUC(K),1)).  (64)

K    * L1          := EB                                  ; LE(K,1).

L1   * COR4        := DBLNEGLAW(IS(K,1),
                     CONTRAPOS(NOT(IS(K,1))),
                     LE(SUC(K),1),[T,NOT(IS(K,
                     1))]TH3(K,1,L1,T),TH4(K)))           ; IS(K,1).          (65)
```

```
+5.

K   * PROP1        := NOT(LE(SUC(K),K))                    ; PROP.

    * T1           := TH4(1)                               ; PROP1(1).

K   * P            := EB                                   ; PROP1(K).

P   * T2           := CONTRAPOS(LE(SUC(SUC(K)),
                      SUC(K)),LE(SUC(K),K),[T,
                      LE(SUC(SUC(K)),
                      SUC(K))]TH2(SUC(K),K,T),P)  ; PROP1(SUC(K)).

-5.

K   * TH5          := INDUCTION([X,                                    (66)
                      NAT]PROP1"-5"(X),T1"-5",
                      [X,NAT][T,
                      PROP1"-5"(X)]T2"-5"(X,T),
                      K)                                  ; NOT(LE(SUC(K),K)).

L   * L1           := EB                                   ; LE(K,L).

L1  * COR5         := CONTRAPOS(IS(K,SUC(L)),                          (67)
                      LE(SUC(L),L),[T,IS(K,
                      SUC(L))]EQPRED1(K,SUC(L),
                      T,[X,NAT]LE(X,L),L1),
                      TH5(L))                             ; NOT(IS(K,SUC(L))).
```

```
+THEOREM.

    * N          := EB                                   ; NAT.

N   * F          := EB                                   ; [X,NAT]NAT.

F   * BOUNDEDBY  := ALL([X,NAT]LE(<X>F,N))               ; PROP.          (68)

F   * INJECTIVE  := ALL([X,NAT][T,LE(X,                                   (69)
                    N)]ALL([Y,NAT][U,LE(Y,
                    N)]IMP(IS(<X>F,<Y>F),IS(X,
                    Y))))                                ; PROP.

F   * K          := EB                                   ; NAT.

K   * IMAGE      := SOME([X,NAT]AND(LE(X,N),                              (70)
                    IS(<X>F,K)))                         ; PROP.

F   * SURJECTIVE := ALL([X,NAT][T,LE(X,                                  (71)
                    N)]IMAGE(X))                         ; PROP.

N   * INJTHENSURJ := [F,[X,NAT]NAT][T,                                   (72)
                    BOUNDEDBY(N,
                    F)]IMP(INJECTIVE(N,F),
                    SURJECTIVE(N,F))                     ; PROP.


+1.

    * F          := EB                                   ; [X,NAT]NAT.

F   * BD         := EB                                   ; BOUNDEDBY(1,F).

BD  * K          := EB                                   ; NAT.

K   * L1         := EB                                   ; LE(K,1).

L1  * T1         := CONVEQ(<1>F,K,1,COR4(<1>F,
                    <1>BD),COR4(K,L1))                   ; IS(<1>F,K).

L1  * T2         := ANDI(LE(1,1),IS(<1>F,K),
                    REFLLE(1),T1)                        ; AND(LE(1,1),IS(<1>F,
                                                           K)).

L1  * T3         := SOMEI([X,NAT]AND(LE(X,1),
                    IS(<X>F,K)),1,T2)                    ; IMAGE(1,F,K).

BD  * T4         := [X,NAT][T,LE(X,1)]T3(X,T)            ; SURJECTIVE(1,F).

    * T5         := [F,[X,NAT]NAT][T,
                    BOUNDEDBY(1,F)][U,
                    INJECTIVE(1,F)]T4(F,T)               ; INJTHENSURJ(1).
```

```
N   * P          := EB                                  ; INJTHENSURJ(N).

P   * F          := EB                                  ; [X,NAT]NAT.

F   * BD         := EB                                  ; BOUNDEDBY(SUC(N),F).

BD  * INJ        := EB                                  ; INJECTIVE(SUC(N),F).

INJ * I          := EB                                  ; IS(<SUC(N)>F,SUC(N)).

I   * K          := EB                                  ; NAT.

K   * A          := IFTHENELSE(IS(<K>F,
                    SUC(N)),N,<K>F)                     ; NAT.

K   * J          := EB                                  ; IS(<K>F,SUC(N)).

J   * T6         := THEN(IS(<K>F,SUC(N)),N,
                    <K>F,J)                             ; IS(A,N).

J   * T7         := EQPRED2(A,N,T6,[X,
                    NAT]LE(X,N),REFLLE(N))              ; LE(A,N).

K   * NJ         := EB                                  ; NOT(IS(<K>F,SUC(N))).

NJ  * T8         := ELSE(IS(<K>F,SUC(N)),N,
                    <K>F,NJ)                            ; IS(A,<K>F).

NJ  * T9         := EQPRED2(A,<K>F,T8,[X,
                    NAT]LE(X,N),COR3A(<K>F,N,
                    <K>BD,NJ))                          ; LE(A,N).

K   * T10        := ANYCASE(IS(<K>F,SUC(N)),
                    LE(A,N),[T,IS(<K>F,
                    SUC(N))]T7(T),[T,
                    NOT(IS(<K>F,
                    SUC(N)))]T9(T))                     ; LE(A,N).
```

```
K   * L1        := EB                                   ; LE(K,N).

L1  * J         := EB                                   ; IS(<K>F,SUC(N)).

J   * T11       := CONVEQ(<K>F,<SUC(N)>F,
                   SUC(N),J,I)                          ; IS(<K>F,<SUC(N)>F).

J   * T12       := <T11><REFLLE(SUC(N))><SUC(
                   N)><COR1(K,N,L1)><K>INJ              ; IS(K,SUC(N)).

L1  * T13       := CONTRAPOS(IS(<K>F,SUC(N)),
                   IS(K,SUC(N)),[T,IS(<K>F,
                   SUC(N))]T12(T),COR5(K,N,
                   L1))                                 ; NOT(IS(<K>F,SUC(N))).

L1  * L         := EB                                   ; NAT.

L   * L2        := EB                                   ; LE(L,N).

L2  * J         := EB                                   ; IS(A(K),A(L)).

J   * T14       := TR3EQ(<K>F,A(K),A(L),<L>F,
                   SYMEQ(A,<K>F,T8(T13)),J,
                   T8(L,T13(L,L2)))                     ; IS(<K>F,<L>F).

J   * T15       := <T14><COR1(L,N,
                   L2)><L><COR1(K,N,
                   L1)><K>INJ                           ; IS(K,L).

I   * G         := [X,NAT]A(X)                          ; [X,NAT]NAT.

I   * T16       := [X,NAT]T10(X)                        ; BOUNDEDBY(N,G).

I   * T17       := [X,NAT][T,LE(X,N)][Y,
                   NAT][U,LE(Y,N)][V,IS(<X>G,
                   <Y>G)]T15(X,T,Y,U,V)                 ; INJECTIVE(N,G).
```

```
K   * L1        := EB                                   ; LE(K,SUC(N)).

L1  * J         := EB                                   ; IS(K,SUC(N)).

J   * T18       := CONVEQ(<SUC(N)>F,K,SUC(N),
                   I,J)                                 ; IS(<SUC(N)>F,K).

J   * T19       := ANDI(LE(SUC(N),SUC(N)),
                   IS(<SUC(N)>F,K),
                   REFLLE(SUC(N)),T18)                  ; AND(LE(SUC(N),
                                                          SUC(N)),IS(<SUC(N)>F,
                                                          K)).

J   * T20       := SOMEI([X,NAT]AND(LE(X,
                   SUC(N)),IS(<X>F,K)),
                   SUC(N),T19)                          ; IMAGE(SUC(N),F,K).

L1  * L2        := EB                                   ; LE(K,N).

L2  * T21       := <L2><K><T17><T16><G>P                ; IMAGE(N,G,K).

L2  * L         := EB                                   ; NAT.

L   * A1        := EB                                   ; AND(LE(L,N),IS(<L>G,
                                                          K)).

A1  * T22       := ANDE1(LE(L,N),IS(<L>G,K),
                   A1)                                  ; LE(L,N).

A1  * T23       := ANDE2(LE(L,N),IS(<L>G,K),
                   A1)                                  ; IS(<L>G,K).

A1  * T24       := DIVEQ(<L>F,K,<L>G,T8(L,
                   T13(L,T22)),T23)                     ; IS(<L>F,K).

A1  * T25       := ANDI(LE(L,SUC(N)),IS(<L>F,
                   K),CORI(L,N,T22),T24)                ; AND(LE(L,SUC(N)),
                                                          IS(<L>F,K)).

A1  * T26       := SOMEI([X,NAT]AND(LE(X,
                   SUC(N)),IS(<X>F,K)),L,T25)           ; IMAGE(SUC(N),F,K).

L2  * T27       := SOMEE([X,NAT]AND(LE(X,N),
                   IS(<X>G,K)),IMAGE(SUC(N),
                   F,K),T21,[X,NAT][T,
                   AND(LE(X,N),IS(<X>G,
                   K))]T26(X,T))                        ; IMAGE(SUC(N),F,K).

L1  * T28       := ORE(IS(K,SUC(N)),LE(K,N),
                   IMAGE(SUC(N),F,K),COR3B(K,
                   N,L1),[T,IS(K,
                   SUC(N))]T20(T),[T,LE(K,
                   N)]T27(T))                           ; IMAGE(SUC(N),F,K).

I   * T29       := [X,NAT][T,LE(X,
                   SUC(N))]T28(X,T)                     ; SURJECTIVE(SUC(N),F).
```

```
INJ * L1          := EB                                   ; LE(<SUC(N)>F,N).

L1  * K           := EB                                   ; NAT.

K   * B           := IFTHENELSE(IS(<K>F,
                     SUC(N)),<SUC(N)>F,<K>F)              ; NAT.

K   * I           := EB                                   ; IS(<K>F,SUC(N)).

I   * T30         := THEN(IS(<K>F,SUC(N)),
                     <SUC(N)>F,<K>F,I)                    ; IS(B,<SUC(N)>F).

I   * T31         := EQPRED2(B,<SUC(N)>F,T30,
                     [X,NAT]LE(X,N),L1)                   ; LE(B,N).

K   * NI          := EB                                   ; NOT(IS(<K>F,SUC(N))).

NI  * T32         := ELSE(IS(<K>F,SUC(N)),
                     <SUC(N)>F,<K>F,NI)                   ; IS(B,<K>F).

NI  * T33         := EQPRED2(B,<K>F,T32,[X,
                     NAT]LE(X,N),COR3A(<K>F,N,
                     <K>BD,NI))                           ; LE(B,N).

K   * T34         := ANYCASE(IS(<K>F,SUC(N)),
                     LE(B,N),[T,IS(<K>F,
                     SUC(N))]T31(T),[T,
                     NOT(IS(<K>F,
                     SUC(N)))]T33(T))                     ; LE(B,N).

K   * L2          := EB                                   ; LE(K,N).

L2  * I           := EB                                   ; IS(B,<SUC(N)>F).

I   * NJ          := EB                                   ; NOT(IS(<K>F,SUC(N))).

NJ  * T35         := DIVEQ(<K>F,<SUC(N)>F,B,
                     T32(NJ),I)                           ; IS(<K>F,<SUC(N)>F).

NJ  * T36         := <T35><REFLLE(SUC(N))><SUC(
                     N)><COR1(K,N,L2)><K>INJ              ; IS(K,SUC(N)).

I   * T37         := DBLNEGLAW(IS(<K>F,SUC(N)),
                     CONTRAPOS(NOT(IS(<K>F,
                     SUC(N))),IS(K,SUC(N)),[T,
                     NOT(IS(<K>F,
                     SUC(N)))]T36(T),COR5(K,N,
                     L2)))                                ; IS(<K>F,SUC(N)).

L2  * NI          := EB                                   ; NOT(IS(B,<SUC(N)>F)).

NI  * T38         := T32(CONTRAPOS(IS(<K>F,
                     SUC(N)),IS(B,<SUC(N)>F),
                     [T,IS(<K>F,SUC(N))]T30(T),
                     NI))                                 ; IS(B,<K>F).
```

```
L2  * L     := EB                                   ; NAT.

L   * L3    := EB                                   ; LE(L,N).

L3  * I     := EB                                   ; IS(B(K),B(L)).

I   * J     := EB                                   ; IS(B(K),<SUC(N)>F).

J   * T39   := DIVEQ(B(L),<SUC(N)>F,B(K),
               I,J)                                 ; IS(B(L),<SUC(N)>F).

J   * T40   := CONVEQ(<K>F,<L>F,SUC(N),
               T37(J),T37(L,L3,T39))                ; IS(<K>F,<L>F).

I   * NJ    := EB                                   ; NOT(IS(B(K),
                                                      <SUC(N)>F)).

NJ  * T41   := EQPRED1(B(K),B(L),I,[X,
               NAT]NOT(IS(X,<SUC(N)>F)),
               NJ)                                  ; NOT(IS(B(L),
                                                      <SUC(N)>F)).

NJ  * T42   := TR3EQ(<K>F,B(K),B(L),<L>F,
               SYMEQ(B,<K>F,T38(NJ)),I,
               T38(L,L3,T41))                       ; IS(<K>F,<L>F).

I   * T43   := ANYCASE(IS(B(K),
               <SUC(N)>F),IS(<K>F,<L>F),
               [T,IS(B(K),
               <SUC(N)>F)]T40(T),[T,
               NOT(IS(B(K),
               <SUC(N)>F))]T42(T))                  ; IS(<K>F,<L>F).

I   * T44   := <T43><COR1(L,N,
               L3)><L><COR1(K,N,
               L2)><K>INJ                           ; IS(K,L).

L1  * H     := [X,NAT]B(X)                          ; [X,NAT]NAT.

L1  * T45   := [X,NAT]T34(X)                        ; BOUNDEDBY(N,H).

L1  * T46   := [X,NAT][T,LE(X,N)][Y,
               NAT][U,LE(Y,N)][V,IS(<X>H,
               <Y>H)]T44(X,T,Y,U,V)                 ; INJECTIVE(N,H).
```

```
K   * L2          := EB                                   ; LE(K,SUC(N)).

L2  * I           := EB                                   ; IS(K,SUC(N)).

I   * T47         := <L1><<SUC(N)>F><T46><T45><
                     H>P                                  ; IMAGE(N,H,<SUC(N)>F).

I   * L           := EB                                   ; NAT.

L   * A1          := EB                                   ; AND(LE(L,N),IS(<L>H,
                                                            <SUC(N)>F)).

A1  * T48         := ANDE1(LE(L,N),IS(<L>H,
                     <SUC(N)>F),A1)                       ; LE(L,N).

A1  * T49         := ANDE2(LE(L,N),IS(<L>H,
                     <SUC(N)>F),A1)                       ; IS(<L>H,<SUC(N)>F).

A1  * T5Ø         := CONVEQ(<L>F,K,SUC(N),
                     T37(L,T48,T49),I)                    ; IS(<L>F,K).

A1  * T51         := ANDI(LE(L,SUC(N)),IS(<L>F,
                     K),COR1(L,N,T48),T5Ø)                ; AND(LE(L,SUC(N)),
                                                            IS(<L>F,K)).

A1  * T52         := SOMEI([X,NAT]AND(LE(X,
                     SUC(N)),IS(<X>F,K)),L,T51)           ; IMAGE(SUC(N),F,K).

I   * T53         := SOMEE([X,NAT]AND(LE(X,N),
                     IS(<X>H,<SUC(N)>F)),
                     IMAGE(SUC(N),F,K),T47,[X,
                     NAT][T,AND(LE(X,N),
                     IS(<X>H,<SUC(N)>F))]T52(X,
                     T))                                  ; IMAGE(SUC(N),F,K).
```

```
L2  * L3        := EB                                   ; LE(K,N).

L3  * I         := EB                                   ; IS(K,<SUC(N)>F).

I   * T54       := ANDI(LE(SUC(N),SUC(N)),
                   IS(<SUC(N)>F,K),
                   REFLLE(SUC(N)),SYMEQ(K,
                   <SUC(N)>F,I))                        ; AND(LE(SUC(N),
                                                          SUC(N)),IS(<SUC(N)>F,
                                                          K)).

I   * T55       := SOMEI([X,NAT]AND(LE(X,
                   SUC(N)),IS(<X>F,K)),
                   SUC(N),T54)                          ; IMAGE(SUC(N),F,K).

L3  * NI        := EB                                   ; NOT(IS(K,<SUC(N)>F)).

NI  * T56       := <L3><K><T46><T45><H>P                ; IMAGE(N,H,K).

NI  * L         := EB                                   ; NAT.

L   * A1        := EB                                   ; AND(LE(L,N),IS(<L>H,
                                                          K)).

A1  * T57       := ANDE1(LE(L,N),IS(<L>H,K),
                   A1)                                  ; LE(L,N).

A1  * T58       := ANDE2(LE(L,N),IS(<L>H,K),
                   A1)                                  ; IS(<L>H,K).

A1  * T59       := EQPRED2(<L>H,K,T58,[X,
                   NAT]NOT(IS(X,<SUC(N)>F)),
                   NI)                                  ; NOT(IS(<L>H,
                                                          <SUC(N)>F)).

A1  * T60       := DIVEQ(<L>F,K,<L>H,T38(L,
                   T57,T59),T58)                        ; IS(<L>F,K).

A1  * T61       := ANDI(LE(L,SUC(N)),IS(<L>F,
                   K),COR1(L,N,T57),T60)                ; AND(LE(L,SUC(N)),
                                                          IS(<L>F,K)).

A1  * T62       := SOMEI([X,NAT]AND(LE(X,
                   SUC(N)),IS(<X>F,K)),L,T61)           ; IMAGE(SUC(N),F,K).

NI  * T63       := SOMEE([X,NAT]AND(LE(X,N),
                   IS(<X>H,K)),IMAGE(SUC(N),
                   F,K),T56,[X,NAT][T,
                   AND(LE(X,N),IS(<X>H,
                   K))]T62(X,T))                        ; IMAGE(SUC(N),F,K).

L3  * T64       := ANYCASE(IS(K,<SUC(N)>F),
                   IMAGE(SUC(N),F,K),[T,IS(K,
                   <SUC(N)>F)]T55(T),[T,
                   NOT(IS(K,
                   <SUC(N)>F))]T63(T))                  ; IMAGE(SUC(N),F,K).
```

```
L2  * T65        := ORE(IS(K,SUC(N)),LE(K,N),
                    IMAGE(SUC(N),F,K),COR3B(K,
                    N,L2),[T,IS(K,
                    SUC(N))]T53(T),[T,LE(K,
                    N)]T64(T))                    ; IMAGE(SUC(N),F,K).

Li  * T66        := [X,NAT][T,LE(X,
                    SUC(N))]T65(X,T)              ; SURJECTIVE(SUC(N),F).

INJ * T67        := ORE(IS(<SUC(N)>F,SUC(N)),
                    LE(<SUC(N)>F,N),
                    SURJECTIVE(SUC(N),F),
                    COR3B(<SUC(N)>F,N,
                    <SUC(N)>BD),[T,
                    IS(<SUC(N)>F,
                    SUC(N))]T29(T),[T,
                    LE(<SUC(N)>F,N)]T66(T))       ; SURJECTIVE(SUC(N),F).

P   * T68        := [F,[X,NAT]NAT][T,
                    BOUNDEDBY(SUC(N),F)][U,
                    INJECTIVE(SUC(N),F)]T67(F,
                    T,U)                          ; INJTHENSURJ(SUC(N)).

-1.

N   * THEOREM    := INDUCTION([X,                                          (73)
                    NAT]INJTHENSURJ(X),T5"-1",
                    [X,NAT][T,
                    INJTHENSURJ(X)]T68"-1"(X,
                    T),N)                         ; [F,[X,NAT]NAT][T,
                                                    BOUNDEDBY(N,
                                                    F)]IMP(INJECTIVE(N,
                                                    F),SURJECTIVE(N,F)).

-THEOREM.

-NATURALS.

-LOGIC.
```

Technical data

The text was checked by the verifying program described by Zandleven [2] on a Burroughs B6700 at the Technological University, Eindhoven, operated by B. Jonker and Mrs. H. v. Helden. The verification took 62 seconds machine time. There were 201 applications of β-reduction, 390 of δ-reduction and none of η-reduction (in fact the checking was done for the η-free part of AUT-QE).

Glossary

Identifier	Ref.*	Meaning	
all	23	for all $x \in N$ P(x) holds	$\forall x \in N\,[P(x)]$.
and	18	A and B	$A \wedge B$.
ande1	20	"and"-elimination rules	$\frac{A \wedge B}{A}$.
ande2	21		$\frac{A \wedge B}{B}$.
andi	19	"and"-introduction rule	$\frac{A,\ B}{A \wedge B}$.
anycase	7	in any case (i.e. whether or not A is true) B holds	$\frac{A \Rightarrow B,\ \sim A \Rightarrow B}{B}$.
ax2	49	Peano's axioms 2 to 5	$\frac{k = \ell}{k' = \ell'}$.
ax3	50		$\frac{}{k' \neq 1}$.
ax4	51		$\frac{k' = \ell'}{k = \ell}$.
ax5	53		$\frac{\text{progressive}(s),\ 1 \in s}{k \in s}$.
axindividual	39	axiom for "individual"(cf. "individual")	$\frac{\exists!x \in N\,[P(x)]}{P(\text{no})}$.
boundedby	67	the values of f are bounded above by n	$\forall x \in N\,[f(x) \leq n]$.
con	2	contradiction	
contrapos	4	contraposition	$\frac{A \Rightarrow B,\ \sim B}{\sim A}$.
conveq	33	equality by convergence	$\frac{k = m,\ \ell = m}{k = \ell}$.
cor1	59	corollaries of theorems on the naturals	$\frac{k \leq \ell}{k \leq \ell'}$.
cor3a	62		$\frac{k \leq \ell',\ k \neq \ell'}{k \leq \ell}$.
cor3b	63		$\frac{k \leq \ell'}{k = \ell' \vee k \leq \ell}$.
cor4	65		$\frac{k \leq 1}{k = 1}$.
cor5	67		$\frac{k \leq \ell}{k \neq \ell'}$.
dblneglaw	6	double negation law	$\frac{\sim\sim A}{A}$.
diveq	34	equality by divergence	$\frac{m \neq k,\ m = \ell}{k = \ell}$.
else	42	second theorem on "if then else" (cf. "if then else" and "then")	$\frac{\sim A}{\textit{if } A \textit{ then } k \textit{ else } \ell = \ell}$.
eqpred1	29	equality preserves predicates	$\frac{k = \ell,\ P(k)}{P(\ell)}$.
eqpred2	31		$\frac{k = \ell,\ P(\ell)}{P(k)}$.
ifthenelse	40	the natural number which is k if A is true and ℓ if A is false	*if* A *then* k *else* ℓ.
imp	1	A implies B	$A \Rightarrow B$.
image	70	k is image under the restriction of f to [1,n]	$\exists x \in N\,[x \leq n \wedge f(x) = k]$
in	44	k is in s, i.e. k is an element of s	$k \in s$.
individual	38	the unique natural no for which P holds	$\frac{\exists!x \in N\,[P(x)]}{\text{no}}$.
induction	54	induction for predicates	$\frac{P(1),\ \forall x \in N\,[P(x) \Rightarrow P(x')]}{P(k)}$.
ine	47	"in"-elimination rule (cf. "in")	$\frac{k \in \{x \in N \mid P(x)\}}{P(k)}$.
ini	46	"in"-introduction rule (cf. "in")	$\frac{P(k)}{k \in \{x \in N \mid P(x)\}}$.
injective	69	the restriction of f to [1,n] is injective	$\forall x \leq n\ \forall y \leq n\,[f(x) = f(y) \Rightarrow x = y]$.

* this indicates the line in the AUT-QE text where the identifier is introduced.

injthensurj	72	the property to be proved: if f is bounded by n and f is injective on [1,n] then f is surjective on [1,n]	$\forall f\colon N \to N\ [f \text{ bounded by } n \Rightarrow (f \text{ injective} \Rightarrow f \text{ surjective})]$.
is	27	k is ℓ, i.e. k equals ℓ	$k = \ell$.
le	55	k is less then or equal to ℓ	$k \leq \ell$.
nat	22	the type of the naturals	N.
not	3	not A	$\sim A$.
notcase1	15	we have not case 1, so case 2 holds	$\frac{A \vee B,\ \sim A}{B}$.
notcase2	16	we have not case 2, so case 1 holds	$\frac{A \vee B,\ \sim B}{A}$.
nottwo	36	P does not hold for two different naturals i.e. if P(x) and P(y) then x = y	$!x \in N\ [P(x)]$.
one	37	there is a unique x ∈ N for which P holds	$\exists! x \in N\ [P(x)]$.
or	12	A or B	$A \vee B$.
ore	17	"or"-elimination rule	$\frac{A \vee B,\ A \Rightarrow C,\ B \Rightarrow C}{C}$.
ori1	13	"or"-introduction rules	$\frac{A}{A \vee B}$.
ori2	14		$\frac{B}{A \vee B}$.
progressive	52	s is progressive, i.e. if x ∈ S then x' ∈ S	$\forall x \in N\ [x \in S \Rightarrow x' \in S]$.
refleq	28	reflexivity of equality	$\frac{}{k = k}$.
refl1e	56	reflexivity of less-or-equal	$\frac{}{k \leq k}$.
set	43	type of sets of naturals	$P(N)$.
setof	45	the set of naturals for which P holds	$\{x \in N \mid P(x)\}$.
some	24	for some x ∈ N P(x) holds	$\exists x \in N\ [P(x)]$.
somee	26	existence-elimination rule	$\frac{\exists x \in N\ [P(x)],\ \forall x \in N\ [P(x) \Rightarrow A]}{A}$.
somei	25	existence-introduction rule	$\frac{P(k)}{\exists x \in N\ [P(x)]}$.
suc	48	successor of k	k'.
surjective	71	the restriction of f to [1,n] is surjective	$\forall x \leq n\ [\text{image}(n,f,x)]$.
symeq	30	symmetry of equality	$\frac{k = \ell}{\ell = k}$.
th1	5	in § logic: some theorems and rules of logic	$\frac{A}{\sim\sim A}$.
th2	8		$\frac{\sim A}{A \Rightarrow B}$.
th3	9		$\frac{A,\ \sim B}{\sim(A \Rightarrow B)}$.
th4	10		$\frac{\sim(A \Rightarrow B)}{A}$.
th5	11		$\frac{\sim(A \Rightarrow B)}{\sim B}$.
th1	58	in § naturals: some theorems on order in the naturals	$\frac{}{k \leq k'}$.
th2	60		$\frac{k' \leq \ell'}{k \leq \ell}$.
th3	61		$\frac{k \leq \ell,\ k \neq \ell}{k' \leq \ell}$.
th4	64		$\frac{}{\sim(k' \leq 1)}$.
th5	66		$\frac{}{\sim(k' \leq k)}$.

theorem	23	the theorem proved: if f is bounded by n and f is injective on [1,n] then f is surjective on [1,n]	$\forall f\colon N \to N\,[f \text{ bounded by } n \Rightarrow (f \text{ injective} \Rightarrow f \text{ surjective})]$.
then	41	first theorem on "if then else" (cf. "ifthenelse" and "if")	$\dfrac{A}{\mathit{if}\ A\ \mathit{then}\ k\ \mathit{else}\ l = k}$.
treq	32	transitivity of equality	$\dfrac{k = l,\ l = m}{k = m}$.
tr3eq	35	transitivity for 3 equalities	$\dfrac{k = l,\ l = m,\ m = n}{k = n}$
tlee	57	transitivity of less-or-equal	$\dfrac{k \leq l,\ l \leq m}{k \leq m}$.

References

[1] Van Daalen, D.T.; A description of AUTOMATH and some aspects of its language theory; this volume.

[2] Zandleven, I.; A verifying program for AUTOMATH; this volume.

LIMA - PAL

par

Georges KIREMITDJIAN*

* Université PARIS-SUD
Laboratoire AL-KHOUARIZM,

INTRODUCTION

1°) Qu'est-ce qu'une théorie formalisée ?

Pour éviter les paradoxes et définir de manière plus précise l'objet de ses investigations, le mathématicien a dû se défier de la seule logique intuitive. Il a d'abord axiomatisé ses théories, mais ce n'était pas suffisant. Il ne distinguait pas encore clairement le langage utilisé pour définir la théorie objet et les phrases appartenant à cette même théorie.

Faire cette distinction, c'est formaliser la théorie. Pour cela, il faut construire une langue symbolique. Cette langue est déterminée par la donnée d'un alphabet, d'une grammaire et d'une logique. Pour définir tout cela, nous utilisons un autre langage, dans notre cas, ce métalangage sera AUTOMATH. Donc AUTOMATH permet de formaliser un texte mathématique en associant aux symboles du système formel une ou des interprétations. Le module ainsi construit sera satisfaisant si les théorèmes du système formel correspondent à des propriétés vraies de la théorie informelle.

Ce problème de l'interprétation ne concerne pas le métalangage AUTOMATH, mais seulement le langage objet qui est la formalisation de la théorie particulière envisagée.

2°) P.A.L

AUTOMATH est donc un langage symbolique permettant de formaliser les mathématiques. Il a été défini par N.6. De Bruijn, et est utilisé depuis 1967 (1).

Il existe un sous-langage d'AUTOMATH : P.A.L . L'extension de P.A.L à AUTOMATH se fait en utilisant une extension du lambda calcul typé de Church.

De fait, l'absence de la notion d'application fonctionnelle, limite considérablement le domaine mathématique formalisé par P.A.L . Cependant, le problème étant d'abord d'étudier un programme A.P.L permettant la vérification automatique des

démonstrations, nous nous sommes limités aux textes écrits en P.A.L. Ce travail pouvant être considéré comme une étape vers la résolution du problème général. En effet, les structures de base mises en oeuvre peuvent se prolonger pour englober la vérification de textes utilisant le lambda calcul.

3°) Automate construit sur ordinateur reconnaissant des textes en P.A.L

Nous verrons lors de la définition de P.A.L qu'un automate peut déterminer si une phrase appartient ou non au langage. Le problème est donc de construire cet automate. Pour ce faire, nous avons utilisé un ordinateur que nous avons programmé en A.P.L. L'exécution de ce programme permet de stocker en mémoire un livre écrit en P.A.L. Il demande d'entrer chaque ligne, l'accepte si elle est valide, sinon renvoie un message d'erreur. Cependant, l'automate réel ainsi construit est soumis à des limitations pragmatiques : temps de réponse raisonnable, encombrement mémoire limité...

Description de P.A.L

1°) Préliminaires.

P.A.L est un métalangage permettant de définir des théories formalisées. Un livre écrit en P.A.L s'interprète comme l'introduction du vocabulaire et de la grammaire d'une théorie formalisée, puis sur cette base, il permet de déduire des théorèmes. La logique de P.A.L est en fait, la structure commune à toutes ces théories formalisables. Pour cet ensemble de théories, il définit les métarègles qu'elles doivent toutes vérifier. Cela est vrai, plutôt pour AUTOMATH que pour P.A.L qui est trop restrictif. Nous allons tout d'abord décrire la structure de P.A.L, puis sa grammaire et sa logique, et ensuite l'utilisation de P.A.L.

2°) Structure - Définitions.

Un texte écrit en P.A.L est appelé LIVRE. La structure physique du livre est une séquence de LIGNES. La structure logique est une arborescence.

La structure de la ligne est la suivante :

IN ID DEF CAT

ID : identificateur qui est le nom donné à l'expression introduite par la ligne

DEF : la définition de cette expression

CAT : la catégorie qui indique le type de l'expression

IN : est un indicateur de contexte.

Chaque mot de P.A.L est représenté par son identificateur. Il y a deux sortes de mots : les variables et les constantes. Ces mots sont introduits dans le texte P.A.L suivant les besoins de la théorie dont il est la formalisation. Les variables sont définies par le symbole 'EB' en zone définition. Les constantes sont

- soit des éléments primitifs définis par 'PN' en zone définition
- soit des expressions formées de constantes et de variables déjà introduites.

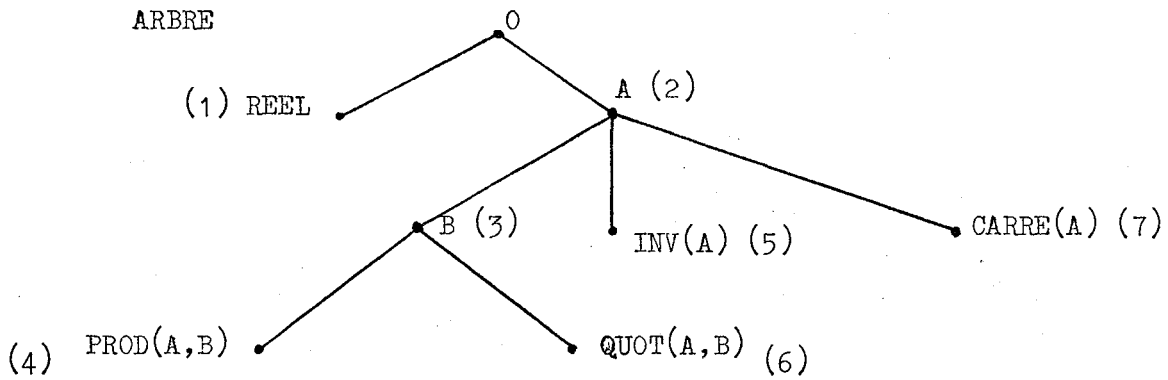
A chaque mot est donc associé une définition et une catégorie.

Les catégories sont définies comme variables ou constantes en mettant 'TYPE' en zone catégorie. Quand de nouvelles catégories sont définies, elles peuvent devenir la catégorie d'un mot en étant placées en zone catégorie de la ligne où est défini ce mot.

L'indicateur de contexte permet d'accrocher la ligne courante à l'arbre que forme le livre. Si la ligne n ' est dans aucun bloc, son indicateur est la racine de l'arbre : 0 . Si elle est dans le bloc ouvert par la variable a , son indicateur est a . Il peut y avoir plusieurs blocs imbriqués. Dans ce cas, on définit la liste des indicateurs de la ligne (LI(N)) . Cette liste correspond aux variables acceptables dans ce contexte. L'identificateur dépend de toutes ces variables.

Exemple de texte en P.A.L

N° de ligne	IN	ID	DEF	CAT
(1)	0	REEL	PN	TYPE
(2)	0	A	EB	REEL
(3)	A	B	EB	REEL
(4)	B	PROD	PN	REEL
(5)	A	INV	PN	REEL
(6)	B	QUOT	PROD(A,INV(B))	REEL
(7)	A	CARRE	PROD(A,A)	REEL



On aurait pu inverser les lignes (3) et (5), (5) et (7). Les permutations de l'arbre sont restreintes par les conditions de précédence dans la définition des objets successifs figurant aux noeuds.

Ainsi, on ne peut pas inverser (1) et (2) car A est de type REEL défini en ligne (1). De même QUOT dont la définition dépend de PROD et INV ne peut être placé plus tôt.

Voyons la liste d'indicateurs des lignes : c'est la suite des noeuds qu'il faut traverser pour remonter jusqu'à la racine. Ex pour INV : A et pour QUOT : B,A.

Interprétation :

ligne 1 : la définition PN : notion primitive ou axiome, indique que l'on définit une constante appelée REEL (zone.ID) indépendante de toute variable (hors contexte car IN = 0) qui sera un type.

ligne 2 : hors contexte (IN = 0), A(ID) est une variable (DEF = 'EB') réelle (CAT).

ligne 3 : dans le bloc A , B est une variable réelle.

ligne 4 : en fonction de A et B , PROD(A,B) est une notion primitive de type REEL.

ligne 5 : de même pour INV(A).

ligne 6 : en fonction de A et de B , QUOT(A,B) = PROD(A,INV(B)) est réel.

ligne 7 : en fonction de A , CARRE(A) = PROD(A,A) est un réel.

Dans ces deux derniers cas REEL semble redondant car on peut construire la catégorie à partir de la définition, QUOT = PROD( ) quotient et produit doivent donc avoir aussi le même type. Donc QUOT a le type de PROD : REEL.

Le travail se borne à vérifier que le type donné en zone catégorie correspond au type construit. Ceci est très important dans la suite car on utilise cette vérification pour constater qu'une démonstration donne le résultat voulu. La ligne prend la forme :

IN	ID	DEF	CAT
	Théorème	expression qui est la démonstration	énoncé du théorème

Remarque sur les substitutions à effectuer dans le contrôle de validité. Dans la ligne (6) on a PROD(A,INV(B)) alors que PROD est défini avec les paramètres A et B , donc le deuxième paramètre B est devenu INV(B) . Pour que la ligne 6 soit acceptable, il faut que B et INV(B) aient même type. De même en ligne (7), dans PROD,B devenant A , implique que A et B aient même type pour que la substitution soit valide.

Nous allons maintenant préciser la grammaire de P.A.L , puis son utilisation.

3°) Grammaire et logique de P.A.L

On peut distinguer trois niveaux de structure : le livre, la ligne et l'expression.

Le livre vide étant valide, un livre valide est défini récursivement par ajout de lignes valides. Une ligne valide, à la suite du livre 1 a pour indicateur

0 ou une variable définie dans le livre 1. Son identificateur peut être un mot quelconque de l'alphabet, à condition qu'il ne soit pas déjà utilisé. Sa définition peut être 'EB' , 'PN' ou une expression valide Σ_2 . Sa catégorie peut être 'TYPE' ou une expression valide Σ_1 . On veut de plus que la catégorie de Σ_1 soit 'TYPE' et celle de Σ_2 soit Σ_1 modulo certaines substitutions.

Reste à voir la définition d'expression valide dans le contexte θ .

Une variable x est admissible si elle appartient à la liste d'indicateurs de θ c'est à dire au contexte.

Une constante est admissible si le contexte de la ligne où elle est définie est inclus dans le contexte de θ .

Maintenant par récurrence nous allons définir une expression parenthésée admissible.

Si B est une constante définie dans le contexte $x_1, x_2 \ldots x_k$ on a donc dans le livre les lignes suivantes :

$$
\begin{array}{llll}
0 & x_1 & EB & \Gamma_1 \\
x_1 & x_2 & EB & \Gamma_2(x_1) \\
\hline
\hline
x_{k-1} & x_k & EB & \Gamma_k(x_1,\ldots,x_{k-1}) \\
x_k & B & \Omega(x_1,\ldots,x_k) & \Sigma(x_1,\ldots,x_{12})
\end{array}
$$

Alors $B(E_1,\ldots E_k)$ est admissible si

1°) tous les E_i sont admissibles

2°) les catégories des paramètres correspondants sont égaux modulo certaines substitutions.

$$
\begin{array}{lll}
CAT(E_1) \overline{\overline{D}} \Gamma_1 & & \\
CAT(E_2) \overline{\overline{D}} S_{x_1 \to E_1} & \Gamma_2(x_1) & \\
\qquad\qquad \text{on substitue } E_1 \text{ à } x_1 \text{ dans } \Gamma_2(x_1) & & \\
\hline
\hline
CAT(E_k) \overline{\overline{D}} S_{\substack{x_1 \to E_1 \\ \hline \\ x_{k-1} \to E_{k-1}}} & \Gamma_k(x_1,\ldots,x_k) &
\end{array}
$$

Alors la catégorie construite CACO de l'expression $B(E_1,\ldots E_k)$ est dans ces conditions :

si $\Omega = PN$: $S_{\substack{x_1 \to E_1 \\ \hline x_k \to E_k}} \quad \Sigma(x_1,\ldots,x_k)$

si $\Omega \neq PN$: $S_{\substack{x_1 \to E_1 \\ \hline x_k \to E_k}} \quad CACO(\Omega)$.

Note sur $A \underset{D}{=} B$.

Il se peut que dans la définition de A on utilise une constante qui soit elle même fonction d'autres constantes par sa définition

exemple : $A(x) = F(H(x),x)$ et $H(x) = G(x,x)$.

Supposons que F et G soient des notions primitives, alors on dit que A est mis sous forme développée quand

$$FD(A) = F(G(x,x),x)$$

car on ne peut plus remplacer une constante par sa définition.

$A \underset{D}{=} B$ est équivalent à $FD(A) = FD(B)$.

Comme cela, on est sûr que A et B sont logiquement équivalents, même s'ils ont des formes différentes.

Nous venons de voir un mode de définition basé sur la reconnaissance de phrases acceptables. Nous allons préciser cela en donnant une description formelle de la grammaire.

1°) Vocabulaire terminal

mots : éléments du semi-groupe engendré par l'alphabet : 'ABCDEFGHIJKLMNOPQRSTUVWX YZ 0 1 2 3 4 5 6 7 8 9'

symboles spéciaux : 'TYPE', 'PN', 'EB', 'O', 'FIN'

séparateurs : bl ( ) ,

2°) Vocabulaire non terminal

DEBUT , LIVRE , FIN , LIGNE , IN , CAT , ID , DEF , EXP , VAR , CONS , SEXP.

3°) AXIOME ou RACINE : DEBUT

Nous allons donner les règles de production et les règles sémantiques associées, ce à chaque niveau de structure.

NIVEAU LIVRE

Règles de production :

0 DEBUT → LIVRE

1 LIVRE 1 → LIGNE LIVRE 2

2 LIVRE → LIGNE FIN

DEBUT : marque l'initialisation du livre : livre vide

FIN : c'est le livre final

LIVRE 1 : est le livre déjà accepté

LIGNE : est la ligne que l'on soumet à vérification

LIVRE 2 : est le livre comprenant la nouvelle ligne.

C'est à LIVRE que sont associés tous les paramètres qui contiennent toute l'information sémantique à un instant donné.

Quels sont ces paramètres :

TV : table des variables

TC : table des constantes

TEXP : table des expressions

TCAT : table des catégories

TCACO : table des catégories construites

ARB : arbre du livre

CL : compteur de lignes.

LIGNE fait la liaison entre les niveaux de structure LIVRE et LIGNE.

Paramètres sémantiques de LIGNE.

LV : nom de la variable définie dans la ligne

LC : nom de la constante définie dans la ligne

LEXP : expression définie dans la ligne

LCAT : catégorie de la ligne

LCACO : catégorie construite de l'expression

LARB : indicateur

CONTEXTE de la ligne

DERNIERE permet le test de la fin de livre

Notation vide : Φ .

Règles sémantiques

⓪ DEBUT → LIVRE

TV(LIVRE) ← Φ

TC(LIVRE) ← Φ

TEXP(LIVRE) ← Φ

TCAT(LIVRE) ← Φ

TCACO(LIVRE) ← Φ

ARB(LIVRE) ← Φ

CL ← 1

① LIVRE 1 → LIGNE LIVRE 2

TV(LIVRE 2) ← TV(LIVRE 1) LV(LIGNE)

TC(LIVRE 2) ← TC(LIVRE 1) LC(LIGNE)

TEXP(LIVRE 2) ← TEXP(LIVRE 1) LEXP(LIGNE)

TCAT(LIVRE 2) ← TCAT(LIVRE 1) LCAT(LIGNE)

TCACO(LIVRE 2) ← TCACO(LIVRE 1) LCACO(LIGNE)

ARB(LIVRE 2) ← ARB(LIVRE 1) LARB(LIGNE)

CL(LIVRE 2) ← CL(LIVRE 1) + 1

Condition à tester :

(DERNIERE(LIGNE) = NON) ?

② LIVRE → LIGNE FIN

TV(FIN) → TV(LIVRE)

TC(FIN) ← TC(LIVRE)

TEXP(FIN) ← TEXP(LIVRE)

TCAT(FIN) ← TCAT(LIVRE)

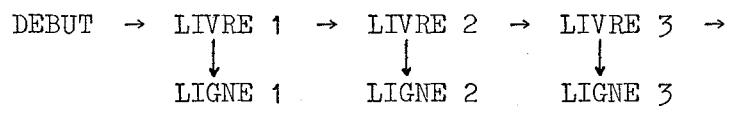
TCACO(FIN) ← TCACO(LIVRE)

ARB(FIN) ← ARB(LIVRE)

CL(FIN) ← CL(LIVRE)

(DERNIERE(LIGNE) = OUI) ?

Exemple



(1)	0	REEL	PN	TYPE
(2)	0	A	EB	REEL
(3)	A	B	EB	REEL

Pour le livre 1 toutes les tables sont vides et = 1 .

Pour la ligne 1 les paramètres sont transmis du niveau LIGNE.

LV(LIGNE 1) = Φ LC(LIGNE 1) = REEL

LEXP(LIGNE 1) = Φ LCAT(LIGNE 1) = 0

LCACO(LIGNE 1) = Φ LARB(LIGNE 1) = 0

Pour le livre 2

TV(LIVRE 2) = Φ TC(LIVRE 2) = REEL

TEXP(LIVRE 2) = Φ TCAT(LIVRE 2) = 0

TCACO(LIVRE 2) = Φ ARB(LIVRE 2) = 0

CL(LIVRE 2) = 2

Donc le livre 2 contient toute l'information sémantique apportée par la ligne 1.

ligne 2

LV(LIGNE 2) = A

LC(LIGNE 2) = Φ

LCAT(LIGNE 2) = REEL

LEXP(LIGNE 2) = Φ

LCACO(LIGNE 2) = Φ

LARB(LIGNE 2) = 0

livre 3

TV(LIVRE 3) = Φ , A

TC(LIVRE 3) = REEL , Φ

TCAT = 0 , REEL

CL(LIVRE 3) = 3

TEXP = Φ , Φ

TCACO = Φ , Φ

ARB = 0 , 0

L'information sémantique connue à un niveau est implicitement accessible à tout niveau inférieur de l'arbre. Il y a deux flots d'informations : celui qui résulte de la mémorisation du livre et qui descend l'arbre et celui qui résulte de l'entrée et de l'analyse de la ligne et qui remonte l'arbre jusqu'à livre où il est mémorisé.

NIVEAU LIGNE

Règles de production

③ LIGNE → IN bl ID bl DEF bl CAT

④ LIGNE → FIN

⑤ IN → 0

⑥ IN → VAR

⑦ CAT → TYPE

⑧ CAT → EXP

⑨ DEF → EB

⑩ DEF → PN

⑪ DEF → EXP

⑫ VAR → mot

⑬ CONS → mot

⑭ ID → mot

Règles sémantiques

⑭ ID → mot

mot représente ici un élément du semi groupe engendré par l'alphabet, c'est donc un métasymbole représentant un élément terminal.

NOM(ID) ← mot c'est l'élément terminal particulier représenté ici par mot qui est la valeur du paramètre NOM.

Au niveau livre nous avons déjà vu des conditions relatives aux paramètres sémantiques. Elles servaient plutôt à déterminer laquelle des deux règles 1 et 2 était applicable. Ici les conditions permettront plutôt de savoir si la ligne est acceptable.

Pour cette règle de production la condition est :

(NOM(ID) ∉ TV U TC) ?

⑬ CONS → mot

NOM(CONS) ← mot

(NOM(CONS) ∈ TC) ? → NL(CONS)

⑫ VAR → mot

NOM(VAR) ← mot

(NOM(VAR) ∈ TV) ? → NL(VAR)

NL(X) est le n° de ligne où est défini X , il est connu si la recherche dans la table aboutit.

⑪ DEF → EXP

EXPRESSION(DEF) ← EXPRESSION(EXP)

CATEGORIE(DEF) ← CATEGORIE(EXP)

TYPE(DEF) ← CONSTANTE

⑩ DEF → PN

EXPRESSION(DEF) ← Φ

CATEGORIE(DEF) ← Φ

TYPE(DEF) ← CONSTANTE

⑨ DEF → EB

EXPRESSION(DEF) ← Φ

CATEGORIE(DEF) ← Φ

TYPE(DEF) ← VARIABLE

⑧ CAT → EXP

EXPRESSION(CAT) ← EXPRESSION(EXP)

(CATEGORIE(EXP) = 0) ?

⑦ CAT → TYPE

EXPRESSION(CAT) ← 0

⑥ IN → VAR

NL(IN) ← NL(VAR)

NOM(IN) ← NOM(VAR)

⑤ IN → 0

NL(IN) ← 0

NOM(IN) ← 0

④ LIGNE → FIN

DERNIERE(LIGNE) ← OUI

③ LIGNE → IN bl ID bl DEF bl CAT

LV(LIGNE) ← {NOM(ID) si TYPE(DEF) = VARIABLE, Φ sinon} ?

LC(LIGNE) ← {NOM(ID) si TYPE(DEF) = CONSTANTE, Φ sinon} ?

LEXP(LIGNE) ← EXPRESSION(DEF)

LCAT(LIGNE) ← EXPRESSION(CAT)

LCACO(LIGNE) ← CATEGORIE(DEF)

LARB(LIGNE) ← NOM(IN)

CONTEXTE(LIGNE) ← LI(CL(LIVRE))

DERNIERE(LIGNE) ← NON

(CATEGORIE(DEF) $\overline{\overline{D}}$ EXPRESSION(CAT)) ?

LI(N) est une fonction qui donne la liste d'indicateurs de la ligne N , N $\leqslant$ CL(LIVRE) (voir le chapitre description d'A.P.L pour des détails sur cette fonction).

NIVEAU EXPRESSION

Règles de production

(15) EXP → VAR

(16) EXP → CONS

(17) EXP → CONS (SEXP)

(18) SEXP → EXP

(19) SEXP → EXP , SEXP

Règles sémantiques associées

(15) EXP → VAR

EXPRESSION(EXP) ← NOM(VAR)

CATEGORIE(EXP) ← TCAT[NL(VAR)]

(NOM(VAR) ∈ CONTEXTE) ?

(16) EXP → CONS

EXPRESSION(EXP) ← NOM(CONS)

CATEGORIE(EXP) ← TCAT[NL(CONS)]

(LI(NL(CONS)) ⊂ CONTEXTE(LIGNE)) ?

(17) EXP → CONS(SEXP)

EXPRESSION(EXP) ← NOM(CONS)(LISTEXP(SEXP))

π ← TCACO[NL(CONS)]

Σ ← TCAT[NL(CONS)]

Ω ← TEXP[NL(CONS)]

1°) Ω = PN

CATEGORIE(EXP) ← $S_{LI(NL(CONS)) \rightarrow LISTEXP(SEXP)}\ \Sigma$

2°) $\Omega \neq$ PN

CATEGORIE(EXP) ← $S_{LI(NL(CONS)) \rightarrow LISTEXP(SEXP)}\ \pi$.

Soient $E_1, E_2 \ldots, E_n$ = LISTEXP(SEXP)

$x_1, x_2 \ldots, x_n$ = LI(NL(CONS)) .

Les E_i sont des expressions dont la catégorie CATEGORIE(E_i) est le i^e élément de LISTCAT(SEXP).

On doit vérifier les conditions suivantes : compatibilité des catégories

$$
? \begin{cases}
\text{CATEGORIE}(E_1) \overset{=}{D} \text{CATEGORIE}(x_1) \\
\text{CATEGORIE}(E_2) \overset{=}{D} S_{x_1 \to E_1}\ \text{CATEGORIE}(x_2) \\
\vdots \\
\text{CATEGORIE}(E_{n-1}) \overset{=}{D} S_{x_1 \to E_1,\ x_2 \to E_2,\ \ldots,\ x_{n-2} \to E_{n-2}}\ \text{CATEGORIE}(x_{n-1}) \\
\text{CATEGORIE}(E_n) \overset{=}{D} S_{x_1 \to E_1,\ \ldots,\ x_{n-1} \to E_{n-1}}\ \text{CATEGORIE}(x_n)
\end{cases}
$$

⑱ SEXP → EXP

LISTEXP(SEXP) ← EXPRESSION(EXP)

LISTCAT(SEXP) ← CATEGORIE(EXP)

⑲ SEXP1 → EXP , SEXP2

LISTEXP(SEXP1) = EXPRESSION(EXP) LISTEXP(SEXP2)

LISTCAT(SEXP1) = CATEGORIE(EXP) LISTCAT(SEXP2) .

Revenons à l'exemple

Nous avons vu que chaque livre contient toute l'information précédemment introduite.

Repartons donc au livre 3

TV(LIVRE 3) = Φ , A

TC(LIVRE 3) = REEL , Φ

TCAT(LIVRE 3) = 0 , REEL

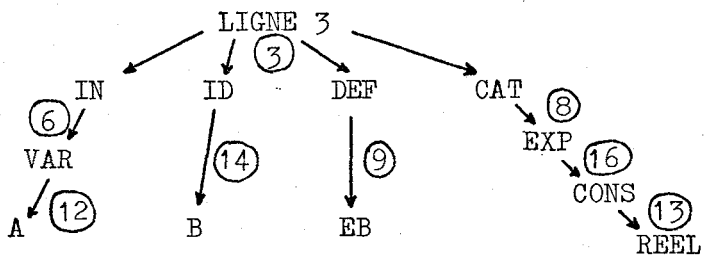
TEXP(LIVRE 3) = Φ , Φ

TCACO(LIVRE 3) = Φ , Φ

ARB(LIVRE 3) = 0 , 0

CL(LIVRE 3) = 3

introduction de la ligne 3 : A B EB REEL



Les paramètres sémantiques sont évalués en remontant l'arbre.

(12) NOM(VAR) = A NL(VAR) = 2

(6) NOM(IN) = NOM(VAR) = A NL(IN) = NL(VAR) = 2

(14) NOM(ID) = B (B $\notin$ A , REEL) ? oui

(9) EXPRESSION(DEF) = Φ

CATEGORIE(DEF) = Φ

TYPE(DEF) = VARIABLE

(13) NOM(CONS) = REEL NL(CONS) = 1

(16) EXPRESSION(EXP) = NOM(CONS) = REEL

CATEGORIE(EXP) = TCAT[NL(CONS)] = TCAT[1] = 0

LI(NL(CONS)) = LI(1) = Φ

CONTEXTE(LIGNE) pas encore défini

(8) EXPRESSION(CAT) = EXPRESSION(EXP) = REEL

(CATEGORIE(EXP) = 0) ? oui .

(3) TYPE(DEF) = VARIABLE

donc LV(LIGNE) = NOM(ID) = B LC(LIGNE) = Φ

LEXP(LIGNE) = EXPRESSION(DEF) = Φ

LCAT(LIGNE) = EXPRESSION(CAT) = REEL

LCACO(LIGNE) = CATEGORIE(DEF) = Φ

LARB(LIGNE) = NOM(IN) = A

CONTEXTE(LIGNE) = LI(3) = A

DERNIERE(LIGNE) = NON .

Revenons à 16

LI(NL(CONS)) ⊂ CONTEXTE(LIGNE) ? oui $\Phi \subset A$.

Comme dernière(ligne) = non c'est la règle 2 qui permet de continuer l'arbre donc

→ LIVRE 3 → LIVRE 4
↓
LIGNE 3 .

2 TV(LIVRE 4) = TV(LIVRE 3) LV(LIGNE 3)

TV(LIVRE 4) = Φ , A , B

de même

TC(LIVRE 4) = REEL , Φ , Φ

TCAT(LIVRE 4) = 0 , REEL , REEL

TEXP(LIVRE 4) = Φ , Φ , Φ

TCACO(LIVRE 4) = Φ , Φ , Φ

ARB(LIVRE 4) = 0 , 0 , A

CL(LIVRE 4) = 4 .

CONVENTION

Nous allons introduire une simplification d'écriture qui ne fait pas strictement partie du langage.

Si b est une constante de longueur k , on peut former une expression

$$b(E_1,\ldots,E_k) .$$

Ceci est la forme normale de l'expression. Voyons la forme simplifiée.

Si le contexte de la ligne est

$$x_1 , x_2,\ldots,x_k ,$$

et si $E_1 = x_1$, $E_2 = x_2,\ldots E_i = x_i$, $E_{i+1} \neq x_{i+1}$ alors on peut se contenter d'écrire

$$b(E_{i+1},\ldots E_k)$$

par exemple B de longueur 4

B est équivalent à FN(B) = $B(x_1,x_2,x_3,x_4)$

$B(x_1,E_1,x_3,E_2)$ est équivalent à

$B(E_1,x_3,E_2)$ sous forme réduite

mais pas à $B(E_1,E_2)$ dont la forme normale est $B(x_1,x_2,E_1,E_2)$.

4°) Utilisation de P.A.L

LIVRE

1 0 BOOL PN TYPE

2 0 B EB BOOL

3 B TRUE PN TYPE

4 0 X EB BOOL

5 X Y EB BOOL

6 Y AND PN BOOL

7 Y ASP1 EB TRUE(X)

8 ASP2 ASP2 EB TRUE(Y)

9 ASP2 AX1 PN TRUE(AND)

10 Y ASP3 EB TRUE(AND)

11 ASP3 AX2 PN TRUE(X)

12 ASP3 AX3 PN TRUE(Y)

13 ASP3 TH AX1(Y,X,AX3,AX2) TRUE(AND(Y,X))

Nous allons voir comment s'interprète le livre ci-dessus.

ligne 1 : 0 : ligne hors contexte, donc BOOL ne dépendant de rien.

PN : BOOL est donc une notion primitive que l'on interprète comme la notion de Booléen de la logique.

TYPE : indique que c'est une catégorie, donc on pourra définir des objets de type booléen, en mettant BOOL dans la zone catégorie.

C'est ce qui est fait en ligne 2

(2) 0 B EB BOOL

EB : indique que B est une variable booléenne.

La ligne 3 introduit la vérité d'une variable booléenne en notion primitive

(3) B TRUE PN TYPE

TRUE(B) pourra être utilisé en zone catégorie mais dans un sens un peu différent par exemple en ligne 7

(7) Y ASP1 EB TRUE(X)

on lit : hypothèse ASP1 dont l'énoncé est TRUE(X). En ligne 6 a été introduite la notion de conjonction logique AND(X,Y) . En effet le contexte de la ligne 6 est X,Y ; donc AND dépend de X et Y .

Voyons la ligne 9

ASP2 AX1 PN TRUE(AND)

contexte X , Y , ASP1 , ASP2

la forme normale de TRUE(AND) est TRUE(AND(X,Y)) .

Après avoir introduit la notion de vérité d'une variable booléenne (1-3) et la conjonction de deux variables booléennes (4-6). On fait deux hypothèses

(7) X vrai (8) Y vrai

alors dans ce contexte : axiome (9) (X et Y) vrai, ensuite (10) hypothèse (X et Y) vrai,alors dans ce cas axiomes (11) X vrai et (12) Y vrai et toujours dans le même contexte, on démontre le théorème dont l'énoncé est en zone catégorie :(Y et X) vrai et dont la démonstration est en zone définition :

a) AX1(X,Y,ASP1,ASP2) on applique l'axiome : (X et Y) vrai, si X booléen est vrai et Y booléen est vrai, à

b) AX1(Y,X,AX3,AX2) en inversant X et Y , le résultat est bien (Y et X) vrai

c'est la catégorie construite de AX1(Y,X,AX3,AX2) elle est donc bien égale à la catégorie de la ligne 13. La démonstration donnée conduit bien au résultat annoncé.

Notons au passage, qu'il a fallu vérifier la compatibilité des catégories correspondantes dans les paramètres de a) et b).

X et Y ont même catégorie booleen

ASP1 et AX3 : TRUE(X) et TRUE(Y)

mais comme nous l'avons vu :

$$CAT(AX3) = S_{\substack{x\to y \\ y\to x}} CAT(ASP1)$$

donc $$TRUE(Y) = S_{\substack{x\to y \\ y\to x}} TRUE(X) = TRUE(Y) .$$

De même pour ASP2 et AX2 .

Autre remarque : la forme normale de la définition de la ligne 13 est :

$$AX1(Y,X,AX3(X,Y,ASP3) , AX2(X,Y,ASP3))$$

le contexte de cette ligne doit donc contenir X,Y,ASP3 alors qu'à première vue on aurait pu croire que X,Y auraient suffi.

Aspect purement formel du texte P.A.L.

Nous avons dit dans l'introduction qu'un texte P.A.L est indépendant de l'interprétation qu'on en donne. Voyons en une illustration sur l'exemple traité plus haut.

Si à la place de BOOL , TRUE , AND on lit NAT , MULT , PPCM . Nous pouvons alors donner au texte l'interprétation suivante :

(1) soit l'ensemble des entiers naturels

(2) soit B un entier

(3) soit MULT(B) l'ensemble des multiples de B

(4-5) soient x et y des entiers

(6) soit PPCM(x,y) le plus petit commun multiple de x et y

(7-8) soient ASP1 et ASP2 des multiples de x et y

(9) alors par axiome on a ax1(x,y,asp1,asp2) est un multiple de PPCM(x,y)

(10) soit ASP3 un multiple de PPCM(x,y)

(11) alors, axiome : ax2(x,y,asp3) est un multiple de x

(12) alors, axiome : ax3(x,y,asp3) est un multiple de y

(13) dans ce cas th : ax1(y,x,ax3,ax2) est un multiple de PPCM(y,x) et on a bien en zone catégorie MULT(PPCM(y,x)) .

Pour la machine TRUE,BOOL n'a pas plus de sens que NAT et PPCM . Il fait le même travail.

Ce qui est vérifié est donc le schème de la démonstration, indépendamment de l'interprétation.

Réalisation pratique de l'automate vérificateur

1°) Structure modulaire - Présentation des modules.

L'analyse syntaxique et lexicographique (vérification que les mots appartiennent au vocabulaire) sont réalisés par les modules :

ENTRE : gère l'entrée des lignes, les décompose en zones

INPUT : exécute l'entrée d'une chaîne de caractères tapée à la console, vérifie les caractères et par appel à la fonction code transforme cette chaîne de niveau 0 en une chaîne de niveau 1 (voir plus loin)

TRAVAIL : traite les zones ID et IN vérifie l'admissibilité dans le contexte et construit l'arbre

TESTO : traite la zone CAT et construit la table des catégories

TEST1 : traite la zone DEF et construit les tables des constantes, des variables, des expressions.

Pour traiter les expression apparaissant en zones DEF et CAT , TESTO et TEST1 font appel à VALIDE et à IS .

VALIDE : fait l'analyse syntaxique des expressions, appelle ANALYSE.

Les modules ANALYSE , SUBSTITUE , IS , FN , FD vérifient les règles sémantiques du langage. Ce sont les plus fondamentaux.

ANALYSE : vérifie que les catégories des paramètres sont admissibles (voir p. 24) et construit la catégorie de l'expression, pour cela il fait appel à IS et à SUBSTITUE

SUBSTITUE : fait les substitutions

FN : met une expression sous forme normale

FD : met une expression sous forme développée

IS : vérifie l'égalité $\underset{D}{=}$ des catégories.

Les modules utilitaires :

DEBUT : pour initialiser un nouveau livre

REPARTIR : pour reprendre un livre ou le modifier

LIVRE : permet d'éditer le livre qui est actuellement accepté

TABLES : permet de connaître les variables et les constantes introduites.

2°) liaison des différents modules.

Codages et formes internes

niveau 0 : ligne alphanumérique d'entrée

0 X EB TYPE

niveau 1 : ligne codée, même structure, mais chaque caractère est représenté par un nombre

37 1 25 1 63 1 21 26 176

niveau 2 : pour les identificateurs, conversion des cinq premiers caractères modulo 47 (p. 37)

niveau 3 : pour ARB les éléments de la liste sont l'indice dans la table des variables (p. 38)

le plus fondamental : partout on remplace un identificateur par le n° de ligne où il a été défini.

niveau 4 : forme finale après traitement des expressions.

les séparateurs sont représentés par 1000000-CODE (séparateur) et pour séparer les expressions dans les listes 1000000 + N° de ligne (voir annexe p. 2)

niveau 5 : forme transitoire d'une expression (sortie de valide vers analyse qui transforme celle-ci en forme finale)

B X1 X2 X3 -1 Σ1 -2 Σ2 -3 Σ3 Σi expression

INPUT : convertit une chaîne de niveau 0 en chaîne de niveau 1

ONCODE : niveau 1 à niveau 2

DECODE : du niveau 2 à 0 (utilisé dans résultat).

Les tables des variables et des constantes sont chacune formée de deux listes :

T : variables codées de niveau 2

NT : numéros de ligne de définition correspondant

L : constantes codées de niveau 2

NL : numéros de ligne de définition correspondant.

Toutes les tables TCAT , TCACO , TEXP sont des listes de niveau 4.

VALIDE : niveau 5 transmis à ANALYSE qui effectue la conversion niveau 4 des expressions qui seront rangées dans les tables précédentes.

IS , FN , FD acceptent des expressions du niveau 4 et fournissent des expressions de même niveau.

Nous allons préciser les transmissions de résultats entre modules manipulant les expressions.

A partir de TESTO une expression qui est en zone catégorie est passée à VALIDE

TESTO → VALIDE

CAT (niveau 1)

de même TEST1 → VALIDE

DEF (niveau 1)

Valide transforme l'expression de niveau 1 en niveau 5 et donne sa catégorie de même niveau.

VALIDE → ANALYSE

$$\left.\begin{matrix} \text{EXP} \\ \text{CATE} \end{matrix}\right\} \text{ niveau 5}$$

ANALYSE transforme EXP , CATE en expressions de niveau 4 et les renvoient vers TESTO ou TEST1 ainsi que le résultat logique VAL = 0 ou 1 .

FN et FD niveau 4 → niveau 4 d'une expression.

IS compare deux expressions de niveau 4 et renvoie un résultat logique VAL = 0 ou 1.

Il reste substitue : I SUBSTITUE C

$$\text{CACO} = S\left\{\begin{matrix} G[1] \rightarrow ZD[1] \\ G[I-1] \rightarrow ZD[I-1] \end{matrix}\right. \quad C$$

le résultat de niveau 4 est transmis par CACO

G est la liste des éléments à remplacer

ZD est la liste des remplaçants niveau 5

C c'est l'expression de niveau 4 sur laquelle s'effectue la substitution

CACO , G , ZD sont implicites dans l'appel de SUBSTITUE.

3°) Utilisation.

Les organigrammes et listings sont placés en annexe. Mais pour utiliser le programme il n'est pas strictement nécessaire de savoir comment il est réalisé.

L'utilisateur mathématicien doit seulement connaître P.A.L.

Le langage de commande se réduit à

DEBUT pour commencer un livre

FIN pour sortir de l'automate (le travail réalisé est conservé)

REPARTIR N pour reprendre un travail déjà commencé ou modifier un livre N est la ligne où l'on veut repartir.

Il y a aussi les utilitaires LIVRE et TABLES.

Mode d'emploi :

1. Mettre la machine en état de faire de l'APL. Charger la zone de travail contenant ce programme.

2. Taper LIVRE pour voir ce que contient la mémoire.

Si l'on veut continuer le livre actuel on tape REPARTIR N ou DEBUT pour en commencer un autre.

3. Ensuite la machine indique la ligne en cours et la possibilité d'arrêter la session. Toutefois si l'on s'est trompé dans une zone que la machine a acceptée et si la zone DEF n'est pas acceptable on tape EB ou PN si c'est la zone CAT on tape TYPE , ainsi la ligne est acceptée, mais ce n'est pas celle que l'on voulait Alors à la ligne suivante on tape FIN et on repart avec REPARTIR à la ligne choisie.

Normalement, le programme ne s'interrompt pas pour cause d'erreur mais il exhibe la ligne litigieuse avec un message indiquant la raison de rejet. On peut alors

rectifier.

4°) Conclusion.

Le but recherché est atteint, on dispose d'un automate simple qui nous aide à vérifier des démonstrations mathématiques écrites en P.A.L. Cependant, l'amélioration de la représentation interne des expressions-sous forme d'arbre plutôt que linéaire pour éviter de refaire l'analyse syntaxique-permettra de rendre la vérification plus rapide. A titre d'indication, sur IBM 75, la vérification du livre en annexe qui démontre $(A \supset A$, vrai) après avoir introduit la logique nécessaire, demande entre 20 et 30 secondes pour un texte de 24 lignes.

Des structures plus adaptées seront disponibles avec APL X et des essais sont en cours en Snobol-Spitbol.

Après avoir optimisé cet algorithme en temps, il faudra l'étendre à la vérification de textes en AUTOMATH. Différentes versions d'AUTOMATH ont été implantées en Algol par l'équipe de N.G. De Bruijn. En A.P.L on introduit l'aspect conversationnel qui en fait un outil facilement utilisable par le mathématicien.

BIBLIOGRAPHIE

(1) NG De Bruijn

The mathematical language AUTOMATH, its usage, and some of its extensions.

Proceeding of the Symposium on Automatic

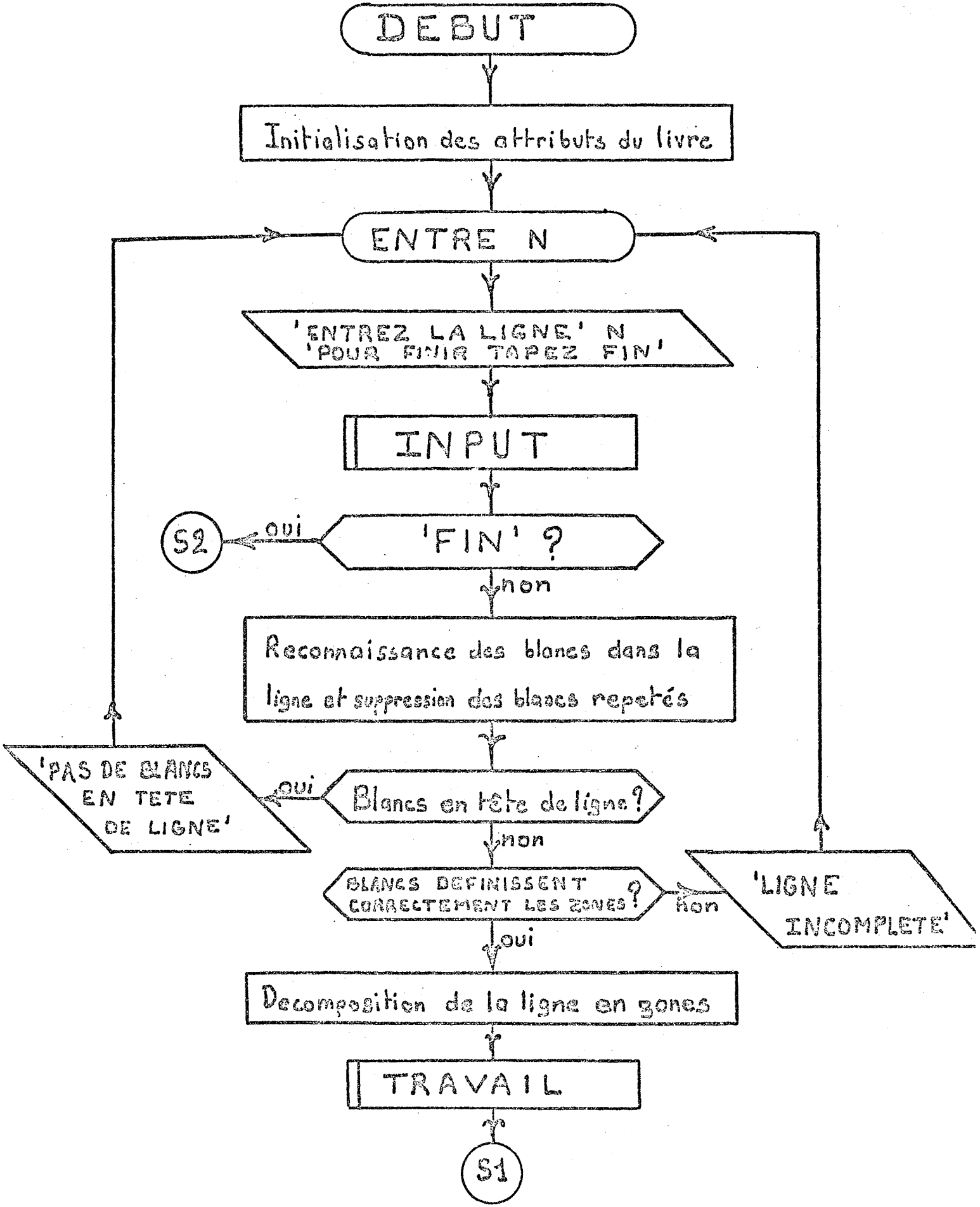
démonstration (IRIA, Versailles, Décembre 1968)

Springer lectures notes series (1969).

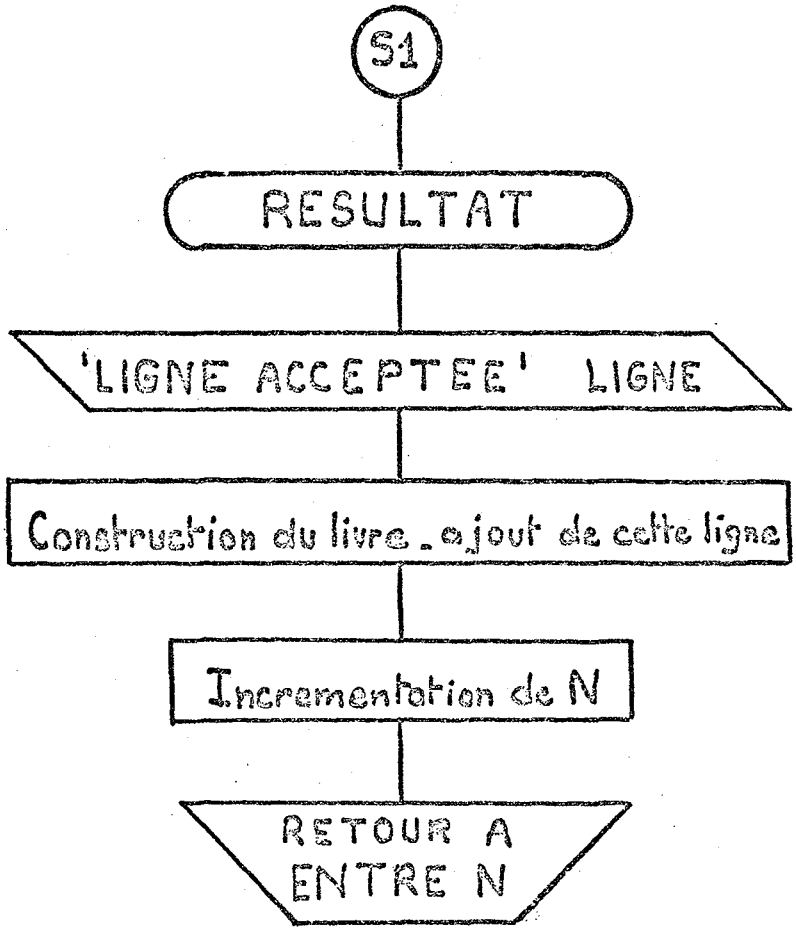
(2) Sandra Pakin APL/360

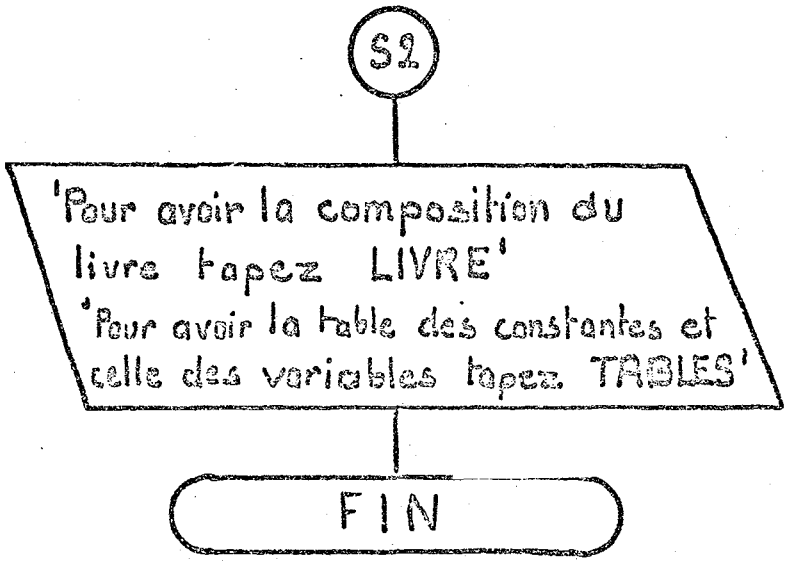
référence manuel

Bernard Robinet le langage A.P.L.

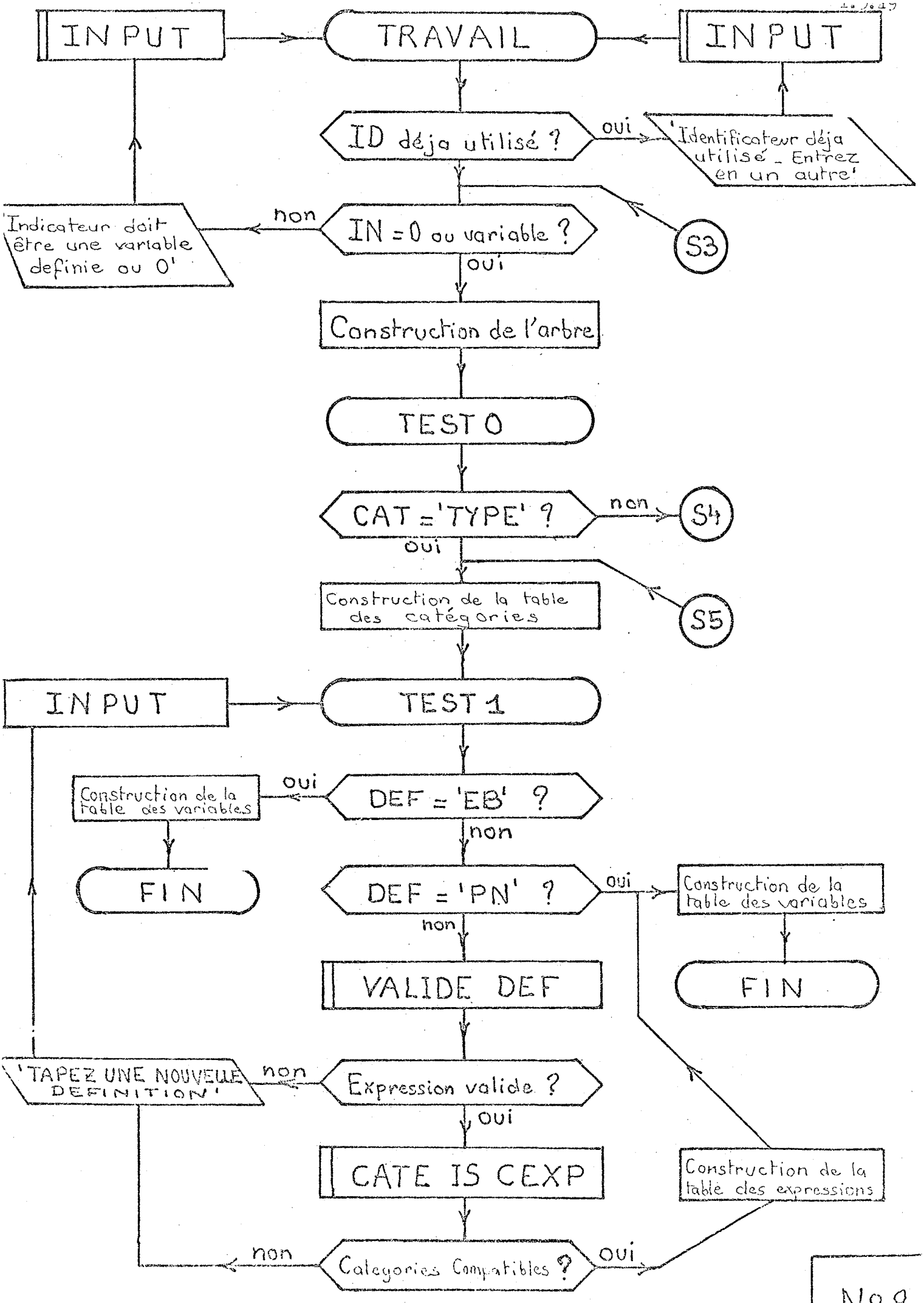


Organigramme n° 1

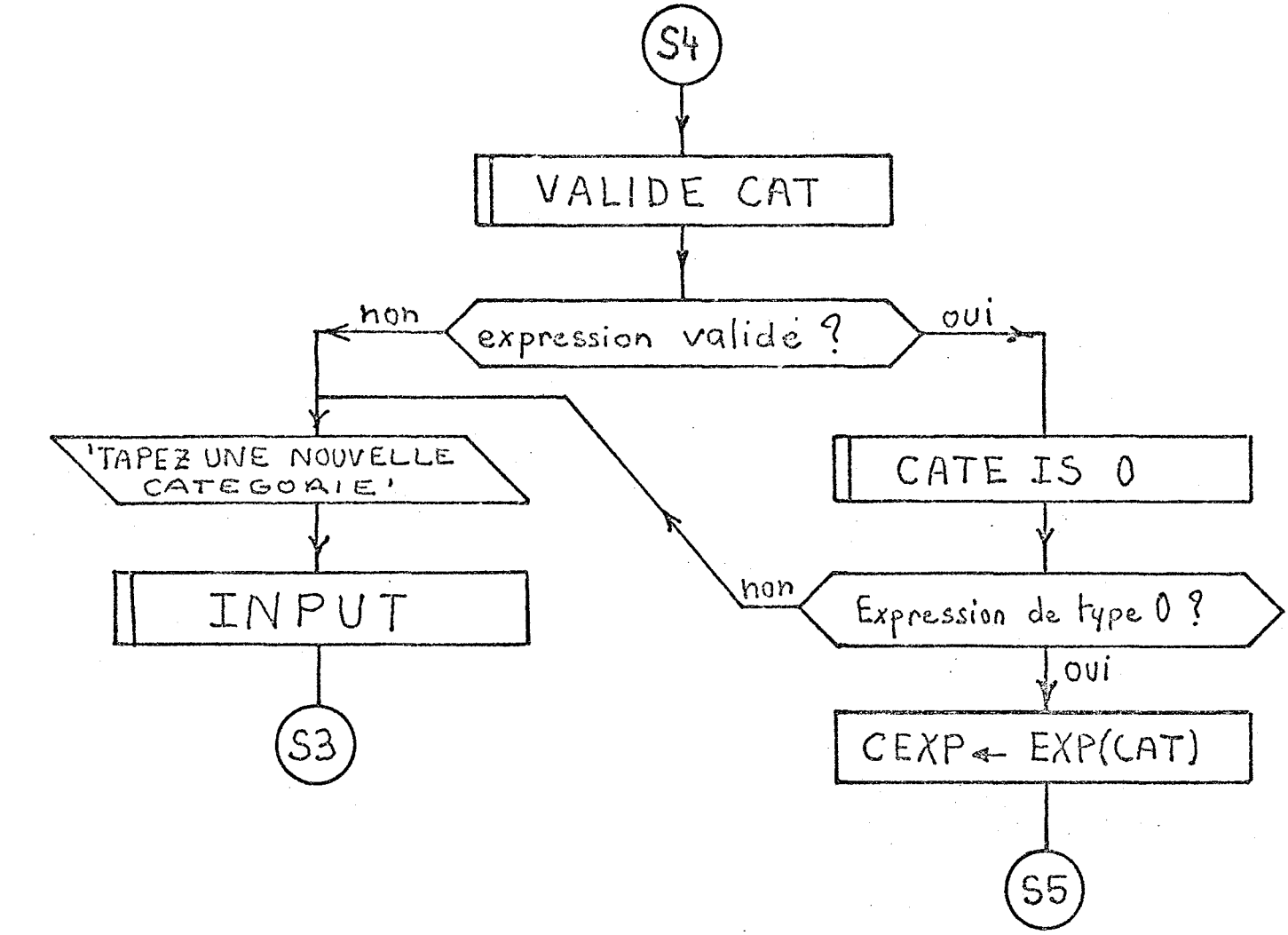


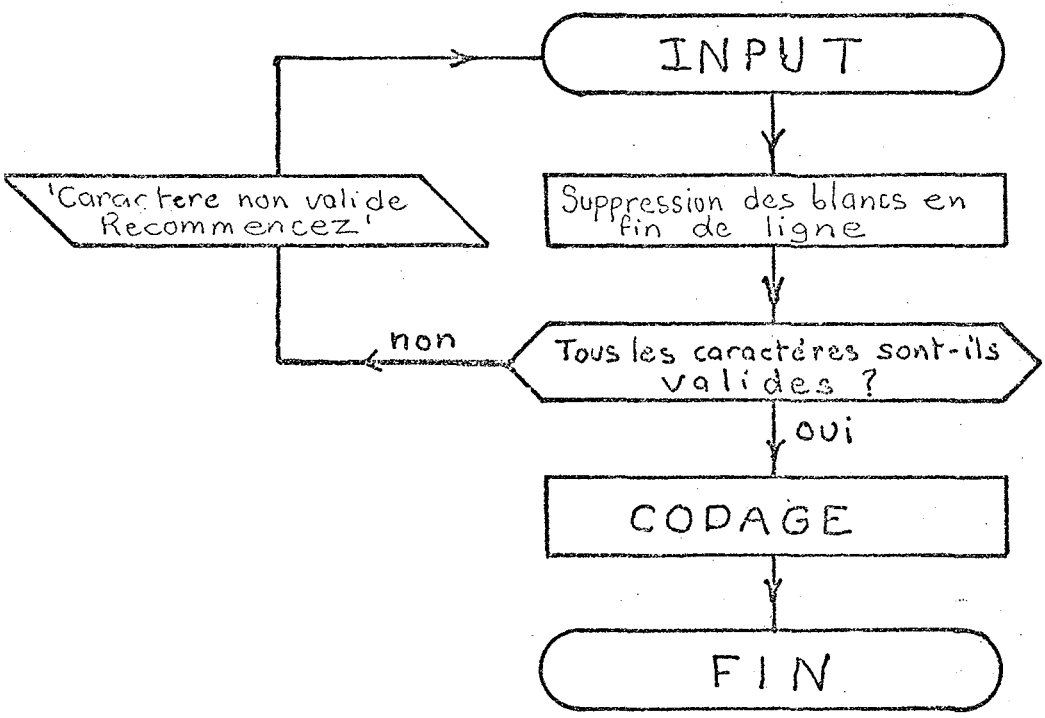


Suite organigramme n° 1

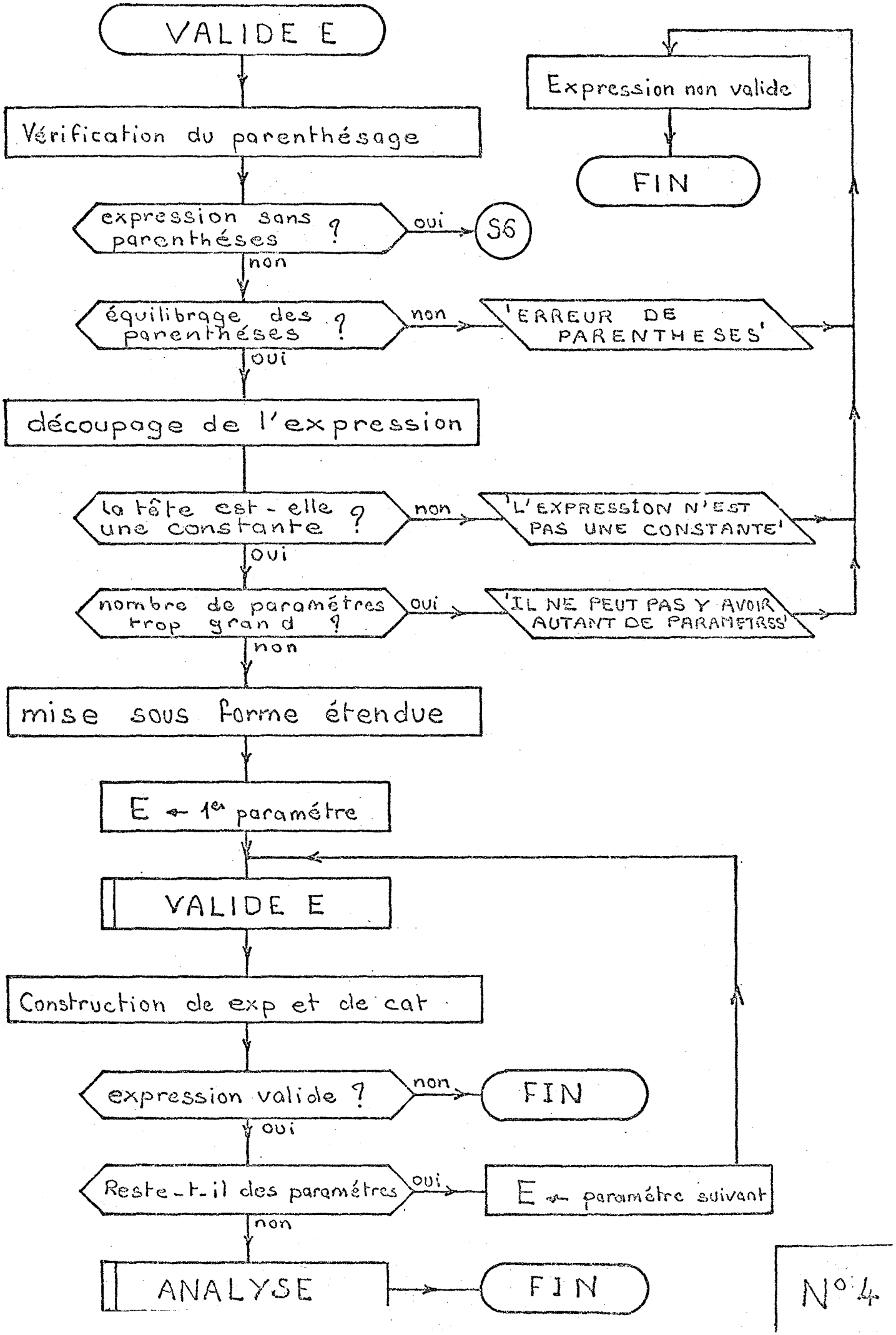

INPUT
TRAVAIL
INPUT
ID déja utilisé ?
oui
'Identificateur déja utilisé _ Entrez en un autre'
Indicateur doit être une variable definie ou 0'
non
IN = 0 ou variable ?
S3
oui
Construction de l'arbre
TEST 0
CAT = 'TYPE' ?
non
S4
oui
Construction de la table des catégories
S5
INPUT
TEST 1
Construction de la table des variables
oui
DEF = 'EB' ?
non
FIN
DEF = 'PN' ?
oui
Construction de la table des variables
non
VALIDE DEF
FIN
'TAPEZ UNE NOUVELLE DEFINITION'
non
Expression valide ?
oui
CATE IS CEXP
Construction de la table des expressions
non
Categories Compatibles ?
oui

N° 2.



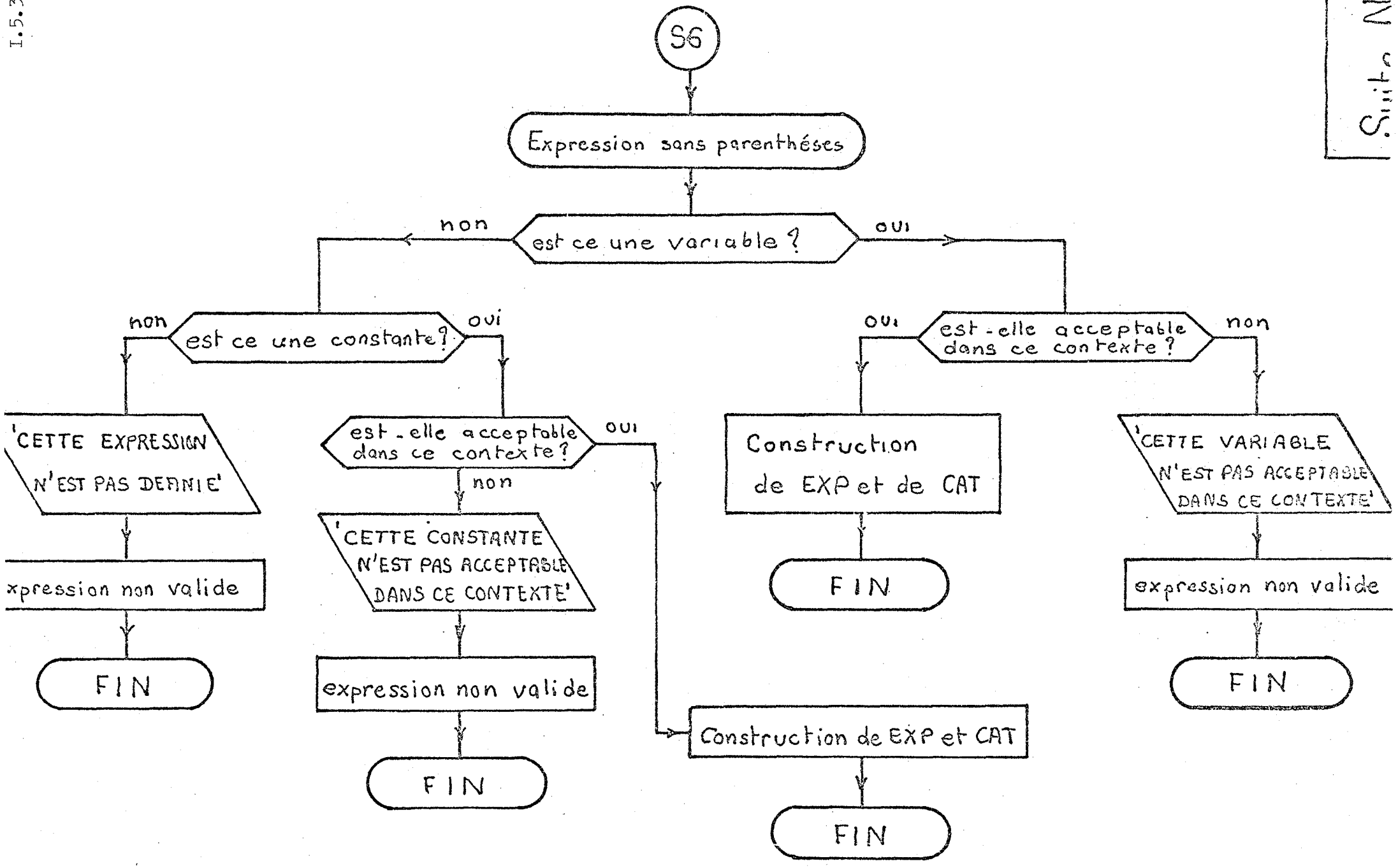


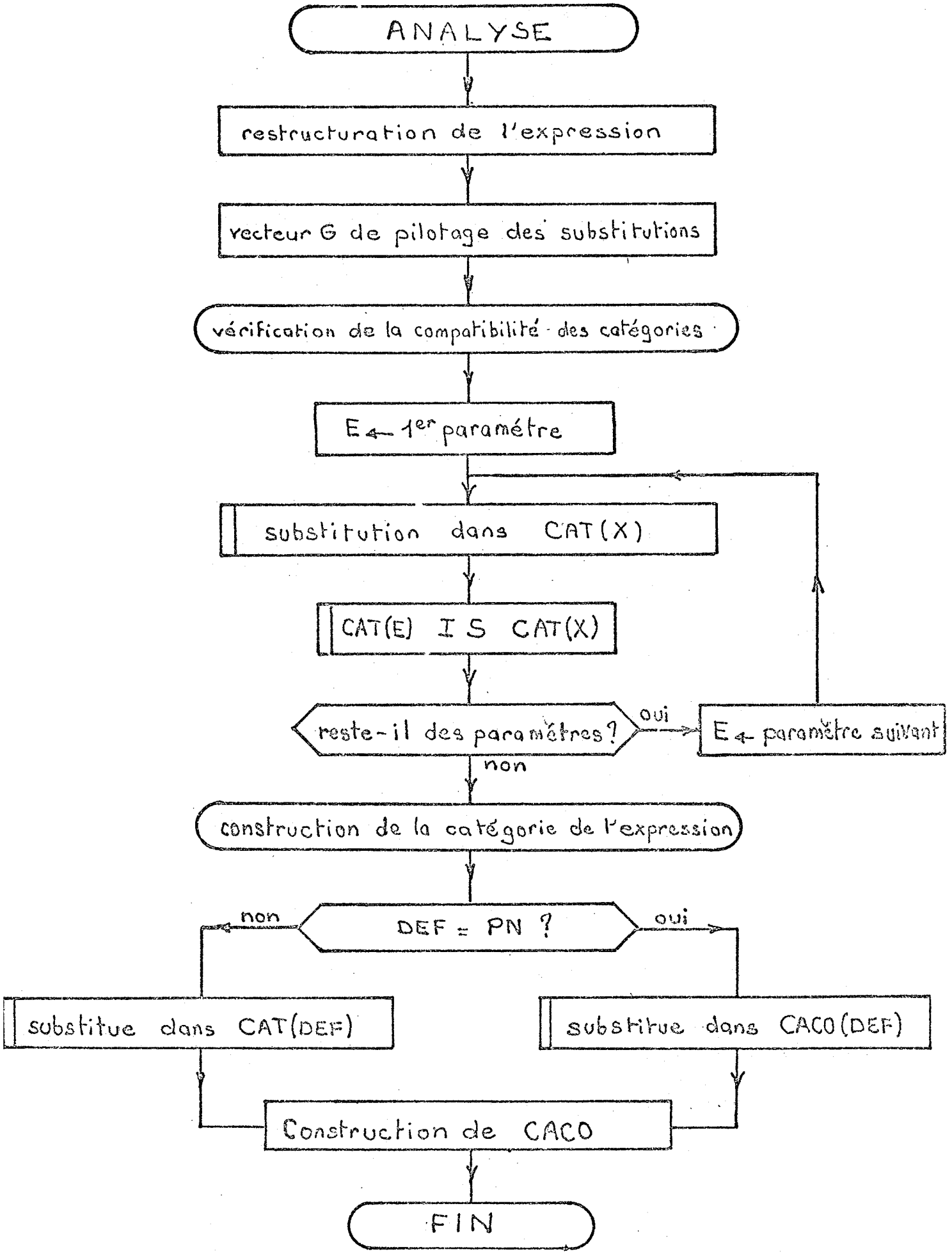
Suite no 9. et no 3


VALIDE E
Vérification du parenthésage
expression sans parenthéses ?
oui
S6
non
équilibrage des parenthéses ?
non
'ERREUR DE PARENTHESES'
oui
découpage de l'expression
la tête est-elle une constante ?
non
'L'EXPRESSION N'EST PAS UNE CONSTANTE'
oui
nombre de paramétres trop grand ?
oui
'IL NE PEUT PAS Y AVOIR AUTANT DE PARAMETRES'
non
mise sous forme étendue
E ← 1er paramétre
VALIDE E
Construction de exp et de cat
expression valide ?
non
FIN
oui
Reste-t-il des paramétres
oui
E ← paramétre suivant
non
ANALYSE
FIN
Expression non valide
FIN

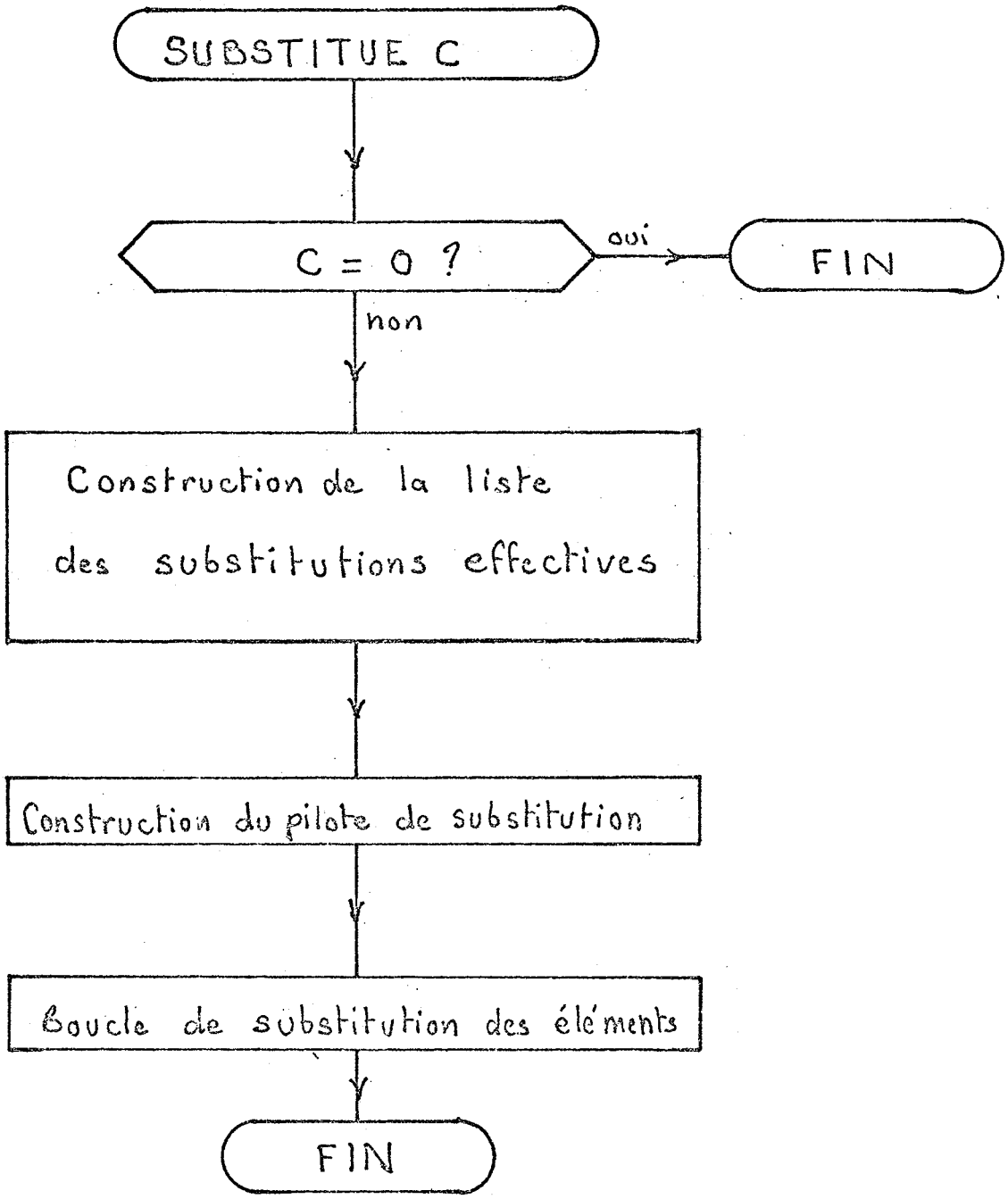
N°4

Suite N°4


S6
Expression sans parenthéses
est ce une variable ?
non
oui
est ce une constante?
non
oui
est - elle acceptable dans ce contexte ?
oui
non
'CETTE EXPRESSION N'EST PAS DEFINIE'
xpression non valide
FIN
est - elle acceptable dans ce contexte?
oui
non
'CETTE CONSTANTE N'EST PAS ACCEPTABLE DANS CE CONTEXTE'
expression non valide
FIN
Construction de EXP et de CAT
FIN
'CETTE VARIABLE N'EST PAS ACCEPTABLE DANS CE CONTEXTE'
expression non valide
FIN
Construction de EXP et CAT
FIN


ANALYSE
restructuration de l'expression
vecteur G de pilotage des substitutions
vérification de la compatibilité des catégories
E ← 1er paramétre
substitution dans CAT(X)
CAT(E) I S CAT(X)
reste-il des paramétres?
oui
E ← paramétre suivant
non
construction de la catégorie de l'expression
non
DEF = PN ?
oui
substitue dans CAT(DEF)
substitue dans CACO(DEF)
Construction de CACO
FIN

N°5


SUBSTITUE C
C = 0 ?
oui
FIN
non
Construction de la liste des substitutions effectives
Construction du pilote de substitution
Boucle de substitution des éléments
FIN

N°6

Commentaire :

I substitue C réalise

$$\frac{S_{x_1 \to \Sigma_1} \quad C}{x_{i-1} \to \Sigma_{i-1}}$$

et utilise $G = x_1, \ldots, x_n$

$ZD = \Sigma_1, \ldots, \Sigma_n$.

Pour les substitutions effectives, on compare x_i et Σ_i s'ils sont égaux, on les supprime de la liste des substitutions.

Ensuite on teste si la liste des substitutions effectives est vide si oui c'est fini sinon on construit le pilote de substitution.

M matrice pilote

expression	C	→	B	(	X	,	Y	)	PIL	ZD
liste TG des éléments à remplacer	X	0	0	1	0	0	0		1	G(Y)
	Y	0	0	0	0	1	0		1	Z
	Z	0	0	0	0	0	0		0	H(X)

on regarde si un élément n'intervient pas dans l'expression. PIL est un vecteur colonne, le i' élément est à 0 si la ligne correspondante est à 0 (c'est-à-dire si l'élément en question n'apparaît pas dans l'expression). A l'aide de PIL on simplifie parallèlement M et TG , ZD

							TG	ZD
M	0	0	1	0	0	0	X	G(Y)
	0	0	0	0	1	0	Y	Z

on teste si TG est vide si aucun élément à substituer n'apparaît dans l'expression

Si oui c'est fini.

Sinon la matrice M pilote la substitution des éléments. Pour le 1er élément on prend la 1ère ligne qui indique la position de celui-ci dans l'expression

B(X , Y)

0 0 1 0 0 0 — on le remplace par G(Y)

B(G(Y) , Y) — on effectue une opération

0 0 0 0 0 0 0 0 0 — similaire sur M

0 0 0 0 0 0 0 1 0

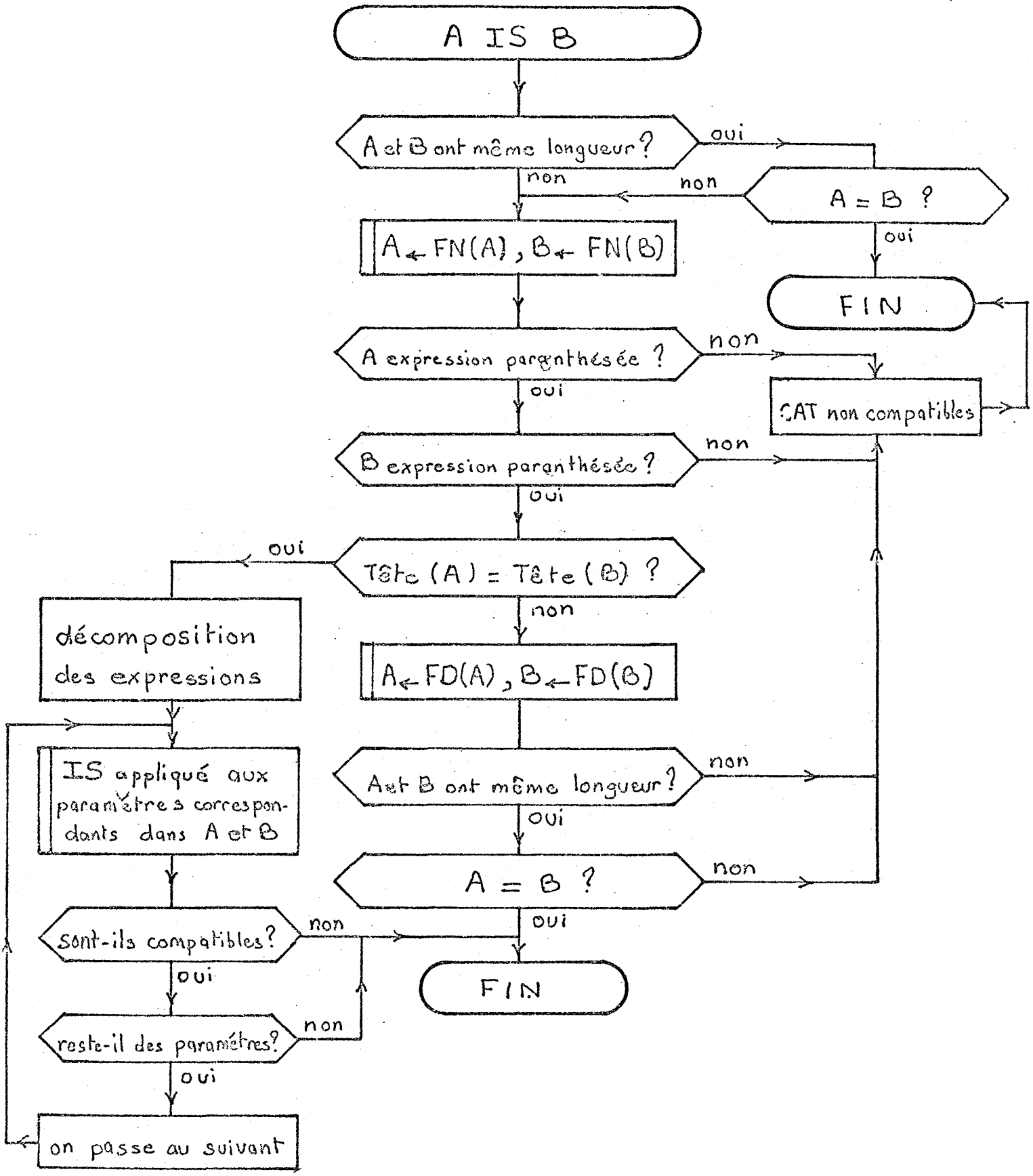
On passe au suivant de la même façon

B(G(Y) , Y)

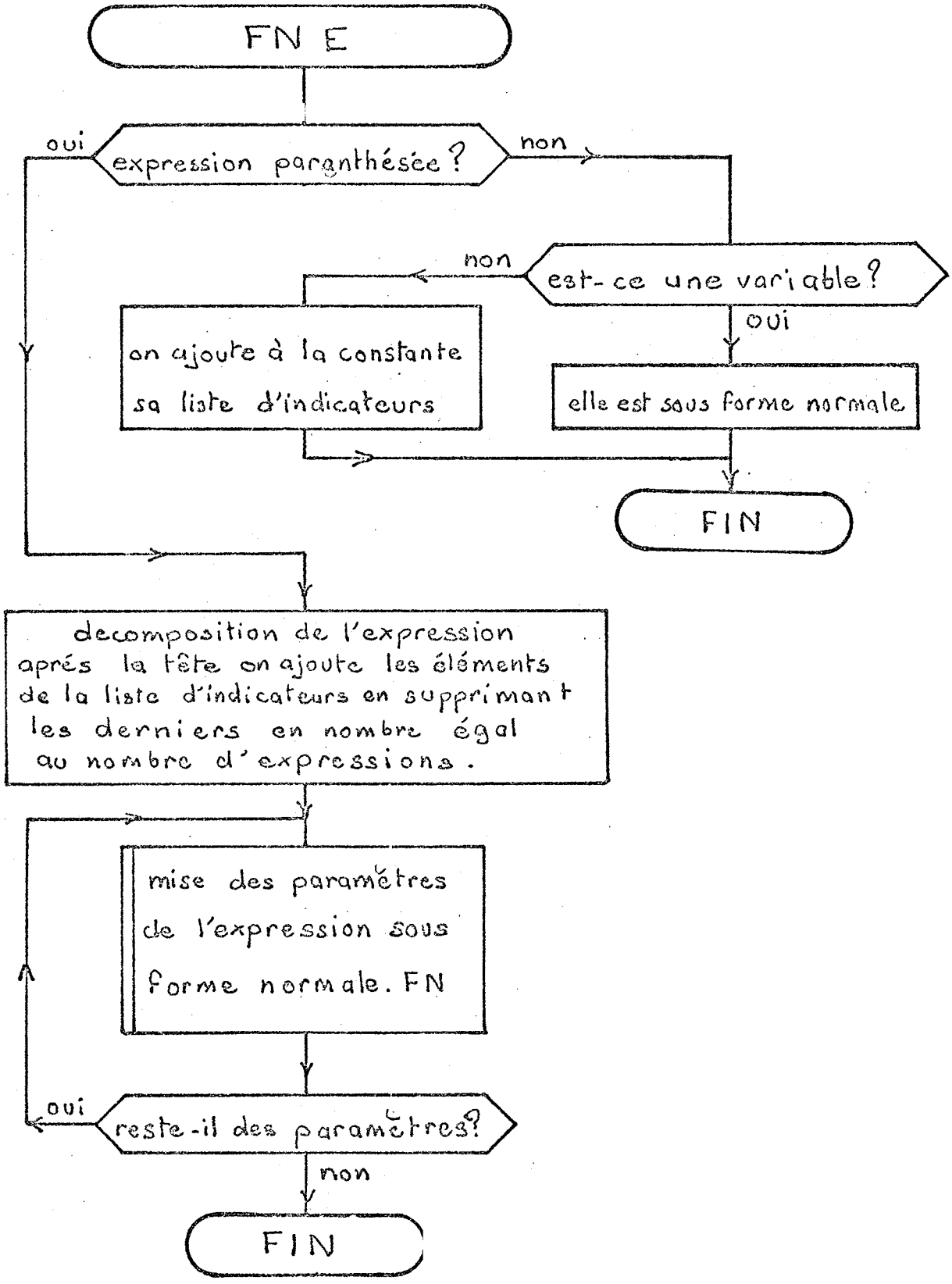
0 0 0 0 0 0 0 1 0 — Y → Z

B(G(Y) , Z)

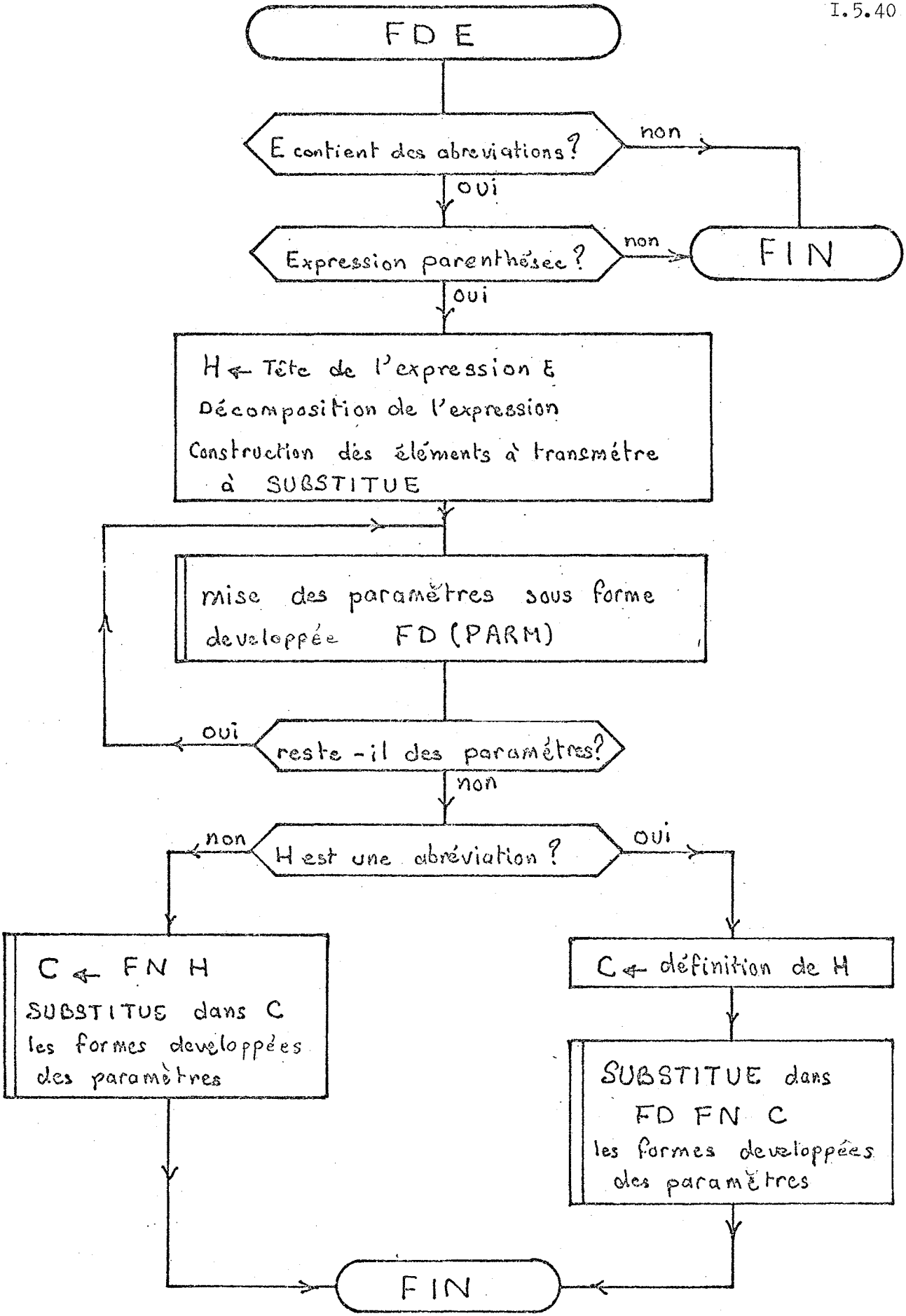
Le cas où un élément apparaît en plusieurs endroits dans l'expression est traité de la même façon. Le résultat de la substitution est placé dans CACO.


A IS B
A et B ont même longueur ?
oui
non
non
A = B ?
oui
A ← FN(A), B ← FN(B)
FIN
A expression parenthésée ?
non
oui
CAT non compatibles
B expression parenthésée ?
non
oui
oui
Tête (A) = Tête (B) ?
décomposition des expressions
non
A ← FD(A), B ← FD(B)
IS appliqué aux paramètres correspondants dans A et B
A et B ont même longueur ?
non
oui
A = B ?
non
oui
sont-ils compatibles ?
non
oui
FIN
reste-il des paramètres ?
non
oui
on passe au suivant

N° 7


FN E
oui
expression paranthésée ?
non
non
est-ce une variable ?
oui
on ajoute à la constante sa liste d'indicateurs
elle est sous forme normale
FIN
decomposition de l'expression aprés la tête on ajoute les éléments de la liste d'indicateurs en supprimant les derniers en nombre égal au nombre d'expressions.
mise des paramétres de l'expression sous forme normale. FN
oui
reste-il des paramètres?
non
FIN

N°8


FDE
E contient des abreviations?
non
oui
Expression parenthésee?
non
FIN
oui
H ← Tête de l'expression E
Décomposition de l'expression
Construction des éléments à transmétre à SUBSTITUE
mise des paramètres sous forme developpée FD (PARM)
oui
reste-il des paramétres?
non
non
H est une abréviation ?
oui
C ← FN H
SUBSTITUE dans C les formes developpées des paramètres
C ← définition de H
SUBSTITUE dans FD FN C les formes developpées des paramètres
FIN

N° 9

ANNEXE

Exemple 1

vue utilisation de P.A.L

LIVRE

```
1     0   BOOL  PH  TYPE
2     0   B   EB  BOOL
3     B   TRUE  PH  TYPE
4     0   X   EB  BOOL
5     X   Y   EB  BOOL
6     Y   AND   PH  BOOL
7     Y   ASP1  EB  TRUE(X)
8     ASP1  ASP2  EB  TRUE(Y)
9     ASP2  AX1   PH  TRUE(AND)
10    Y   ASP3  EB  TRUE(AND)
11    ASP3  AX2   PH  TRUE(X)
12    ASP3  AX3   PH  TRUE(Y)
13    ASP3  TH  AX1(Y,X,AX3,AX2)  TRUE(AND(Y,X))
```

TABLES

TABLES DES VARIABLES

2	B
4	X
5	Y
7	ASP1
8	ASP2
10	ASP3

TABLE DES CONSTANTES

1	BOOL
3	TRUE
6	AND
9	AX1
11	AX2
12	AX3
13	TH

vérification en moins de 3 s (CPU 360/75)

TABLES CONSTRUITES

ARB

0 0 1 0 2 3 3 4 5 3 6 6 6

TCAT

1000001 0 1000002 1 1000003 0 1000004 1 1000005 1 1000006 1 1000007 3 999961 4 999960 1000008 3 999961 5 999960
1000009 3 999961 6 999960 1000010 3 999961 6 999960 1000011 3 999961 4 999960 1000012 3 999961 5 999960
1000013 3 999961 6 999961 5 999962 4 999960 999960 1000014

TEXP

1000001 1000002 1000003 1000004 1000005 1000006 1000007 1000008 1000009 1000010 1000011 1000012 1000013 9 999961
5 999962 4 999962 12 999962 11 999960 1000014

TCACO

1000001 1000002 1000003 1000004 1000005 1000006 1000007 1000008 1000009 1000010 1000011 1000012 1000013 3 999961
6 999961 5 999962 4 999960 999960 1000014

Exemple 2

TABLES

TABLES DES VARIABLES

2	*P*
4	*Q*
8	*M*
11	*ASP1*
12	*ASP2*
14	*A*
15	*B*

TABLE DES CONSTANTES

1	*EBF*
3	*VRAI*
5	*INCLU*
6	*S1*
7	*AX1*
9	*S2*
10	*AX2*
13	*MP*
16	*VS2*
17	*VS4*
18	*VS1*
19	*VS3*
20	*H1*
21	*H2*
22	*H3*
23	*H4*
24	*TH*

LIVRE

1	0	*EBF*	*PN*	*TYPE*
2	0	*P*	*EB*	*EBF*
3	*P*	*VRAI*	*PN*	*TYPE*
4	*P*	*Q*	*EB*	*EBF*
5	*Q*	*INCLU*	*PN*	*EBF*
6	*Q*	*S1*	*INCLU(INCLU(Q,P))*	*EBF*
7	*Q*	*AX1*	*PN*	*VRAI(S1)*
8	*Q*	*M*	*EB*	*EBF*
9	*M*	*S2*	*INCLU(INCLU(INCLU(Q,M)),INCLU(INCLU,INCLU(M)))*	*EBF*
10	*M*	*AX2*	*PN*	*VRAI(S2)*
11	*Q*	*ASP1*	*EB*	*VRAI*
12	*ASP1*	*ASP2*	*EB*	*VRAI(INCLU)*
13	*ASP2*	*MP*	*PN*	*VRAI(Q)*
14	0	*A*	*EB*	*EBF*
15	*A*	*B*	*EB*	*EBF*
16	*B*	*VS2*	*INCLU(A,INCLU(B,A))*	*EBF*
17	*B*	*VS4*	*INCLU(A,A)*	*EBF*
18	*B*	*VS1*	*VS2(A,INCLU(B,A))*	*EBF*
19	*B*	*VS3*	*INCLU(VS2,VS4)*	*EBF*
20	*B*	*H1*	*AX1(A,INCLU(B,A))*	*VRAI(VS1)*
21	*B*	*H2*	*AX1(A,B)*	*VRAI(VS2)*
22	*B*	*H3*	*AX2(A,INCLU(B,A),A)*	*VRAI(INCLU(VS1,VS3))*
23	*B*	*H4*	*MP(VS1,VS3,H1,H3)*	*VRAI(INCLU(VS2,VS4))*
24	*B*	*TH*	*MP(VS2,VS4,H2,H4)*	*VRAI(VS4)*

EBF : expression bien formée

axiomes

AX1 : $\vdash P \supset (Q \supset P)$

AX2 : $\vdash (P \supset (Q \supset M)) \supset ((P \supset Q) \supset (P \supset M))$

Règle du modus ponens

MP : $\vdash P$ et $\vdash P \supset Q$ alors $\vdash Q$

De 1 à 13 introduction de la logique nécessaire

14 à 24 présentation de la démonstration du théorème $\vdash (A \supset A)$.

En 4 étapes

H1 : $\vdash A \supset ((B \supset A) \supset A)$

dans AX1 on met A à la place de P et $B \supset A$ à la place de Q

H2 : $\vdash A \supset (B \supset A)$

dans AX1 on met A à la place de P et B à la place de Q

H3 : $\vdash (A \supset ((B \supset A) \supset A)) \supset ((A \supset (B \supset A)) \supset (A \supset A))$

dans AX2 on met A à la place de P

$B \supset A$ à la place de Q

A à la place de M

H4 : $\vdash (A \supset (B \supset A)) \supset (A \supset A)$

d'après MP appliqué à M_1 et M_3

TH : $\vdash A \supset A$

d'après MP appliqué à H_2 et H_4

```
      ∇TRAVAIL[□]∇
    ∇ Z←TRAVAIL;M
[1]  ⍝VERIFICATION DE L'IDENTIFICATEUR
[2]  B1:M←(T,L)⍳ID←ENCODE ID
[3]   →(M>⍴(T,L))/B2
[4]   'IDENTIFICATEUR DEJA UTILISE EN LIGNE ';(NT,NL)[M]
[5]   'ENTREZ EN UN AUTRE'
[6]   ID←INPUT
[7]   →B1
[8]  ⍝VERIFICATION DE L'INDICATEUR
[9]  B2:M←T⍳IN←ENCODE IN
[10]  →(M≤(⍴T))/S1
[11]  M←0
[12]  →(IN=180548197)/S1
[13]  'INDICATEUR DOIT ETRE UNE VARIABLE DEF OU 0'
[14]  'ENTREZ EN UN AUTRE'
[15]  IN←INPUT
[16]  →B2
[17] ⍝CONSTRUCTION DE L'ARBRE'
[18] S1:ARB←ARB,M
[19]  TEST0
[20]  TEST1
    ∇
```

```
[28]  J←J+1
[29]  →(J≤(+/M[K;]))/RT
[30]  PIL←PIL,POS[J]↓~M[K;]
[31]  →((ρM)[2]≠ρPIL)/S3
[32]  UN←((ρM)[2])ρ1
[33]  UN[1↓POS]←0
[34]  M←UN/M
[35]  M←PIL\M
[36] S3:CACO←CA,POS[J]↓CACO
[37]  K←K+1
[38]  →(K≤ρDEV)/B1
    ∇
      ∇TESTO[
            ∨
            0[□]∇
    ∇ Z←TESTO;CATE;EXP;VAL
[1]  ⍝VERIFICATION DE LA CATEGORIE
[2]   CEXP←0
[3]  B1:→(105210534=ENCODE CAT)/S3
[4]   INIT
[5]   VALIDE CAT
[6]   →VAL/S2
[7]   CATE IS 0
[8]   →VAL/S2
[9]   CEXP←EXP
[10]  →S3
[11] S2:□←ALPHA[CAT]
[12]  'TAPEZ UNE NOUVELLE CATEGORIE'
[13]  CAT←INPUT
[14]  →B1
[15] ⍝CONSTRUCTION DE LA TABLE DES CATFGORIES
[16] S3:TCAT←TCAT,CEXP
    ∇
```

```
      ∇TEST1[□]∇
    ∇ Z←TEST1;CATE;EXP;VAL
[1]   ⍝VERIFICATION DE LA DEFINITION
[2]   B1: →(29589555 84511922 ∈ENCODE DEF)/S1,S2
[3]    INIT
[4]    VALIDE DEF
[5]    →VAL/S3
[6]    CATE IS CEXP
[7]    →VAL/S3
[8]   ⍝CONSTRUCTION DE LA TABLE DES EXPRESSIONS
[9]    TEXP←TEXP,EXP
[10]   TCACO←TCACO,CATE
[11]  ⍝CONSTRUCTION DE LA TABLE DES CONSTANTES
[12]   LDEF←LDEF,N
[13]  S2:L←L,ID
[14]   NL←NL,N
[15]   →0
[16]  ⍝CONSTRUCTION DE LA TABLE DES VARIABLES
[17]  S1:T←T,ID
[18]   NT←NT,N
[19]   →0
[20]  S3: ⍞←ALPHA[DEF]
[21]   'TAPEZ UNE NOUVELLE DEFINITION'
[22]   DEF←INPUT
[23]   →B1
    ∇
```

```
      ∇SÜBSTITUE[□]∇
    ∇ I SUBSTITUE C;J;DEV;TG;M;PIL;K;CA
[1]   CACO←C
[2]  ⍝SI LA CATEGORIE EST TYPE ON NE PEUT RIEN SUBSTITUER
[3]   →(C=0)/0
[4]  ⍝CONSTRUCTION DE LA LISTE DES SUBSTITUTIONS EFFECTIVE
      S
[5]   TG←DEV←⍳0
[6]   J←1
[7]  RET:→(J≥I)/S2
[8]  ⍝CE N'EST PAS LA PEINE DE SUBSTITUER UN ELEMENT ALUI
      MEME
[9]   →(∧/G[J]=ZD PREND J)/S1
[10]  TG←TG,G[J]
[11]  DEV←DEV,J
[12] S1:J←J+1
[13]  →RET
[14] ⍝PILOTAGE DES SUBSTITUTIONS
[15] S2:→(0=⍴DEV)/0
[16]  M←TG∘.=CACO←FN C
[17]  PIL←∨/M
[18]  DEV←PIL/DEV
[19]  M←PIL⌿M
[20]  →(0=⍴DEV)/0
[21] ⍝SUBSTITUTION
[22]  K←1
[23] B1:CA←PIL←⍳0
[24]  J←1
[25]  POS←0,((+/M[K;])↑⍳M[K;])
[26] RT:CA←CA,(POS[J]↓(POS-1)[J+1]↑CACO),TG←ZD PREND DEV[K
      ]
[27]  PIL←PIL,(POS[J]↓(POS-1)[J+1]↑~M[K;]),(⍴TG)⍴0
```

```
      ∇IS[□]∇
    ∇ Z←A IS B
[1]   →((ρA)≠ρB)/S1
[2]   →(∧/A=B)/0
[3]  S1:A←FD FN A
[4]   B←FD FN B
[5]   →((ρA)≠ρB)/S2
[6]   →(∧/A=B)/0
[7]  S2:'VOIR LES CAT';A;' ET  ';B
[8]   VAL←1
    ∇
      ∇LIVRE[□]∇
    ∇ LIVRE;I;B;POS;BOK
[1]   I←1
[2]   POS←(+/B)↑∇B←'-'∘.=BOK←1↓BOOK
[3]   POS←0,POS
[4]   B←'-'∘.=BOOK
[5]  RET:''
[6]   5 0 ▼I;'      ';POS[I]↓(POS-1)[I+1]↑BOK
[7]   I←I+1
[8]   →(I<+/B)/RET
    ∇
      ∇LI[□]∇
    ∇ Z←LI N;R
[1]   Z←0ρ1
[2]   →(0 ρN)/0
[3]   →(0=R←ARB[N])/0
[4]   Z←NT[R],LI NT[R]
    ∇
```

```
      ∇FN[⎕]∇
    ∇ C←FN E;H;SE;POS;V;A;J                                    I.5.50
[1]   C←⍳0
[2]   POS←+\(E∈999961)-E∈999960
[3]   →(0=+/POS)/SC
[4]   H←1↑E
[5]   SE←2↓¯1↓E
[6]   POS←2↓¯1↓POS
[7]   V←+/POS←(POS-SE∈999962)∈0
[8]   C←C,2↑E
[9]   A←(⍴LI H)-V+1
[10]  →(0=A)/S2
[11]  C←C,(((¯1+2×A)⍴0,999962)+((¯1+2×A)⍴1,0)\A↑⌽LI H),
      999962
[12] S2:POS←0,(V↑⍒POS),1+⍴SE
[13]  J←1
[14] B1:E←POS[J]↓(POS-1)[J+1]↑SE
[15]  C←C,(FN E),(J≠V+1)⍴999962
[16]  J←J+1
[17]  →(J≤V+1)/B1
[18]  C←C,999960
[19]  →0
[20] SC:→(E∈NT)/S1
[21]  A←2×⍴LI E
[22]  →(0=A)/S1
[23]  C←C,E,999961,(((A-1)⍴0,999962)+((A-1)⍴1,0)⌽LI E),
      999960
[24]  →0
[25] S1:C←C,E
```

```
      ∇FD[[]]∇
    ∇ Z←FD E;C;ZD;POS;SE;V;J;H;G;CACO
[1]   Z←E
[2]   →(∧/~(LDEF∈E))/0
[3]   ZD←⍳0
[4]   Z←⍳0
[5]   POS←↓ (E∈⎕99961)-E∈999960
[6]   →(0=+/POS)/S1
[7]   H←1↑E
[8]   G←⍎(LI H)
[9]   C←TEXP PRENDRE H
[10]  SE←2↓¯1↓E
[11]  POS←2↓¯1↓POS
[12]  V←+/POS←(POS-SE∈999962)∈0
[13]  POS←0,(V↑∇POS),1+⍴SE
[14]  J←1
[15] B1:E←POS[J]↓(POS-1)[J+1]↑SE
[16]  E←FD E
[17]  ZD←ZD,(-J),E
[18]  J←J+1
[19]  →(J≤V+1)/B1
[20]  →(0≠⍴C)/S2
[21]  C←FN H
[22]  J SUBSTITUE C
[23]  Z←CACO
[24]  →0
[25] S2:J SUBSTITUE FD FN C
[26]  Z←CACO
[27]  →0
[28] S1:Z←E
```