Department of Mathematics
Technological University
Eindhoven,
The Netherlands.

PREPRINT. To be published in the
Proceedings of the Symposium
on APL (Paris, December 1973),
ed. P. Braffort.

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 theentries, 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. ⊢, ∅,..., 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 by 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 $\mathcal{B}$ we need the indicator string, the middle expression or the category expression of a certain line of $\mathcal{B}$. Each line in the book $\mathcal{B}$ 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 $Q \not\equiv x$ 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 $\underset{\eta}{>}$ 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 $\underset{\eta}{>}$ gets the value false.

There appear two procedures in $\underset{\eta}{>}$, 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(<A>B):=if CAT(B)≡[x,P]Q

then [x/A]Q

else <A>CAT(B)

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

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

Futher CAT is not defined for 1-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

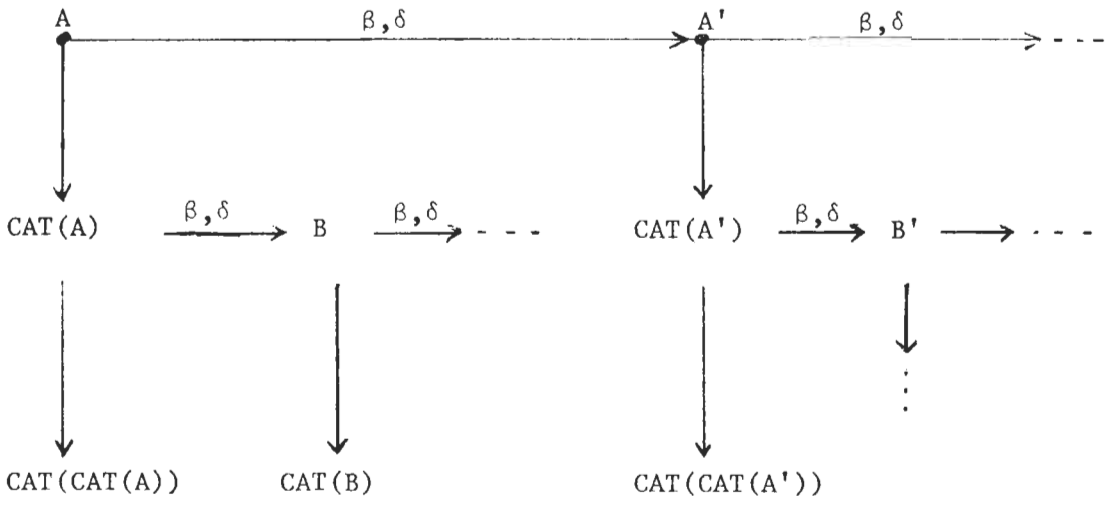
d(Σ) or the shape <B>C.

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 $A \geq C \leq B$ for some C

(see also [3, §6.3.1]).

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

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

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

Roughly speaking, in order to check $A \stackrel{D}{=} B$, the procedure tries to reduce A and/or B until either the two expressions are identical or the decision $A \stackrel{D}{\neq} B$ 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 $\Sigma_2 \stackrel{D}{=} \Sigma_4$.

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 $A_1 \stackrel{D}{=} B$.

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

8.2. Type inclusion

If we want to verify A <u>E</u> B, we check $\vdash A$ and $\vdash B$, compute CAT(A) and check $CAT(A) \stackrel{D}{=} B$; 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]type $\overset{D}{=}$type

will be accepted as correct, but *not*

type$\overset{D}{=}$[x,A]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 δTHAN 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 δreverse (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 >β 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:

$\emptyset$ * α :=EB ; type.
α * a :=EB ; α
a * f :=PN ; type
$\emptyset$ * 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, $\alpha \overset{SD}{\neq}$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.	

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.

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].