1. This [illegible] of the automatic [illegible] which is currently being operated at the THE, Technological University of Eindhoven.
The description is given in terms of a number of procedures, written in an Algol-like language.
Although most parts of the text will speak for themselves, some notes and conventions have to be explained.

2. The Algol-like language

2.1. Basis is Algol 60.

2.2 Several types are added, e.g. [illegible] string.

2.3 A construction case ... of begin ... end is added, which is used instead of a repeated if-then-else construction. The values of the case selector are placed before the actions as labels.

example

```
case color of
begin
    blue : TV color 1;
    green : TV color 2;
    red : TV color 3;
    otherwise : no TV color
end;
```

gives the same result as:

```
if color = blue then TV color 1 else
if color = green then TV color 2 else
if color = red then TV color 2 else no TV color.
```

2.4. Some non-Algol symbols are used, e.g. ≠ ... and ∅

2.5. Some procedures are described with an infix notation but in procedure definitions, e.g. ≠, + and ≥.

Each procedure, whose identifier is written in capitals or non-Algol symbols, is described.

3 Some notations

3.1 Expressions are denoted by $A, B, C, D, E, \ldots$

Expression strings are denoted by $\Sigma, \Gamma, \ldots$

A[illegible] denoted by Π

The expressions occurring in an expression string Σ are denoted by Σ_i.

3.2 Each string Σ can be separated into two parts:

~~$\Sigma(-)$: the whole string except the last expression~~

Σ [illegible] : the last expression of Σ.

$\Sigma(-)$: the rest of Σ.

Example:

Σ = [illegible]

then

Σ [illegible]

Σ [illegible]

3.3 Each expression has a shape, i.e. the characteristic form in which it appears. E.g. the shape of the expression

$\{f(a)\} g [x, y] z)$ is [illegible] $\{A\} B$

When using this shape-symbolism, we permit ourselves to use the subexpressions of it. So, in the given example we may speak about the expression A by which we mean the expression $f(a)$.

The shapes (and their symbolism) which are used are:

type

[illegible]

variable [illegible]

$d(\Sigma)$

$\{A\} B$

$[x, A] B$

[illegible]

3.4. The concept of shape is also used for Automath-lines.

[illegible]

[illegible] ([illegible])

I [illegible] ([illegible])

[illegible] (a [illegible])

3.5. The [illegible] [illegible] [illegible] [illegible] [illegible] treated in a conversational system, in order to control the amount of work the program should do in certain cases (mostly, when an error has been made). ~~[illegible] of the [illegible]~~

The statements, whose execution is [illegible], [illegible] by human intervention, are [illegible] before [illegible] [illegible] [illegible] ?

3.6.

3.6. Remark 3.5 suggests a more general [illegible]: The meaning of this [illegible] is to give a reason [illegible] how the [illegible] is constructed, [illegible] [illegible] words.
Of course, in the actual implementation, [illegible] human intervention features are [illegible], but this will probably not [illegible] the [illegible].

3.2. When [illegible] a string, we use ((and)),
e.g.

$$\Sigma = ((\Sigma^{(\cdot)} , \Sigma^{(\cdot)}))$$

$\emptyset$ denotes the empty string.

[illegible]

4. Preliminaries.

Some procedures besides the indicated ones, middle expression or category expressions.
We will introduce for these:
expression procedure INDEXTR(d);
defined name d;
comment: INDEXTR becomes the value of the index expression of the line in which d is defined; ——— ;

expression procedure MIDDLE(d);
defined name d;
comment: MIDDLE becomes the value of the middle expression of the line in which d is defined; ——— ;

expression procedure CATEGORY(d);
defined name d;
comment: CATEGORY becomes the value of the category expression of the line in which d is defined; ——— ;

5. Substitution

5.1. The first operation on expressions we describe is a substitution.
We define a transformation S with three parameters:
E : an expression;
v : a string of variables (variable identifiers);
T : an equally long string of objects. ~~to be substituted for v in E.~~
Each v_i in E is replaced by the corresponding object T_i.

The procedure S will become the name of the resulting expression, denoted by $S_T^v E$.

Also the analogue $S_T^v \Sigma$ is defined.

5.2 expression procedure $S^{v}_{\Gamma}\ E$;

```
expression value E; expression string value v, Γ;
comment shape(v_i) must be : variable;
S :=
case shape(E) of
begin
    variable : if ∃_i (v_i ≡ E) then Γ_i else E;
    d(Σ)     : d(String S^v_Γ Σ);
    {Θ} B    : {S^v_Γ Θ} S^v_Γ B;
    [x, Θ] B : [x, S^v_Γ Θ] S^v_Γ B;
    otherwise : E;
end;
```

5.3. expressionstring procedure $\mathrm{String}\ S^{v}_{\Gamma}\ \Sigma$;

```
expressionstring value v, Γ, Σ;
comment shape(v_i) must be : variable.
    String S is the string-analogue of S;
```

$\mathrm{String}\ S :=$ if $\Sigma \equiv \emptyset$ then $\emptyset$
else $((\mathrm{String}\ S^{v}_{\Gamma}\ \Sigma^{(-1)},\ S^{v}_{\Gamma}\ \Sigma^{(+)}))$

5.4. Another substitution is needed to perform the so-called α-reductions (renaming of bound variables). What we want is to replace, in an expression, all occurrences of one bound variable by a new bound variable. We will call this procedure dummyS [illegible]

E : the expression in which the substitution occurs.
old : the [illegible] variable
new : the new bound variable

[illegible] result of dummyS [illegible]

5.5
```
expression procedure dummy S^old_new E ;
begin var value old, new; expression value E ;
dummy S :=
case shape E of
begin
   boundvar   : if E ≡ old then new else E ;
   λ ( Σ )    : λ ( String dummy S^old_new E ) ;
   { A } B    : { dummy S^old_new A } dummy S^old_new B ;
   [x, A] B   : [x, S^old_new A] S^old_new B ;
   otherwise  : E ;
end ;
```

5.6.
```
expressionstring procedure String dummy S^old_new Σ
begin var value old, new; expressionstring value Σ ;
String dummy S :=
if Σ ≡ ∅ then ∅
else (( String dummy S^old_new Σ^[illegible] , dummy S^old_new Σ^[illegible] )) ;
```

6. Reductions.

The reductions which are implemented are:

α-reduction

β-reduction

δ-reduction

η-reduction.

6.1 α-reduction.

The separate α-reduction has not been implemented. If one wants α-reduction, this can only be done by using dummy S[illegible].

6.2 β-reduction.

The β-reduction is a rather strong one. If the presented expression has the shape {.}B, but the shape of B is not [x, .]D, then the procedure tries to reduce B to this shape, and then performs the β-reduction.

Also a boolean procedure $A \geq_{\beta} B$ is described, which tries to β-reduce the expression A. When it succeeds, the procedure [illegible] in the value true, and A becomes the result.

6.3 δ-reduction.

The δ-reduction performs (if possible) one single definition application on the presented expression, whose shape is [illegible] δ(Σ ;). The result is [illegible] the middle expression of the [illegible] whose definition [illegible] depends; the free variables in it [illegible] must be replaced by the [illegible] expressions Σ;.

Also a boolean procedure $A \geq_{\delta} C$ is described, in the same way as for β-reduction.

6 11. η-reduction

It is [illegible] possible to discuss η-reduction.
In this case η-reduction is suppressed for the [illegible]
[illegible] operation.)

As with the [illegible], if the [illegible] expression has
the shape $[x, x]B$, but B has not the shape $[x]D$, then
the [illegible] tries to reduce B to this shape.
[illegible] $[x]B$, if B does have this shape
$[x]C$, but if B [illegible], the reduction [illegible]
B [illegible] η-reduction on B.)

If no [illegible] procedure $A \overset{*}{\underset{\eta}{\rightarrow}} B$ is discussed, it is the
same [illegible] η-reduction.

[illegible]

6.5 procedure BETA (E, possible, result);
expression result; expression value E; boolean possible;
if shape(E) = {A}B then
begin boolean continue; continue := true;
while shape(B) ≠ [*, C*]D and continue do
case shape(B) of
begin
{C}D: BETA(B, continue, B);
d(Σi): DELTA(B, continue, B);
otherwise: continue := false
end case;
if shape(B) = [*, C*]D then
begin possible := true; result := S_A^* D;
end
end
else possible := false;

```
6.6. procedure DELTA (E, possible, result);
     expression result; expression value E; boolean possible;
     if shape(E) = d(Σ_i) then
     begin if d is a defined notion then
           begin possible := true; result := S_{Σ_i}^{indsh(d)} definition(d);
           end
           else possible := false
     else possible := false;

6.7. procedure ETA (E, possible, result);
     expression result; expression value E; boolean possible;
     if etareduction allowed then
        if shape(E) = [x, A]B then
           if shape(D) = {C}D then
              if x ≡_D C then
                 if x ∉ FV(D) then
                 begin possible := true; result := D;
                 end
                 else possible := false                    *
              else
              begin boolean betapossible;
                 BETA(D, betapossible, B);
                 if betapossible then ETA([x, A]B, possible, result)
                 else possible := false;
              end
           else
           if shape(B) = [x, C]D then
           begin boolean etapossible;
              ETA(B, etapossible, B);
              if etapossible then ETA([x, A]B, possible, result)
              else possible := false;
           end
           else possible := false
        else possible := false
     else possible := false;
```

```
procedure ETA( E, possible, result);
expression value E; boolean possible, expression result;
{ reduce an abstracted thing
  if shape(E) = [x,A]B then
  case shape(C) of
  begin
    {x}x : if B ≥ B1 then ETA( [x,A]B1, possible, result)
           else possible := false;
    {C}D : if x ∉ C then
             if x ∈ FV(D) then
             begin possible := true; result := D;
             end
             else
             begin try to reduce(D) to an expression
                   not containing x (*) and give
                   if (possible) then (result);
               if not possible then
                 if B ≥ B1 then ETA( [x,A]B1, possible, result)
                 else possible := false
               else;
             end *
           else
             if B ≥ B1 then ETA( [x,A]B1, possible, result)
             else possible := false;
    [x,d]D : if B ≥ B1 then ETA( [x,A]B1, possible, result)
             else possible := false;
    otherwise: possible := false;
  end
  else possible := false
else possible := false;
```

for example,

6.7 1. * is a shortcoming.
It is possible that D looks like $f(x,y)$,
but $f(x,y)$ is defined by: y.
Then, after S-reducing D, the expression is
η-reducible.

6.8. boolean procedure A $\geq_p$ B ;
expression value A; expression B;
comment if A is reducible by BETA, then B becomes the result;
begin boolean possible;
BETA (A, possible, B); $\geq_p$:= possible;
end;

6.9. boolean procedure A $\geq_\delta$ B ;
expression value A; expression B;
comment if A is reducible by DELTA, then B becomes the result;
begin boolean possible;
DELTA (A, possible, B); $\geq_\delta$:= possible;
end;

6.10. boolean procedure A $\geq_\eta$ B ;
expression value A; expression B;
comment if A is reducible by ETA, then B becomes the result;
begin boolean possible;
ETA (A, possible, B); $\geq_\eta$:= possible;
end;

7. Other expression-computing procedures

7.1. Two other expression-computing procedures are needed:

CAT(E)
DOM(E).

7.2. CAT(E).

Every correct expression has a category (i.e. [illegible] type and prop). The category [illegible] is computed [illegible] ~~[illegible] expression [illegible]~~ ~~expression [illegible] a wider category~~ (i.e. one [illegible]; cf. [illegible] A ∈ P and B ∈ C, then [illegible] A ∈ C) The "automatic" category may be computed by the procedure CAT, as follows:

```
expression procedure CAT(E);
expression value E;
CAT :=
case shape(E) of
begin
   variable :  CATEGORY(E);
   d(Σ)     :  S_Σ^{INDSTR(d)} CATEGORY(d),

   {A}B     :  {A} CAT(B);
   [x,A]B   :  [x,A] CAT(B);
   otherwise undefined;
end;
```

73. DOM(E);

The domain of an λ-expression is the category of the abstraction [illegible] to which it may be applied. λ-expressions are (in [illegible] written) as [x, A]B, [illegible]

If we have an Automath line:

I * d := E₁ . E₂ .

then the domain of d is computed by taking the domain of E₁, [illegible] taking the [illegible] of E₂.
In some cases, the domain of E₁ is not computable (e.g. $E_1 \geq PN$), in other cases the domain of E₂ is not computable (e.g. E₂ = type).
For this reason the procedure tries first to compute the domain of E₂. If this does not succeed, β-reduction on E₁ is tried, and if this succeeds, the result is again presented to DOM.
[illegible] of the procedure.

```
expression procedure DOM(E);
expression value E;
DOM :=
case shape(E) of
begin
  variable : DOM(CAT(E));
  d(Σ)     : if DOM(CAT(E)) = undefined then
               if E ≥δ E1 then DOM(E1)
               else undefined
             else DOM(CAT(E));
→ {Σ}[illegible] : if E ≥β E1 then DOM(E1) else undefined.

  [x, A]B  : A;
  otherwise : undefined;
end;
```

8. Definitional equivalence ($\stackrel{D}{=}$)

8.1. General remarks

Formally — [illegible] clear [illegible] expressions A and B, is defined by:

$$!\ \exists_C\ (A \gg C \wedge B \gg C) \text{ then } A \stackrel{D}{=} B .$$

($\gg$ means: a sequence of zero [illegible] reductions)

For this reason the procedure $\stackrel{D}{=}$ presented [illegible] the same [illegible] which obtain [illegible] two expressions. The question as to how to reduce these two expressions, and in what order, is a matter of strategy. Of course, [illegible] in fact [illegible] en [illegible] is naturally [illegible] greater.

And [illegible] question [illegible] type procedure [illegible]

$$\text{if } B \in \text{types}$$
$$\text{then } [x, A]\, B \in [x, A]\, \text{type} ,$$
$$\text{but also } [x, A]\, B \in \text{type}$$

For this reason the procedure $\stackrel{D}{=}$ is written so that

~~$[x, A]\, \text{type} \stackrel{D}{=} [x, A]\, \text{type}$~~

~~and~~ $[x, A]\, \text{type} \stackrel{D}{=} \text{type}$,

is confirmed as correct. (N.B. when the expression in the left is the automatically computed category of $[x, A]\, B$)

Not accepted is: $\text{type} \stackrel{D}{=} [x, A]\, \text{type}$.

So for this kind of expressions the procedure is not symmetric.

8.2. Some procedures for $\stackrel{D}{=}$.

8.2.1. OLDER THAN.

"If one wants to check $d(\Sigma) \stackrel{D}{=} b(\Gamma)$, and $d \neq b$, then one tries to reduce one of the two, with δ-reductions. It seems a good strategy to reduce the younger of the two (i.e., the constant b or d, which is the more recently defined), for, in this way, we have a chance of reducing it to the other."

So we have:

```
boolean procedure d OLDER THAN b;
defined name value d, b;
comment OLDER THAN := the line in which
        d is defined, appears later in the book
        than the line in which d is defined; ——;
```

8.2.2. NF. (normal form)

Deep analysis shows that it is possible to show two expressions, of shape: $\{D\}D$ and $\{C\}D$, where

$$D \stackrel{D}{=} C, \quad \{D\}D \stackrel{\beta}{\geq} E_1, \quad \{C\}D \stackrel{\beta}{\not\geq} E_2,$$

and, still,

$$\{D\}D \stackrel{D}{=} \{C\}D.$$

Example:

For this reason the procedure NF is introduced:

```
express on procedure NF(E);
expression value E;
comment NF := the normal form of E; ——;
```

8.2.2. dummy S_{new}^{old} ;.

This procedure (already) has been defined : see 5.5.

8.2.4. SD

This is the string analogue of D .
The definition follows that of D .

§3 boolean procedure $E_1 \overset{D}{=} E_2$;
expression value E_1, E_2 ;
$\overset{D}{=}$:=
case (shape(E_1), shape(E_2)) of
begin
(type, type) : true ;
(type, otherwise) : false ;
(prop, prop) : true ;
(prop, otherwise) : false ;
(variable, variable) : E_1 ... E_2
(variable, $d(\Sigma)$) : if $E_2 \geq_\delta E_{22}$ then $E_1 \overset{D}{=} E_{22}$ else false ;

(variable, $\{A\}B$) : if $E_2 \geq_\beta E_{22}$ then $E_1 \overset{D}{=} E_{22}$ else false ;

(variable, $[x, H]D$) : if $E_2 \geq_\gamma E_{22}$ then $E_1 \overset{D}{=} E_{22}$ else false ;

(variable, otherwise) : false ,
(bound var, otherwise) : consider (variable, ...) ;
($d(\Sigma)$, $b(\Gamma)$) : if $d \equiv b$ then
if $\Sigma \overset{SD}{=} \Gamma$ then true
else if $E_1 \geq_\delta E_{11}$ then $E_{11} \overset{D}{=} E_2$ else false
~~if $E_2 \geq_\delta E_{22}$ then $E_1 \overset{D}{=} E_{22}$ else false~~
else if d OLDER THAN b then
if $E_2 \geq_\delta E_{22}$ then $E_1 \overset{D}{=} E_{22}$ else false
else if $E_1 \geq_\delta E_{11}$ then $E_{11} \overset{D}{=} E_2$ else false ;

($d(\Sigma)$, $\{A\}B$) : if $E_2 \geq_\beta E_{22}$ then $E_1 \overset{D}{=} E_{22}$ else
if $E_1 \geq_\delta E_{11}$ then $E_{11} \overset{D}{=} E_2$ else false

($d(\Sigma)$, $[x, A]B$) : if $E_2 \geq_\gamma E_{22}$ then $E_1 \overset{D}{=} E_{22}$ else
if $E_1 \geq_\delta E_{11}$ then $E_{11} \overset{D}{=} E_2$ else false ;

($d(\Sigma)$, otherwise) : consider reverse (i.e. (shape(E_2), shape(E_1)))

$(\{A\}B, \{C\}D)$: if $A \overset{D}{=\!=} C$ and $B \overset{D}{=\!=} D$ then true
else if $E_1 \geq_\beta E_{11}$ then $E_{11} \overset{D}{=\!=} E_2$ else
if $E_2 \geq_\beta E_{21}$ then $E_1 \overset{D}{=\!=} E_{21}$ else error ;

$(\{A\}B, [x, C]D)$: if $E_1 \geq_\beta E_{11}$ then $E_{11} \overset{D}{=\!=} E_2$ else
if $E_2 \geq_\beta E_{21}$ then $E_1 \overset{D}{=\!=} C_{21}$ else false ;

$(\{A\}B$, otherwise) : error, reverse ;

$([x, A]B$, type) : $B \overset{D}{=\!=} E_2$;
$([x, A]B$, prop) : $B \overset{D}{=\!=} E_2$;

$([x, A]B, [y, C]D)$: if $A \overset{D}{=\!=} C$ then $B \overset{D}{=\!=} \text{dummy} \overset{y}{\underset{x}{\ }} D$ else false ;

$([x, A]B$, otherwise) : error, reverse ;
end ;

84. boolean procedure $\Sigma_1 \overset{SD}{=\!=} \Sigma_2$;
expression string value Σ_1, Σ_2 ;
comment $\overset{SD}{=\!=}$ is the string analogue of $\overset{D}{=\!=}$;

$\overset{SD}{=\!=}$:= if $\Sigma_1 = \emptyset$ then $\Sigma_2 = \emptyset$
else $\Sigma_1^{(-)} \overset{SD}{=\!=} \Sigma_2^{(-)}$ and $\Sigma_1^{(+)} \overset{D}{=\!=} \Sigma_2^{(+)}$;

9. Correctness of expressions. (⊢)

9.1. Correctness of an expression, say E, is only defined with respect to an indicatorstring, say I.
We will denote this by:

$$I \vdash E$$

The use of 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) ≚ DOM(B)

2: if shape(E) = d(Σ) then
firstly: all E_i must be correct,
secondly: all E_i must have correct categories.
With the second case there is a difficulty.
Let us consider the following book:

∅ * nat := EB ; type.
nat * a := EB ; nat.
a * f := PN ; type
∅ * real := EB ; type
real * b := EB ; real
b * g := f(real, b) ; type.

Now b ⊢ f(real, b), nevertheless the string of types expected by f is not definitionally equivalent to the string of given types:
type, nat ≠ type, real

We may conclude that, after checking the definitional equivalence of the first two categories, we have to replace, in the category string of (nat, a), the variable nat by real.

This replacement ('substitution') is, in a more general way, done by the procedure CORRECTCATS

9.2 boolean procedure CORRECTCATS(Σ, I);
expressionstring value Σ, I;

CORRECTCATS :=
if Σ ≡ ∅ then I ≡ ∅ else
CORRECTCATS(Σ^(-), I^(-)) and

$$CAT(\Sigma^{(+)}) \stackrel{D}{=} S_{\Sigma^{(-)}}^{I^{(-)}} CAT(I^{(+)});$$

9.3 boolean procedure I ⊢ E;
expressionstring value I; expression value E;
⊢ :=
case shape(E) of
begin
type : true;
prop : true;
variable : ∃_I (I_i ≡ E);
d(Σ) : I ⊢_S Σ and CORRECTCATS(Σ, INDSTR(d));

{A}B : I ⊢ A and I ⊢ B and CAT(A) ≡ DOM(B);

[x, A]B : I ⊢ A and (I, x) ⊢ B;
otherwise : false;
end;

9.4 boolean procedure I ⊢_S Σ;
expressionstring value I, Σ;
comment ⊢_S is the string analogue of ⊢;
⊢_S :=
if Σ ≡ ∅ then true
else I ⊢_S Σ^(-) and I ⊢ Σ^(+);

9.5. A comment about $I \vdash [x, A]B$.

In this case, the checker adds, after checking $I \vdash A$, a waste-line to the book, of the form:
in the expression B, I ✓ x :≡ ε B ; A.
Now, the bound variable x is replaced by the free variable x —— .
For this reason the correctness of a bound variable will never be looked for, and its CAT or DCH will never be computed.
Only in $\overset{D}{=}_{com}$ the shape bound var —— occurs.

10 The correctness of an Automath line.

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

```
boolean procedure CORRECT(LINE);
automath line value LINE;
CORRECT :=
case shape(LINE) of
begin
    I ⊢ N :≡ εB ; E.   : I ⊢ E ;
    I ⊢ N :≡ PN ; E.   : I ⊢ E ;
    I ⊢ N :≡ E1 ; E.   : I ⊢ E1 and I ⊢ E2 and
                         CAT(E1) =D= E2 ;
    otherwise : false ;
end ;
```

11. Final remarks.

11.1. Again we say: the given procedures are only an outline of the real checker. Many more parameters are passed through the procedures to avoid duplications, to control critical passages, to give the possibility of sending information about the progress of the checker to the user, etc.

11.2. With regard to efficiency, improvements are possible. For example, more of the strategy incorporated in [illegible], in more or less [illegible] experience and research, will lead to a better strategy.

[illegible]

11.3. We are pleased to report, in any event, that the checker has been working satisfactorily up to now!