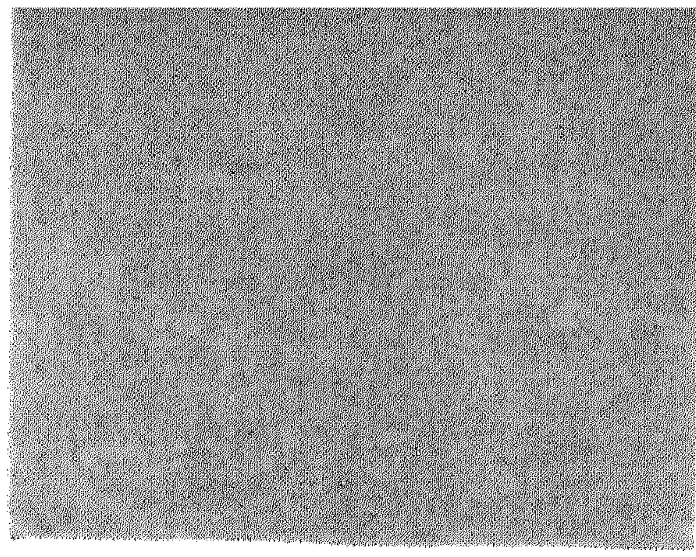
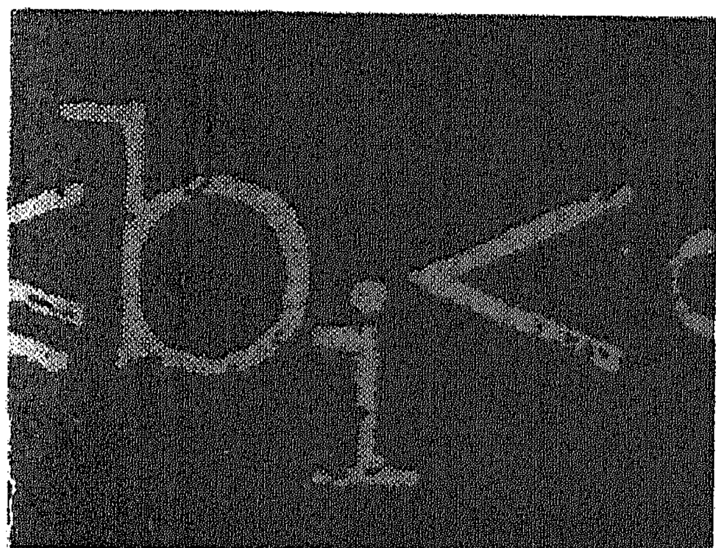
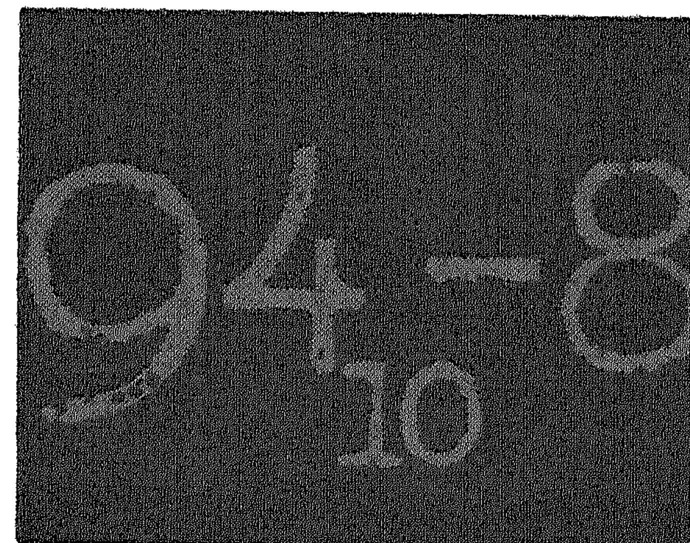
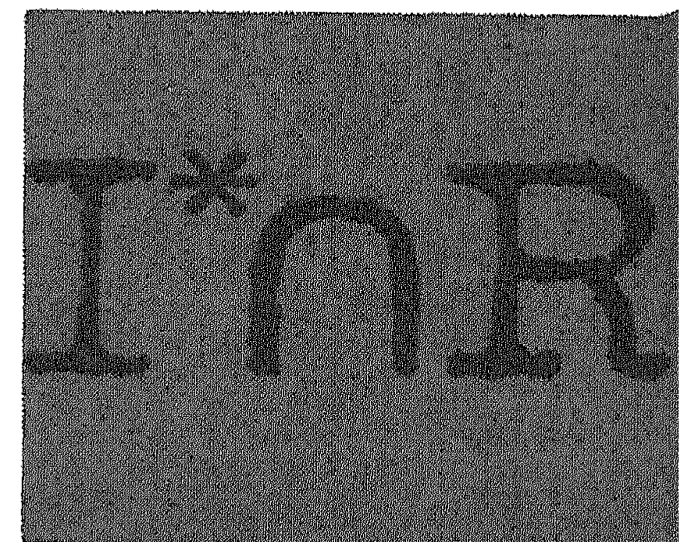
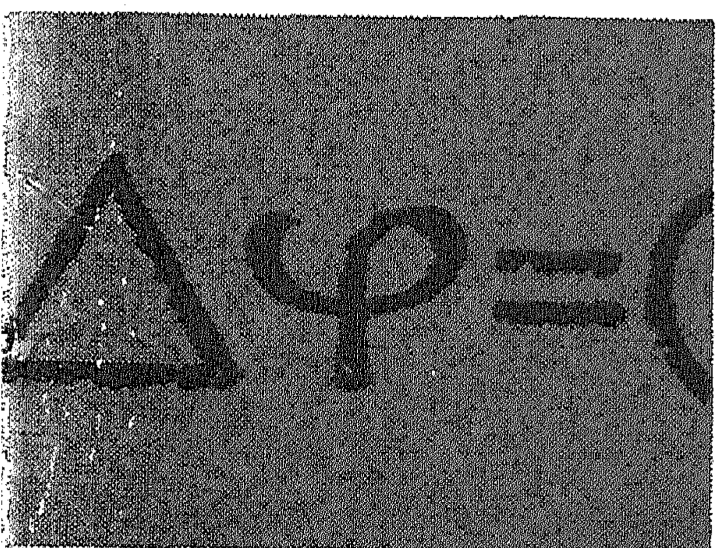
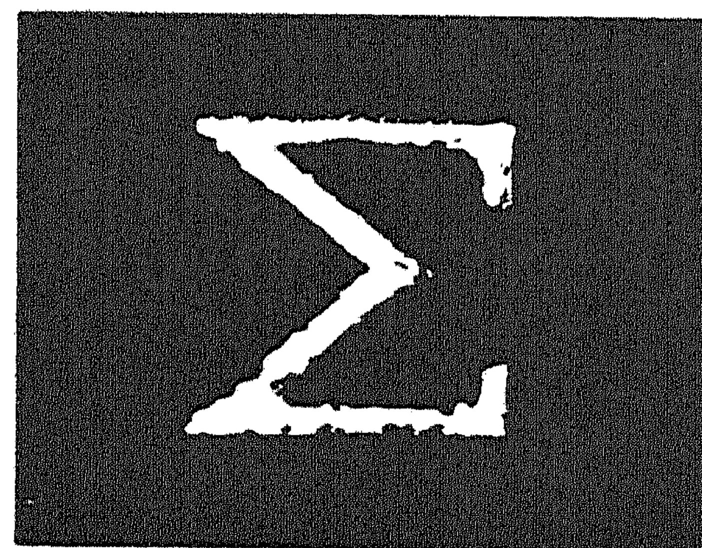
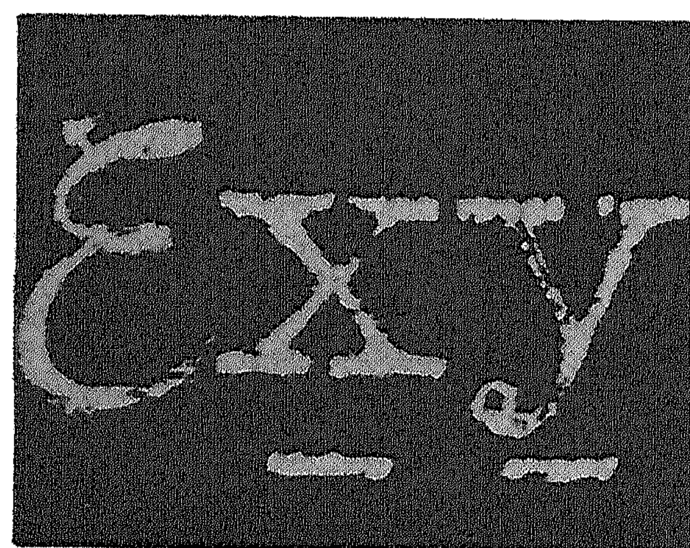
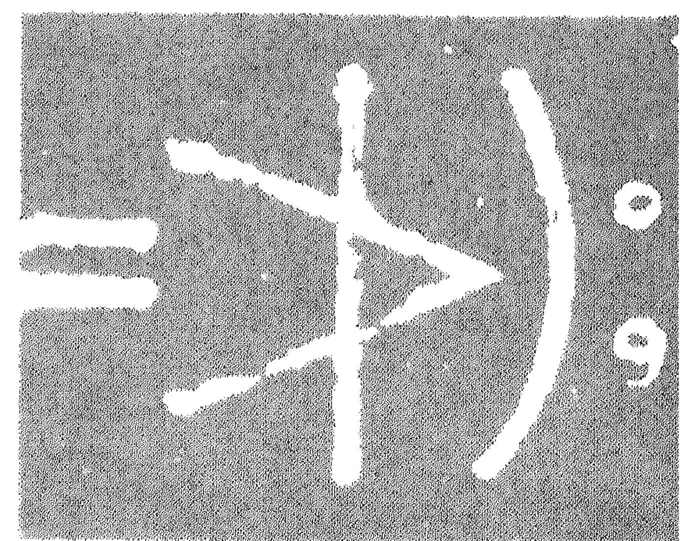
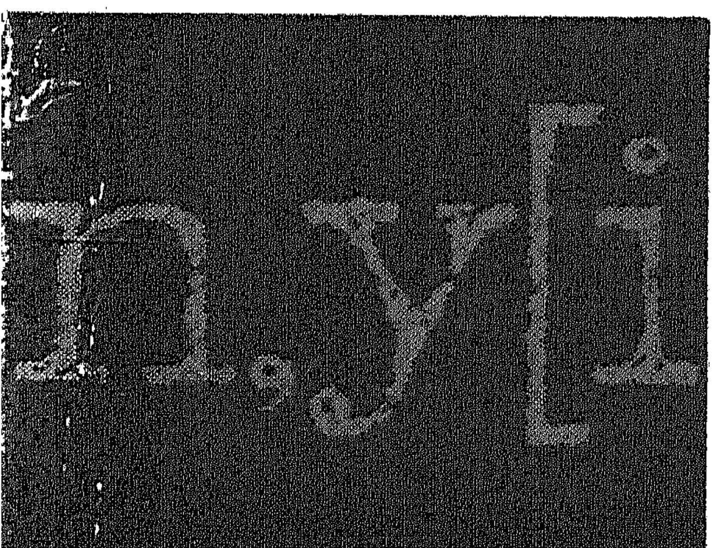
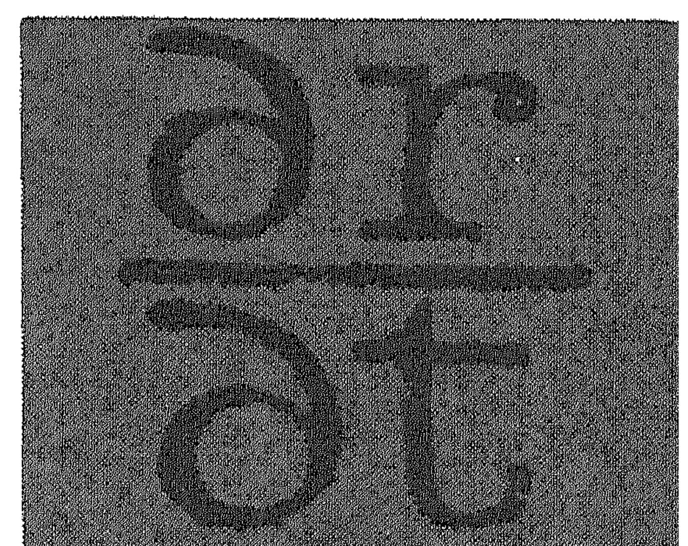


FORMULA MANIPULATION IN ALGOL 60

PART 1

R. P. VAN DE RIET



MATHEMATICAL CENTRE TRACTS 17

MATHEMATICAL CENTRE TRACTS

17

MATHEMATICAL CENTRE TRACTS
17

FORMULA MANIPULATION IN ALGOL 60

PART 1

BY

R.P. VAN DE RIET

RA

2nd Edition

MATHEMATISCH CENTRUM AMSTERDAM

1970

Acknowledgements

The author is grateful to professor A. van Wijngaarden for the stimulating discussions during the course of the investigations and for his critical remarks, concerning this treatise, which led to considerable improvements.

The author is grateful to professor H.A. Lauwerier for his approval for performing the investigations, as well as for his interest.

The author is grateful to dr. M. van den Tempel of the UNILEVER Research Laboratory in Vlaardingen for his interest in the investigations which originated from a study on the damping of surface waves by soap.

The author is grateful to professor P.J. Zandbergen, of the Technological University of Twente, who suggested investigating the blunt-body problem.

The author is grateful to drs. C. Traas, of the "Nationaal Lucht- en Ruimtevaart Laboratorium" in Amsterdam, who was very willing to supply his theoretical and numerical results concerning the blunt-body problem.

The author is grateful to the staff of the Computation department of the Mathematical Centre for running the programs, to Mrs. H. Roqué for typing and retyping the manuscript and to Messrs. D. Zwarst and J. Suiker for reproducing it.

PREFACE

to the second edition

This second edition does not deviate substantially from the first one. The changes concern minor misprints mostly; a few more serious changes in the text and in the ALGOL 60 programs concern simplification, truncated power series and the erase-but-retain process.

Amsterdam, April 1970 R.P.R.

SUMMARY

The first part of this treatise describes:

a. ALGOL 60 procedures for the following formula manipulations:
   1. Simplification; for example,
      "$(x^2 - 1)/(x^2 + 2x + 1)$" is transformed into
      "$(-.5x + .5)/(-.5x - .5)$" and
      "$\sin^2(x + y) + \cos^2(y + x)$" is transformed into "1".
   2. Manipulation of truncated power-series (the coefficients of which are arbitrary; e.g. truncated power-series).
   3. Differentiation.
   4. Complex Conjugation.
   5. Substitution.
   6. Calculating the greatest common divisor.

b. The syntax and the semantics of a special programming language in which one can program several formula manipulations in the form of a "formula program".
   Besides the above-mentioned, these formula manipulations are:
   7. Solving quasi-linear equations with nonnumeric coefficients.
   8. Producing output, the form of which may be specified to be either ordinary, or special.
      Output, produced in the special form, has the form of ALGOL 60 statements suitable for use in a complex-arithmetic ALGOL 60 program.

Finally, the ALGOL 60 program is given which reads the formula program from input tape and executes the instructions.

The second part of this treatise describes:

a. The ALGOL 60 procedures for complex arithmetic.

b. An ALGOL 60 program for solving the multidimensional, nonlinear and implicit Cauchy problem for several unknown functions:
   "$F(\nabla u,u,x) = 0$", where "∇u" stands for all derivatives "$\partial u_i/\partial x_j$".

This program defines output in the form of ALGOL 60 statements to be used in another ALGOL 60 program which computes, in an efficient way, the Taylor coefficients of the unknown functions.

TABLE OF CONTENTS OF PART 1

SUMMARY I
TABLE OF CONTENTS III
INTRODUCTION 1
1. A SIMPLE FORMULA MANIPULATION SYSTEM 6
2. THE GENERAL FORMULA MANIPULATION SYSTEM 14
2.1. The form of formulae in the general system 14
2.2. How to use the general system 17
2.3. The internal representation of formulae in the system 19
2.4. The heading of the general system 20
2.5. The internal representation of formulae 22
2.6. The basic procedures 24
2.7. The "erase but retain process" 32
2.8. The procedures *S*, *D*, *P*, *Q*, *POWER*, and *INT POW* 38
2.9. The functions 50
2.10. The simplification procedures 52
2.11. The procedures *QUOTIENT* and *COMMON DIVISOR* 61
2.11.1. Definition of the integral-division process 61
2.11.2. The procedure *QUOTIENT* 70
2.11.3. The common-divisor process 74
2.12. Supplementary equipment 81
2.13. Output 86
2.14. Garbage collection 91
3. THE FORMULA PROGRAM (A convenient input/output system) 95
3.1. Introduction 95
3.2. The definition of a formula program 97
3.2.1. The syntax of a formula 97
3.2.2. The semantics of a formula 98
3.2.2.1. A formula 98
3.2.2.2. An identifier 98
3.2.2.3. A truncated power-series 99
3.2.2.4. An elementary-function designator 100

3.2.2.5. A derivative 100
3.2.2.6. A simplified formula 101
3.2.2.7. A complex conjugate 101
3.2.2.8. A result of substitution 101
3.2.2.9. An integral quotient and a common divisor 102
3.2.3. The syntax of a formula program 103
3.2.4. The semantics of a formula program 104
3.2.4.1. The heading 104
3.2.4.2. A compound formula statement 105
3.2.4.3. A formula block 105
3.2.4.4. NLCR 106
3.2.4.5. EXPAND, NOT EXP 106
3.2.4.6. A formula assignment statement 107
3.2.4.7. A print-string statement 107
3.2.4.8. The output statement involving OUTPUT R 108
3.2.4.9. The output statement involving OUTPUT C 108
3.2.4.10. A special-derivative statement 109
3.2.4.11. A declaration to be of real type statement 110
3.2.4.12. A coefficients of power series statement 110
3.2.4.13. A solve linear equations statement 111
3.2.4.14. Formula identifier vs. algebraic variable 113
3.2.5. Execution of a formula program 114
3.2.6. Discussion 116
3.2.7. Examples 117
3.3. The formula-program processor 147
3.3.1. The first part of the processor 147
3.3.2. The second part of the processor 153
3.3.2.1. The input of a formula 154
3.3.2.2. The output of a formula 157
3.3.2.2.1. The optimal form 157
3.3.2.2.2. The ordinary form 158
3.3.2.2.3. The complex form 160
3.3.2.2.4. The output of numbers 162

3.3.2.2.5. The output procedures 166
3.3.2.3. The derivative procedure 174
3.3.3. The third part of the processor 176
Appendix 183
References 186
SAMENVATTING 190

INTRODUCTION

The electronic digital computer, already extensively used in numerical mathematics, will be used more and more in other branches of mathematics. One of these branches is elementary algebra, used throughout by mathematicians and physicists.
It occurs frequently that one has to do quite an amount of long and tedious, but elementary algebraic calculations, to solve some problem in, say, mathematical physics.
It is for these calculations, henceforth called "formula manipulations", that the computer may become useful.
Two possible systems to instruct the computer to manipulate formulae are described.

Before these systems are discussed it is necessary to elucidate the term "formula manipulation".
From Jean Sammet [20] the following rather rough definition is borrowed:

"Formula manipulation means the use of a computer to operate on mathematical expressions in which not all the variables are replaced by numbers and in which some meaningful mathematical operation is to be done ...".

In this treatise the mathematical expressions are particular arithmetic expressions (see the ALGOL 60 report [11]), which, in the sequel, are called formulae; the form of the formulae is described in chapters 1 and 2. The meaningful mathematical operations are, for example: differentiation, simplification, complex conjugation, calculating a common divisor of two formulae, substitution.

The two systems with which one can manipulate formulae on a computer consist of sets of ALGOL 60 procedures.
Chapter 1 describes one system, which manipulates formulae of a very simple form and is therefore called the simple system.
The formulae of the simple system are built up by means of algebraic variables and the operators + and ×.

The capabilities of this system are differentiation and outputting.
Chapter 2 describes the other system, which manipulates formulae built up by means of numbers (including complex numbers), algebraic variables, truncated power-series with arbitrary formulae as coefficients, trigonometric and logarithmic functions and the operators +, −, ×, /, ↑.
This system, called the general system, has the following capabilities: simplification, manipulation of truncated power-series, differentiation, complex conjugation, substitution.

The reason for describing the simple system is that it serves as an introduction to the general system; a program using the simple system is completely explained.
Moreover, due to the fact that the simple system is much shorter and considerably less time consuming than the general system, it may be more useful than the general system in all those formula manipulations, where the formulae have the simple form as defined above. Even if the formulae are more complicated, e.g. if as constituants also numbers and as operators also"raising to an integral power"is allowed, but do not contain the division operator, then the simple system can be very useful since it can easily be extended into a system which handles these formulae and which has the important capability of simplification. In fact, the combination of division and simplification in the general system is one of the reasons why it is rather lengthy and time-consuming.

Chapter 3 gives the ALGOL 60 program which contains the general system and which is able to read an input tape and to produce an output tape.
On the input tape the user may specify his formula manipulations by means of a, so-called, "formula program". The definition of a formula program and the instructions how to prepare the input tape are, together with examples, described in detail in this chapter.
This program, which henceforth will be called "formula-program processor", may produce output tape in two forms. One form is the normal form, i.e. formulae are outputted in ordinary notation,for example "a + b × c"; the other form is adapted for use as statements in an ALGOL 60 program for performing complex arithmetic.

Chapter 4 is devoted to the ALGOL 60 procedures for complex-arithmetic computations. Procedures for the elementary operations +, −, × and / are given as well as procedures for the elementary functions: exp, ln, sin, cos, sqrt, arcsin, arccos and arctan. The procedures for the multivalued functions are constructed in such a way that the user may specify his own principal value.

Chapter 5 applies the general system to solving the nonlinear, multidimensional and implicit Cauchy problem for several unknown functions and their derivatives. The principle of the solution process is the theorem of Cauchy-Kowalewski about the existence and convergence of Taylor series for the unknown functions. The Taylor coefficients are calculated by means of an ALGOL 60 program, whose main part is generated by another ALGOL 60 program. Much effort has been given to make the first program as efficient as possible with respect to the memory space in which the numerous Taylor coefficients should be stored.

There exist already quite a number of publications on the subject of formula manipulation, almost all reviewed in the already quoted paper [20] and in the papers [21,22] of the same author.
Several systems for formula manipulation have been developed, such as FORMAC [3], Formula ALGOL [12] or the ALTRAN-ALPAK system [4], to mention some important and general systems. All these systems have, however, the disadvantage that they require special translators.
Another disadvantage is that a user, who has access to a computer for which one of these systems is written, has to know machine code if he wants to modify the system somewhat.

Since both systems,described in this report,are completely written in ALGOL 60, everyone,who has access to a computer with an ALGOL 60 compiler,can not only use them, but can also write some modified system which can solve his special problem.
Furthermore, a consequence of writing the systems in ALGOL 60 is their unambigious definition.

The disadvantages of, in particular, the general system should also be mentioned: the system needs much storage space and much time.

If one does not need the full generality of the system one may modify the system so as to gain storage space and time. (This has been done, for example, for the program of chapter 5.) Another way of gaining time is to write machine-code procedures for the internal representation of formulae.

For applications and preliminary versions of formula-manipulating ALGOL 60 procedures, the reader is referred to [14,15,2] in which programs are discussed for obtaining, respectively:

1. the second-order approximation of the solution of the (nonlinear) Navier-Stokes equations coupled with the (nonlinear) diffusion equation, together with complicated boundary conditions at a free surface of a liquid;
2. the Taylor expansion of functions defined by the Cauchy problem in a less sophisticated way than is done in chapter 5;
3. the asymptotic expansion of rather arbitrary integrals by means of the method of steepest descent (the saddle-point method).

In [17] preliminary versions of chapters 1, 2 and 3 are given, while the contents of chapter 4 has been published in [18].

The ALGOL 60 programs, given in the sequel, are reproduced by means of a FRIDEN Flexowriter with a type different from the type of the typewriter which has been used for the accompanying text. In order to facilitate the reading of the accompanying text, ALGOL 60 symbols are printed in *italics*.

In an appendix we give some properties of the X8-ALGOL 60 compiler and a description of the procedures which are used without declaration. These are standard procedures of the Mathematical Centre.

We mention, in particular, the input/output procedures.
READ (and *read*), *RESYM*, for *READ*ing numbers and *SYM*bols; *NLCR*, *PUNLCR* for printing and *PU*nching a *N*ew *L*ine *C*arriage *R*eturn symbol; *PRINTTEXT* and *PUTEXT* for *PRINT*ing and *PU*nching a *TEXT* given between the Mathematical Centre version of string quotes "⊀" and "⊁"; *ABSFIXT*, *ABSFIXP*, *FIXT*, *FIXP*, *FLOT*, *FLOP* for *T*yping and *P*unching numbers in *FIX*ed-point or in *FLO*ating-point notation; *PRSYM* and *PUSYM* for *PR*inting and *PU*nching a *SYM*bol.

For typographical reasons the ALGOL 60 symbol ÷ is printed as :.

We made sometimes use of the property of the X8-ALGOL 60 compiler that expressions are evaluated from left to right and that parameters, of procedures, which are called by value,are evaluated in the order in which they occur in the value list.
It is possible to make the programs independent of these special properties at the cost of some extra programming.
Wherever these properties are used this is signalled by means of a note.

Metalinguistic variables which are used in this treatise, but which are not defined are considered to be defined in the ALGOL 60 report [11].

The programs,described in the sequel,were run and tested for ALGOL 60 correctness on the Electrologica X8 computer and the Electrologica X1 computer, both of the Mathematical Centre. For the X8 the ALGOL 60 compiler written by F.E.J. Kruseman Aretz was used; whereas the two ALGOL 60 compilers,respectively written by E.W. Dijkstra and J.A. Zonneveld and by P.J.J. van de Laarschot and J. Nederkoorn,were used for the X1.

Chapter 1

A SIMPLE FORMULA MANIPULATION SYSTEM

Prior to chapter 2 which describes the general system, this chapter contains a completely worked out program, which performs some very simple formula manipulations.
The procedures declared in this program,together with the heading of the program and some initializing statements,form the simple system.
By means of the reproduced program it will be shown how formula-manipulation is internally executed by the computer. Since the internal representation of the formulae is essentially the same for the simple and for the general system, it is not necessary to explain in chapter 2 the detailed mechanism by which the computer treats formulae in the general system.

The formulae to be manipulated occur in the ALGOL 60 program as particular expressions of type integer.
They are syntactically defined as follows:

```
<formula>::= <algebraic variable>|<formula designator>|
             <sum>|<product>|<derivative>
<algebraic variable>::= <variable>
<formula designator>::= <variable>
<sum>::= S(<formula>, <formula>)
<product>::= P(<formula>, <formula>)
<derivative>::= DER(<formula>, <algebraic variable>)
```

In this definition, which is only valid for this chapter, the metalinguistic variables which are not defined are defined in the ALGOL 60 report [11].

Some examples of formulae are:	which read in ordinary notation:
x	x
a[7]	a_7

S(DER(a,x),DER(b,x)) $\partial a/\partial x + \partial b/\partial x$

P(x,y) $x \times y$

A typical assignment statement in which formulae occur is for example:

```
derivative:= if f = x then one else
             if type = sum then S(DER(a,x),DER(b,x)) else
             if type = product then
                S(P(a,DER(b,x)),P(DER(a,x),b)) else
             zero
```

Here, *derivative*, *f*, *a* and *b* are formula designators; *x*, *one* and *zero* are algebraic variables.

We now give a complete program in which the procedures constituting the simple system are declared.

```
begin comment SIMPLE FORMULA MANIPULATION SYSTEM
  R 1050 RPR 180766/01;
  integer one,zero,sum,product,algebraic variable,k;
  integer array F[1:1000,1:3];

  integer procedure STORE(lhs,type,rhs); value lhs,type,rhs;
  integer lhs,type,rhs;
  begin STORE:= k:= k + 1; F[k,1]:= lhs; F[k,2]:= type; F[k,3]:= rhs end;

  integer procedure TYPE(f,lhs,rhs); value f; integer f,lhs,rhs;
  begin lhs:= F[f,1]; TYPE:= F[f,2]; rhs:= F[f,3] end;

  integer procedure S(a,b); value a,b; integer a,b;
  S:= if a = zero then b else if b = zero then a else STORE(a,sum,b);

  integer procedure P(a,b); value a,b; integer a,b;
  P:= if a = zero ∨ b = zero then zero else
       if a = one then b else if b = one then a else
       STORE(a,product,b);

  integer procedure DER(f,x); value f,x; integer f,x;
  begin integer a,type,b; type:= TYPE(f,a,b);
```

```
    DER:= if f = x then one else
           if type = sum then S(DER(a,x),DER(b,x)) else
           if type = product then S(P(a,DER(b,x)),P(DER(a,x),b)) else
           zero
  end DER;

INITIALIZE: sum:= 1; product:= 2; algebraic variable:= 3; k:= 0;
  one:= STORE(0,algebraic variable,0);
  zero:= STORE(0,algebraic variable,0);

ACTUAL PROGRAM:
  begin integer f,x,y;
    procedure PR(s); string s;
    comment PR prints the string s without the string quotes ‹ and › ;
    PRINTTEXT(s);
    procedure OUTPUT(f); value f; integer f;
    begin integer a,type,b; type:= TYPE(f,a,b);
      if f = one then PR(‹1›) else
      if f = zero then PR(‹0›) else
      if f = x then PR(‹x›) else
      if f = y then PR(‹y›) else
      begin PR(‹(›); OUTPUT(a);
        if type = sum then PR(‹+›) else
        if type = product then PR(‹×›);
        OUTPUT(b); PR(‹)›)
    end end OUTPUT;

    x:= STORE(0,algebraic variable,0);
    y:= STORE(0,algebraic variable,0); NLCR;
    f:= S(x,y); OUTPUT(f); NLCR;
    f:= P(x,y); OUTPUT(f); NLCR;
    f:= P(S(x,y),S(x,y)); OUTPUT(f); NLCR;
    f:= DER(f,x); OUTPUT(f); NLCR;
    f:= DER(f,y); OUTPUT(f); NLCR
end end
```

The program consists of a standard part and an ad hoc part. The outer block, containing a number of procedure declarations and some statements labelled *INITIALIZE*, is standard and forms the simple system. The inner block, labelled *ACTUAL PROGRAM*, is ad hoc and defines the specific formula manipulations to be performed.

The variables of type integer: *one*, *zero*, *sum*, *product*, *algebraic variable* and k, declared in the heading of the program, get values after the label *INITIALIZE*.

The variable k is used as a pointer in the array $F[1:1000, 1:3]$, in which the formulae are internally represented.

As will be seen in the sequel, all algebraic variables and all formula designators occur in the program as variables of type integer.
In executing the program, they become equal to integers which define the location in F where the internal representation of the corresponding formulae are stored.

The effect of executing the following statements will now be examined:

```
 one:= STORE(0,algebraic variable,0);
zero:= STORE(0,algebraic variable,0)
```

The procedure *STORE* augments k by 1 (k was originally 0) and stores the three values of its three parameters: *lhs*, *type* and *rhs* into $F[k,1]$, $F[k,2]$ and $F[k,3]$, respectively; moreover, *STORE* itself becomes equal to k.
Thus, after executing the above statements, *one* = 1 and *zero* = 2.
Furthermore, $F[1,1] = 0$, $F[1,2] = 3$, $F[1,3] = 0$,
$F[2,1] = 0$, $F[2,2] = 3$, $F[2,3] = 0$.

In the sequel this is abbreviated to:

one = 1(0,3,0),
zero = 2(0,3,0).

The first statements of the actual program are:

```
x:= STORE(0,algebraic variable,0);
y:= STORE(0,algebraic variable,0),
```

whereupon x and y get the values 3(0,3,0) and 4(0,3,0),respectively.

The effect of the statement *NLCR* is to give the printer a *New Line Carriage Return* command.

The next statement is $f:= S(x,y)$ and f gets the value 5(3,1,4), since neither x nor y are equal to the variable *zero*. The effect of the statement *OUTPUT(f)* is the printing of the symbols "$(x+y)$".
This can be seen as follows:
By means of the procedure *TYPE*, which is the counterpart of the procedure *STORE*, the variables a, *type*, and b of *OUTPUT* get the values 3, 1 and 4,respectively.
Since f is not equal to the variables *one*, *zero*, x or y, the symbol "(" is printed.
A call of *OUTPUT(3)* has the effect of printing the symbol "x".
Since *type* = *sum* the symbol "+" is printed.
The symbol "y" is printed after a call of *OUTPUT(4)*, and finally the symbol ")" is printed.

The effect of the statements $f:= P(x,y)$ and *OUTPUT(f)* is: f = 6(3,2,4) and the symbols "$(x\times y)$" are printed.

A more complicated statement is $f:= P(S(x,y),S(x,y))$.
The parameters a and b of the procedure P are called by value, whence they are calculated beforehand; a gets the value 7(3,1,4) and b the value 8(3,1,4).
Since neither a nor b are equal to *one* or *zero*, P and thus f gets the value 9(7,2,8).
The procedure *OUTPUT* will now print the symbols "$((x+y)\times(x+y))$".

The statement $f:= DER(f,x)$ is discussed now and the roles of the algebraic variables *one* and *zero* will become apparent.

The variables a, *type* and b of *DER* get the values 7, 2 and 8,respectively. *DER* becomes equal to $S(P(7,DER(8,x)),P(DER(7,x),8))$.
First,$P(7,DER(8,x))$ will be calculated and this in turn activates the calculation of $DER(8,x)$.

In the calculation of *DER(8,x),a, type* and *b* of *DER* become equal to 3, *sum* and 4,respectively. *DER* becomes equal to *S(DER(3,x),DER(4,x))*. *DER(3,x)* gets the value *one* and *DER(4,x)* the value *zero*. *S(one,zero)* becomes equal to *one*, and thus *DER(8,x)* = *one*. *P(7,DER(8,x))* becomes equal to 7, since the parameter *b* of *P* is equal to *one*.
In the same way *P(DER(7,x),8)* gets the value 8.
Finally,*DER(f,x)* and thus *f* gets the value 10(7,*sum*,8).
Next, *OUTPUT* prints the symbols "*((x+y)+(x+y))*".

It is left to the reader to verify that the effect of the last statements:

f:= DER(f,y) and *OUTPUT(f)*

is that *f* gets the value 11(1,1,1) and that the symbols "*(1+1)*" are printed.

This chapter is closed with the following remarks:

1. As shown above, the computer handles a formula by means of a number which defines the location of the internal representation of this formula in the array *F*. The actions of the procedures *STORE*, *S*, *P* and *DER* consist primarily of side-effects, namely storing formulae; whereas the calculated values of the procedure identifiers are only important within the computer. The programmer is interested in the form of a formula such as it is stored in *F*, but not in the actual value of the corresponding formula designator.
   In the discussion of the next sections, therefore, usually reference is made to the formula defined by a certain formula designator, rather than to the numerical value of this formula designator.
   In notation this has the consequence that, for example, the following sentence:

   "The value of the formula designator defining the formula "f" becomes equal to the value of the formula designator defining the formula "g".

   is abbreviated to:

"The formula "f" becomes equal to the formula "g""

or more simply:""f" becomes equal to "g"".
If, however, we want to refer to the numerical value of the formula designator "f", rather than to the formula defined by it, then we use the notation "f*".

2. For the internal representation of algebraic variables, three array elements of *F* were used, whereas one array element, *F*[*k*,*2*], was effectively used.
   The other array elements may be used, however, to store more information of the algebraic variables, which will be done in the general system.
   If there occur a large number of algebraic variables then one may also specify an algebraic variable as having, in the computer, a negative value in contrast to all other formulae. Then the procedure *TYPE* has to be changed appropriately.

3. The array *F* was used to store formulae; evidently there will be difficulties if there are more than 1000 formulae to be stored.
   In the general system of chapter 2, the user may choose the length of *F*, whilst he is also provided with means to erase noninteresting formulae. Moreover, only 7/6 machine word is used to store the three characterizing quantities of a formula.

4. If the user wants a system which applies the distributive law to formulae, he may use, instead of the already declared procedure *P*, the following procedure:

```
integer procedure P(a,b); value a,b; integer a,b;
begin integer ta,tb,la,lb,ra,rb;
      ta:= TYPE(a,la,ra); tb:= TYPE(b,lb,rb);
      P:= if a = zero ∨ b = zero then zero else
          if a = one then b else
          if b = one then a else
```

```
    if ta = sum then S(P(la,b),P(ra,b)) else
    if tb = sum then S(P(a,lb),P(a,rb)) else
    STORE(a,product,b)
end;
```

Using this procedure, all formulae are stored internally in expanded form.

Chapter 2

THE GENERAL FORMULA MANIPULATION SYSTEM

2.1. The form of formulae in the general system

The formulae used in the general system which occur in an ALGOL 60 program should have the form syntactically defined below, in which metalinguistic variables used but not defined are defined in the ALGOL 60 report [11].

<formula>::= *one*|*zero*|*minone*|*im unit*|<algebraic variable>|
<formula designator>|<sum>|<difference>|<product>|<quotient>|
<power>|<integral power>|<number>|<truncated power-series>|
<function>|<extended sum>|<derivative>|<simplified formula>|
<complex conjugate>|<result of substitution>|
<integral quotient>|<common divisor>

<algebraic variable>::= <variable>

<formula designator>::= <variable>

<sum>::= *S(*<formula>,<formula>*)*

<difference>::= *D(*<formula>, <formula>*)*

<product>::= *P(*<formula>,<formula>*)*

<quotient>::= *Q(*<formula>, <formula>*)*

<power>::= *POWER(*<formula>, <formula>*)*

<integral power>::= *INT POW(*<formula>, <integral-arithmetic expression>*)*

<integral-arithmetic expression>::= <arithmetic expression>

<number>::= <integral number>|<real number>|<complex number>

<integral number>::= *IN(*<integral-arithmetic expression>*)*

<real number>::= *RN(*<arithmetic expression>*)*

<complex number>::= *CN(*<arithmetic expression>, <arithmetic expression>*)*

<truncated power-series>::= *TPS(*<integral variable>, <degree>,<algebraic variable>,<coefficient depending on integral variable>*)*

<integral variable>::= <variable>

```
<degree>::= <integral-arithmetic expression>
<coefficient depending on integral variable>::= <formula>
<function>::= <special-function designator>
<special-function designator>::= <special-function identifier>(<formula>)
<special-function identifier>::= EXP|LN|SIN|COS|ARCTAN|SQRT
<extended sum>::= Sum(<integral variable>,<integral-arithmetic
      expression>,<integral-arithmetic expression>,<formula
      depending on integral variable>)
<formula depending on integral variable>::= <formula>
<derivative>::= DER(<formula>,<algebraic variable>)
<simplified formula>::= SIMPLIFY(<formula>)
<complex conjugate>::= CC(<formula>)
<result of substitution>::= SUBSTITUTE(<formula>,<integral variable>,
     <integral-arithmetic expression>,<integral-arithmetic expression>,
     <formula depending on integral variable>,<formula depending on
      integral variable>)
<integral quotient>::= QUOTIENT(<formula>,<formula>,<remainder>)
<remainder>::= <formula designator>
<common divisor>::= COMMON DIVISOR(<formula>,<formula>)
```

The following remarks are made:

1. An integral-arithmetic expression is an arithmetic expression of type integer.

2. "degree" is an arithmetic expression taking nonnegative integral values only.

3. Examples of "coefficient depending on integral variable" and "formula depending on integer variable" are *RN(1/i)* or *a*[*i*], where *a*[*i*] is a formula designator.

4. In the sequel, formulae will not only be written in the text as defined above, but also in ordinary notation.

5. The symbols *one*, *zero*, *minone* and *im unit* are identifiers of variables of type integer and the symbols S, D, P, etc. are identifiers of procedures of type integer in the general system. This means that a formula occurring in an ALGOL 60 program is an integral-arithmetic expression.

   Examples of formulae are:

   S(one,x)
   S(INT POW(SIN(x),2), INT POW(COS(y),2))
   Q(CN(1,1), CN(1,-1))
   TPS(i,n,x, if i = 0 then zero else RN(1/i))
   SUBSTITUTE(f,i,1,2,a[i],RN(1/i))
   COMMON DIVISOR(Sum(i,0,3,INT POW(x,3-i)), D(P(x,x),one))

   which have in ordinary notation the meaning:

   $1+x$
   $\sin^2 x + \cos^2 y$
   (1+i)/(1-i) ("i" is the imaginary unit)
   $x + x^2/2 + \ldots + x^n/n + O(x^{n+1})$
   if the variables $a[1]$ and $a[2]$ occur in f, then they are replaced by 1 and $\frac{1}{2}$, respectively.

   A common divisor of $x^3 + x^2 + x + 1$ and $x^2 - 1$ is $x + 1$.

2.2. How to use the general system

There are two ways to use the general system.
If the tools of the system are sufficient, one may put the formulae on input tape and use the program as reproduced in chapter 3.

The other way of using the general system is described in the remainder of this section; it amounts to:

1. Copy the general system as reproduced in this chapter.
2. Write a program which defines the operations on formulae to be performed; this program will henceforth be called "actual program".
3. Combine the general system and the actual program, preceded by the statement: *INITIALIZE* and closed by two extra *end*s, into one program.
4. Prepare an input tape containing 7 numbers defining upper bounds for:
   a. the number of formulae to be used,
   b. the number of real numbers to be used,
   c. the number of complex numbers to be used,
   d. the number of truncated power-series to be used,
   e. the maximal degree of the truncated power-series to be used,
   and defining:
   f. the absolute accuracy, with which the system has to perform numerical computations,
   g. the relative accuracy, with which the system has to perform numerical computations.

In most cases it is desirable to output formulae. One may then copy the declaration of the procedure *OUTPUT* (section 2.13), in the actual program.
In the procedure *OUTPUT* a call of *OUTPUT VARIABLE* is made; the procedure *OUTPUT VARIABLE* should, therefore, also be declared in the actual program, which defines the output of the algebraic variables.
The algebraic variables and formula designators, used in the actual program, should be declared in the heading of the actual program as variables of type integer or as integer-array elements. Before the algebraic variables are used, they should have got a value by means of a statement of the form:
<algebraic variable>*:= STORE(*<integral-arithmetic expression>,
algebraic variable, <integral-arithmetic expression>*)*

The easiest way to perform these actions is to write the procedure *OUTPUT VARIABLE* in such a way that it combines both the outputting and the initializing instructions.
An example of a possible declaration of *OUTPUT VARIABLE* may be found in section 2.13.

2.3. The internal representation of formulae in the system

The general system is constructed in such a way that *one*, *zero*, *minone* and *im unit* are represented as formulae of the form number (in contrast to their representation in the simple system); moreover, an extended sum, a derivative, a simplified formula, a complex conjugate, a result of a substitution, an integral quotient and a common divisor are worked out; i.e. the "operations" *Sum*, *DER*, *SIMPLIFY*, *CC*, *SUBSTITUTE*, *QUOTIENT* and *COMMON DIVISOR* are applied to the formulae occurring as parameters, which results in other formulae. This means that each formula "f" can be characterized by three quantities which will be called: lhs, type and rhs.

The quantity type may be: *sum*, *difference*, *product*, *quotient*, *power*, *integral power*, *number*, *tr pow series*, *function* or *algebraic variable*. In the first five cases, lhs and rhs denote the left-hand-side- and the right-hand-side formulae by means of which "f" is built up.
The meaning of lhs and rhs in the other cases is summarized in the following table:

type	lhs	rhs
integral power	exponent of "f"	base of "f"
number	*integer*, *real* or *complex*	location where the value of the number can be found
tr pow series	degree of truncated power-series	location where the argument and the coefficients can be found
function	*expf*, *lnf*, *sinf*, *cosf*, *arctanf* or *sqrtf*	argument of function
algebraic variable	may be used to define extra information	

2.4. The heading of the general system

The heading of the general system runs as follows:

```
begin comment GENERAL SYSTEM for FORMULA MANIPULATION
  second corrected version R1050 RPR 280267/04/02;
  integer sum,difference,product,quotient,power,integral power,
  number,tr pow series,algebraic variable,function,
  integer,real,complex,expf,lnf,sinf,cosf,arctanf,sqrtf,
  one,zero,minone,im unit,di,k,krn,kcn,ktps,index,
  Fm,FRNm,FCNm,FTPSm1,FTPSm2,t13,kt1,kt2,kt3,ft;
  real null,rel acc; Boolean expand; integer array FDRUM[1:100,1:2],
  TPSDRUM[1:100],INDEX[0:25,1:4],kdr,k1[1:4],p16[0:5];
  array RNDRUM[1:25],CNDRUM[1:25,1:2];
  Fm:= read; FRNm:= read; FCNm:= read; FTPSm1:= read;
  FTPSm2:= read; null:= abs(read); rel acc:= abs(read); t13:= 2 ⋀ 13;
  p16[0]:= 1; for di:= 1,2,3,4,5 do p16[di]:= p16[di-1]×16;
  begin integer array F[0:Fm],FT[0:Fm: 6],FTPS[0:FTPSm1,-1:FTPSm2];
    real array FRN[0:FRNm],FCN1,FCN2[0:FCNm];
```

By means of a call of the procedure *INITIALIZE*, see section 2.13, all the variables, declared above, get initial values. These variables are used throughout the following sections and chapters 3 and 5. Some variables, not self-explanatory by their identifiers, need some attention:

di is a dummy variable to be used as a necessary but noninteresting procedure parameter;

k, *krn*, *kcn* and *ktps* are the pointers of the arrays *F*, *FRN*, *FCN1*, *FCN2* and *FTPS* in which formulae, real numbers, complex numbers and coefficients of truncated power-series are stored;

Fm, *FRNm*, *FCNm*, *FTPSm1* and *FTPSm2* are read from input tape to define the bounds of the arrays *F*, *FRN*, *FCN1*, *FCN2* and *FTPS*;

index is the pointer of the array *INDEX*; this array is used for the book-keeping necessary for erasing formulae;

t13 gets the value 2^{13} and is used to store formulae compactly.

kt1, *kt2*, *kt3* and *ft* are auxiliary variables used for storing formulae.

null and *rel acc*, both read from input tape, are used as the absolute and relative accuracy,respectively,for transforming complex numbers into real numbers and for transforming real numbers into integral numbers;

expand is a Boolean variable defining the state of the general system. If *expand*, all formulae are stored in expanded form, i.e. the distributive law is applied for cancelling brackets around sums; square-root functions and trigonometric-functions are transformed into complex-exponential functions; "f - g" is transformed into "f + (-1) × g"; integral powers are expanded, if possible; and "f ↑ g" is transformed into "exp(g×ln(f))". If not *expand*, all formulae are stored as they are written except for some trivial simplifications (a + 0 → a, a × 0 → 0, number + number → number, etc.).

The arrays *FDRUM*, *RNDRUM*, *CNDRUM*, *TPSDRUM*, *kdr* and *k1* are used in a process, called the "erase but retain process" for erasing some formulae but retaining others by means of secondary storage on the drum (see section 2.7).

The array elements p16[0], ..., p16[5] are equal to 1, ..., 16^5.

2.5. The internal representation of formulae

In order to be able to do formula manipulations on a small computer, it is necessary that the formulae are stored as compactly as possible.
The Electrologica X8 computer of the Mathematical Centre has 48 K machine words in core memory and each machine word has 28 bits, including the parity and sign bits. The "formula-program processor" of chapter 3 needs about 16 K machine words, whilst the ALGOL 60 system needs about 14 K machine words; there remain therefore about 18 K machine words.
Since a considerable amount of storage space is used by the formula-manipulation system for recursive procedure calls and for arrays, in which the formulae are stored during the simplifying processes described in sections 2.10 and 2.11, it was decided to design the internal representation for a number of formulae not exceeding 2^{13}.
This leaves also space for storing real numbers, complex numbers and truncated power-series.
A formula is characterized by its type and by the quantities lhs and rhs. Since lhs and rhs, in case of e.g. a sum, can not be larger than $2^{13} - 1$ (a formula can not refer to itself), both quantities can be stored into one machine word, for which we take the integer array element *F*[*k*], *k* running from 0 to *Fm* (for obvious reasons we do not use the parity- and sign bits). The quantity type is stored into 4 bits of another machine word for which the array element *FT*[*kt1*] is taken. If the lhs and rhs quantities of the *k*-th formula are stored into *F*[*k*], then the quantity type is stored into *FT*[*k* ÷ *6*].
The procedures as reproduced in this section are used in section 2.6.
ST TYPE is used to store the type of a formula; note that the array elements of *FT* get the initial value 0 by a call of *INITIALIZE*.
augment is used to augment one of the five pointers *k*, *krn*, *kcn*, *ktps* and *index*; when the pointer is augmented too much, then this is signalled and the execution of the program is discontinued by a call of the MC procedure *EXIT*.
Note that in the "formula-program processor" of chapter 3 a special procedure *EXIT* is declared, which has almost the same effect as the MC

procedure *EXIT*: the execution of the "formula program" is discontinued but the "formula-program processor" continues in treating a next "formula program", if available.

```
procedure ST TYPE(k,t); value k,t; integer k,t;
begin kt1:= k:6; kt2:= k - kt1×6; kt3:= p16[kt2]; ft:= FT[kt1];
  FT[kt1]:= ft + (t - ft:kt3 + ft:(kt3×16)×16)×kt3
end ST TYPE;

procedure augment(p,pm); integer p,pm;
begin p:= p + 1; if p > pm then
  begin NLCR; PRINTTEXT(≮array too small≯); ABSFIXT(4,0,k);
    PRINTTEXT(≮  ≯); ABSFIXT(4,0,krn); PRINTTEXT(≮  ≯);
    ABSFIXT(4,0,kcn); PRINTTEXT(≮  ≯); ABSFIXT(4,0,ktps);
    PRINTTEXT(≮  ≯); ABSFIXT(4,0,index); EXIT
end end augment;
```

2.6. The basic procedures

The basic procedures, which perform all the operations involving the internal representation of the formulae and which are used very extensively in the programs of this chapter and of chapters 3 and 5, will be described now.
If the internal representation of the formulae is changed, e.g. new core memory becomes available, then the basic procedures and the procedures of sections 2.5 and 2.7 should also be changed but the remainder of the system may remain unaltered.

```
procedure INITIALIZE;
begin expand:= true; krn:= ktps:= index:= -1;
  number:= 1;              tr pow series:= 2;        function:= 3;
  algebraic variable:= 4; sum:= 5;                   difference:= 6;
  product:= 7;             quotient:= 8;             power:= 9;
  integral power:= 10;
  integer:= 1;             real:= 2;                 complex:= 3;
  expf:= 1;                lnf:= 2;                  sinf:= 3;
  cosf:= 4;                arctanf:= 5;              sqrtf:= 6;
  for k:= 0 step 1 until Fm: 6 do FT[k]:= 0; k:= -1;
  zero:= STORE(integer,number,0);
  one:= STORE(integer,number,2);
  minone:= STORE(integer,number,3);
  kcn:= 0; FCN1[0]:= 0; FCN2[0]:= 1;
  im unit:= STORE(complex,number,0);
  FIX
end INITIALIZE;

integer procedure STORE(lhs,type,rhs); value lhs,type,rhs;
integer lhs,type,rhs;
begin augment(k,Fm); F[k]:= lhs + rhs×t13; ST TYPE(k,type);
  STORE:= k
end STORE;
```

```
integer procedure IN(i); value i; integer i;
IN:= if i = 1 then one else
     if i = 0 then zero else
     if i = -1 then minone else
     if abs(i) > 4095 then RN(i) else
     STORE(integer,number,
        (if i < 0 then 1 else 0) + 2 × abs(i));

integer procedure RN(r); value r; real r;
if abs(r) ≤ null then RN:= zero else
if abs(r - entier(r+.5)) ≤ rel acc × abs(r) + null ∧
  abs(r) ≤ 4095 then
RN:= IN(entier(r+.5)) else
begin augment(krn,FRNm); FRN[krn]:= r;
  RN:= STORE(real,number,krn)
end RN;

integer procedure CN(r,i); value r,i; real r,i;
if abs(i) ≤ null then CN:= RN(r) else
if abs(r) ≤ null ∧ abs(i - 1) ≤ rel acc then CN:= im unit else
begin augment(kcn,FCNm);
  FCN1[kcn]:= r; FCN2[kcn]:= i;
  CN:= STORE(complex,number,kcn)
end CN;

integer procedure TPS(i,degree,x,coeffi); value degree,x;
integer i,degree,x,coeffi;
begin if degree < 0 ∨ degree > FTPSm2 then
  begin NLCR; PRINTTEXT(⟨degree of TPS not appropriate⟩); EXIT end;
  begin integer array c[-1:degree]; Boolean B; B:= true;
    c[-1]:= x; i:= 0; x:= c[0]:= coeffi;
```

```
    for i:= 1 step 1 until degree do
    begin c[i]:= coeffi; B:= B ∧ c[i] = zero end;
    if B ∧ TYPE(x,di,di) ≠ tr pow series then TPS:= x else
    begin augment(ktps,FTPSm1); for x:= -1 step 1 until degree do
      FTPS[ktps,x]:= c[x]; TPS:= STORE(degree,tr pow series,ktps)
end end end TPS;

integer procedure TYPE(f,lhs,rhs); value f;
integer f,lhs,rhs;
begin kt1:= f: 6; kt2:= f - kt1×6; kt3:= p16[kt2]; ft:= FT[kt1];
  TYPE:= ft: kt3 - ft: (kt3×16)×16; ft:= F[f]; rhs:= ft: t13;
  lhs:= ft - rhs×t13
end TYPE;

procedure VAL OF INT NUM(f,i); value f; integer f,i;
begin i:= F[f]: t13; i:= EVEN(i)×(abs(i): 2);
  comment EVEN is the MC procedure with:
    EVEN(i) = (-1)↑i;
end VAL OF INT NUM;

procedure VAL OF REAL NUM(f,r); value f;
integer f; real r;
r:= FRN[F[f]: t13];

procedure VAL OF COMP NUM(f,r,i); value f;
integer f; real r,i;
begin di:= F[f]: t13; r:= FCN1[di]; i:= FCN2[di] end;

procedure COEFFICIENT(f,degree,x,coeff); value f,degree;
integer f,degree,x; integer array coeff;
begin di:= F[f]: t13;
```

```
  for x:= 0 step 1 until degree do coeff[x]:= FTPS[di,x];
  x:= FTPS[di,-1]
end COEFFICIENT;

procedure REPLACE(f,g); value f,g; integer f,g;
begin F[f]:= F[g]; ST TYPE(f,TYPE(g,di,di)) end;

procedure FIX;
begin augment(index,25); INDEX[index,1]:= k; INDEX[index,2]:= krn;
  INDEX[index,3]:= kcn; INDEX[index,4]:= ktps
end FIX;

procedure ERASE;
begin k:= INDEX[index,1]; krn:= INDEX[index,2];
  kcn:= INDEX[index,3]; ktps:= INDEX[index,4]; index:= index - 1;
  if index = -1 then begin PRINTTEXT(⊰
FIX missing⊱); EXIT end
end ERASE;

procedure LOWER INDEX;
begin index:= index - 1; if index = -1 then
  begin PRINTTEXT(⊰
FIX missing⊱); EXIT
  end
end LOWER INDEX;

Boolean procedure FIXED(f); value f; integer f;
FIXED:= f ≤ INDEX[index,1];

procedure ERASE BUT RETAIN(i,lb,ub,fi);
begin STORE TO DRUM(i,lb,ub,fi,FIXED); LOWER INDEX end;
```

A call of the procedure *INITIALIZE* is necessary before any action of the program. The formulae: *one*, *zero*, *minone* and *im unit* are stored. It is remarked that the more simple statements: *one:= IN(1); zero:= IN(0); minone:= IN(-1); im unit:= CN(0,1);* substitúted into this procedure would lead to erroneous results.

The procedure *STORE* stores a triple: *lhs*, *type* and *rhs* defining a formula.

A call: *f:= IN(i)* where $|i| \leq 4095$ has as effect that *f* becomes an integral number; if $i = 1$ then *f* becomes *one*, if $i = -1$ then *f* becomes *minone*, if $i = 0$ then *f* becomes *zero*.
If $|i| > 4095$ then *f* becomes a real number.

A call: *f:= RN(r)* has as effect that *f* becomes an integral number if *r* is close (within the relative precision *rel acc*) to an integer and does not exceed 4095, otherwise *f* becomes a real number.

A call: *f:= CN(r,i)* has as effect that, if $|i| \leq$ *null*, then *f* becomes a real number (possibly an integral number), otherwise, *f* becomes a complex number.

It is remarked that, if *null* and *rel acc* are chosen equal to 10^{-10}, the calculation:

f:= EXP(CN(0,2×3.14159265359)),

where *EXP* stores the exponential function, results in the formula *one*.

A call: *f:= TPS(i,5,x,c[i])* has as effect that *f* becomes a truncated power-series in the variable *x* of the fifth degree and with coefficients *c*[*i*], i = 0, 1, 2, 3, 4 and 5 (we use "Jensen's device").

The next five procedures are the counterparts of the last five procedures discussed above; they define information for a given formula *f*.

A call: *t:= TYPE(f,a,b)*, which occurs time and again in the general system, has as effect that *t* becomes equal to the type of *f*, *a* becomes equal to the lhs of *f* and *b* becomes equal to the rhs of *f*.

We will elucidate the use of *FIX*, *ERASE*, *LOWER INDEX* and *ERASE BUT RETAIN* by means of some semantical rules.

Two sequences, S_1 and S_2, of ALGOL 60 statements, are equivalent, denoted by "~", if they have, on execution, the same effect on the formation of formulae.

Let *S1*, *S2* and *S3* be ALGOL 60 statements which store the formulae *f*[*1*], *f*[*2*] and *f*[*3*],respectively. This should be understood in a general sense: *S1* may stand for a set of statements and *f*[*1*] may stand for a set of formulae.

The effect of the procedures *FIX*, *ERASE*, *LOWER INDEX* and *ERASE BUT RETAIN* may now be defined as follows:

```
FIX; S1; ERASE; ~ ;
FIX; S1; LOWER INDEX; ~ S1;
FIX; S1; S2; S3; ERASE BUT RETAIN(i,1,1,f[i]); ~ S1;
FIX; S1; S2; S3; ERASE BUT RETAIN(i,1,2,f[i]); ~ S1; S2;
FIX; S1; S2; S3; ERASE BUT RETAIN(i,1,3,f[i]); ~ S1; S2; S3;
FIX; S1; S2; S3; g[1]:= f[1]; g[2]:= f[3];
      ERASE BUT RETAIN(i,1,2,g[i]); f[1]:= g[1]; f[3]:= g[2]; ~ S1; S3;
```

<u>Remarks</u>:

1. The above rules for using *ERASE BUT RETAIN* are not exhaustive; they only indicate by means of the special example of three formulae *f*[*i*] how *ERASE BUT RETAIN* should be used in the general case.
2. It is not necessary that the sequence of statements, e.g. *FIX; S1; ERASE;* should actually occur in this way in the program, it is only important that during the running of the program these three statements are executed consecutively.

We shall apply the above rules to some examples:

```
FIX; S1; FIX; S2; ERASE; S3; ERASE; ~ ;
FIX; S1; FIX; S2; ERASE; S3; LOWER INDEX ~ S1; S3;
FIX; S1; FIX; S2; LOWER INDEX; S3; ERASE BUT RETAIN(i,2,2,f[i]) ~ S2;
FIX; S1; FIX; S2; ERASE BUT RETAIN(i,2,2,f[i]); S3; LOWER INDEX;
                                                        ~ S1; S2; S3;
```

A call: *VAL OF INT NUM(f,a)* results, only in case f is an integral number, in the assignment to a of the value of that integral number; thus *VAL OF INT NUM(IN(5),a)* results in: $a = 5$.

The procedures *VAL OF REAL NUM* and *VAL OF COMP NUM* are similar to *VAL OF INT NUM*.

A call: *COEFFICIENT(f,d,x,c)* results, only in case f is a truncated power-series

$$f = \sum_{i=0}^{n} c_i \ y^i$$

and $n \geq d$ in: $x = y$ and $c[i] = c_i$, $i = 0, 1, \ldots, d$.

A call: *REPLACE(f,g)* results in a change of the internal representation of the formula f in such a way that the formula f becomes the same as the formula g. The location where f is stored, remains unchanged.

Example: The effect of the statements:

f:= IN(2); g:= f; REPLACE(f,IN(3));

is that both g and f become formula "3"; thus the change of f has as side-effect a change of g. On the other hand, the effect of the statements:

f:= IN(2); g:= f; f:= IN(3);

is that f becomes formula "3" and g becomes formula "2"; thus the change of f does not involve a change of g.

The procedure *FIXED* is described in section 2.7. The procedures *FIX*, *ERASE*, *LOWER INDEX* and *ERASE BUT RETAIN* belong together.
A call of each of the three last procedures should be preceded by a call of *FIX*. Formulae stored after this call are:
erased by a call of *ERASE*,
erased except the formulae given by *fi*, *i* = *lb*(1)*ub*, by a call of *ERASE BUT RETAIN(i,lb,ub,fi)* and
retained by a call of *LOWER INDEX*.

In most other formula manipulation systems use is made of "garbage collection" instead, to get rid of uninteresting formulae and to obtain new memory space.
In section 2.14 we shall investigate the consequences of such a garbage collection for the construction of ALGOL 60 formula-manipulation programs.

2.7. The "erase but retain process"

In a program for formula manipulation, one is faced with the following problem:

> In the course of the computations there have been built up certain formulae, forming the set E, which have rendered good service, but which have become noninteresting; other formulae, forming the set R, built up simultaneously with the formulae of E, are still of interest. How should the formulae of E be erased whilst the formulae of R be retained,in order to create memory space for storing new formulae.

We first suppose that R is empty, then the formulae of E can be erased very simply by the following "erase process":

Step 1: Just prior to the storage of the first formula of E the values of the pointers of the arrays *F*, *FT*, *FRN*, *FCN1*, *FCN2* and *FTPS* are stored into the array *INDEX* (with pointer *index)*; this step is performed by a call of the procedure *FIX*;

Step 2: The computations are performed and the formulae of E are built up;

Step 3: The pointers of the arrays *F*, ... get the values of the pointers previously stored into *INDEX* by step 1; it is the task of the procedure *ERASE* to perform this step.

We now suppose that R is not empty and we use the following more complicated "erase but retain process":

Step a: The same as step 1 above ("E" should be replaced by "E and R");

Step b: The same as step 2 above ("E" should be replaced by "E and R");

Step c: The formulae of R are stored onto the drum;

Step d: The same as step 3;

Step e: The formulae stored on the drum are restored into the arrays *F*, ... and the pointers of *F*, ... are, of course, changed appropriately.

It is the task of the procedure *STORE TO DRUM* to execute steps c, d and e.

The formulae of R are given to the procedure *STORE TO DRUM* as the *fi*, where $i = lb(1)ub$ (we use "Jensen's device").
Consider a certain formula "f" of R. "f" is built up by means of subformulae. Of these subformulae some have been stored already before Step a has been executed which means that they are not affected by the execution of Step d; other subformulae are stored after Step a has been executed and these subformulae should be stored together with "f" itself on the drum. From this investigation it follows that we have to provide the means, in the form of the Boolean procedure *BOOL*, such that *STORE TO DRUM* can decide whether a certain subformula of "f" should be stored on the drum or not. As actual parameter for *BOOL*, the Boolean procedure *FIXED*, of section 2.6 may be chosen. The procedure *FIXED* gets the value *true* or *false* dependent on whether "f" (see remark 1 of chapter 1) is ≤ or > than the pointer *k* of the array *F* lastly stored into *INDEX*. This means that the formula "f" has been stored before or after the execution of Step a.
However, we may also choose as actual parameter for *BOOL* a procedure which does more than *FIXED*, e.g., the procedure *RET VAR* declared in the "formula program processor" of chapter 3. This procedure has the same task as *FIXED*, but it takes, moreover, care of the administration of algebraic variables. This is necessary in order to make the connection between the *identifier list* and the, in *F*, stored algebraic variable. It is for this connection that the procedure *RET VAR* has to know, during the "store to drum process", the value of the auxiliary pointer *k1[1]*; this value is passed to *RET VAR* by means of the integral variable *di*.

The principle of the procedure *STORE TO DRUM* is to store the formulae of R on the drum as if they were stored immediately in their future places (Step e) of the arrays *F*, ... In this way we are sure that the referencing of a formula to its subformulae remains correct. To do this we need 4 auxiliary pointers *k1[i]*, i = 1, 2, 3 and 4. These pointers get initially the values of *INDEX[index,i]*, i = 1, 2, 3 and 4, being the values of the pointers of the arrays *F*, ... stored into *INDEX* by Step a. After the "store to drum process" is executed, the values of the pointers

of *F*, ... are just the values of *k1*[*i*], *i* = 1, 2, 3 and 4. Note that the pointers *k1*[*i*] may be considered as pointers of "virtual" arrays *F*, ... Transferring number by number to the drum would be too time-consuming; therefore,we introduce the arrays *FDRUM*, *RNDRUM*, *CNDRUM* and *TPSDRUM*, with pointers *kdr*[*i*], *i* = 1, 2, 3 and 4, as intermediate storage. These arrays are filled during the "store to drum process".
If one of them has become full, its contents is, by means of a call of *TO DRUM*, sent to the drum as a block of data. The location on the drum where these data are stored is given to *TO DRUM* and is assigned to *FDRUM*[*1,1*], *RNDRUM*[*1*], *CNDRUM*[*1,1*] or *TPSDRUM*[*1*]. The integral variable *p* is used for the book-keeping of the data on the drum. If a certain array, say *FDRUM*, has become full for a second time, its contents is sent again to the drum and *FDRUM*[*1,1*] becomes equal to the number defining the place on the drum where this second block is stored.
This means that the drum may contain chains of blocks of data each block referring to a next one; the lastly stored block may be found using the value of *FDRUM*[*1,1*], ... or *TPSDRUM*[*1*].
This means also that the "restore from drum process", as executed by the statements after the label *STORE FROM DRUM* should be done in reverse order i.e. the formulae should be stored from "top to bottom" in the arrays *F*, ..., in contrast to the way formulae are normally stored into *F*, ...

Special attention needs the treatment of the coefficients of truncated power-series which is different from the treatment of formulae, real numbers and complex numbers, as a consequence of the fact that the degree of the truncated power-series is not fixed.
The coefficients and the variable "x" of a truncated power-series are stored linearly in *TPSDRUM*. Different truncated power-series are separated by an array element equal to *-1000*.

Copying the formulae to the drum is the task of the procedure *cod*, used recursively.

We now give the procedure *STORE TO DRUM*.

```
procedure STORE TO DRUM(i,lb,ub,fi,BOOL); value lb,ub;
  integer i,lb,ub,fi; Boolean procedure BOOL;
begin integer p;
  integer procedure cod(f); value f; integer f;
  begin integer t,a,b;
    procedure SOD(i,max,size,arr); integer i,max,size;
    comment The MC procedure TO DRUM may be used
      without declaration. It stores data given in the array,
      occurring as first parameter, on the drum. The begin-
      address of these data is given by the second parameter;
    begin kdr[i]:= kdr[i] + 1; if kdr[i] > max then
      begin kdr[i]:= 2; TO DRUM(arr,p); if i = 1 then FDRUM[1,1]:= p
        else if i = 2 then RNDRUM[1]:= p + .1 else if i = 3 then
        CNDRUM[1,1]:= p + .1 else TPSDRUM[1]:= p; p:= size + p + 1
    end end SOD;
    di:= k1[1]; if F[f] < 0 then begin cod:= - F[f] - 1; goto END end;
    if BOOL(f) then begin cod:= f; goto END end;
    t:= TYPE(f,a,b);
    if t = sum ∨ t = difference ∨ t = product ∨ t = quotient ∨
    t = power then begin a:= cod(a); b:= cod(b) end else
    if t = function ∨ t = integral power then b:= cod(b) else
    if t = number then
    begin if a = real then
      begin b:= k1[2]:= k1[2] + 1; SOD(2,25,50,RNDRUM);
        VAL OF REAL NUM(f,RNDRUM[kdr[2]])
      end else if a = complex then
      begin b:= k1[3]:= k1[3] + 1; SOD(3,25,100,CNDRUM);
        VAL OF COMP NUM(f,CNDRUM[kdr[3],1],CNDRUM[kdr[3],2])
      end
    end else if t = tr pow series then
    begin integer i,x; integer array c[0:a];
      COEFFICIENT(f,a,x,c); x:= cod(x); for i:= 0 step 1 until a do
      c[i]:= cod(c[i]); b:= k1[4]:= k1[4] + 1; for i:= a + 1 step -1 until -1 do
```

```
    begin SOD(4,100,100,TPSDRUM); TPSDRUM[kdr[4]]:=
      if i = -1 then x else if i = a + 1 then -1000 else c[i]
   end end;
   cod:= k1[1]:= k1[1] + 1; SOD(1,100,200,FDRUM);
   FDRUM[kdr[1],1]:= a + b × t13; FDRUM[kdr[1],2]:= t;
   F[f]:= - k1[1] - 1;
  END: end cod;
  for p:= 1,2,3,4 do begin k1[p]:= INDEX[index,p]; kdr[p]:= 1 end;
  p:= 1; for i:= lb step 1 until ub do fi:= cod(fi);

STORE FROM DRUM:
  begin integer i; procedure SFD(b,arr,pl);
   comment The MC procedure FROM DRUM may be used
     without declaration. It takes data from the drum with begin-address
     defined by its second parameter and stores the data into the
     array given as its first parameter;
   begin A: if k1[i] > INDEX[index,i] then
     begin if kdr[i] = 1 then
      begin kdr[i]:= b; FROM DRUM(arr,pl) end;
      if i = 1 then
      begin F[k1[i]]:= FDRUM[kdr[i],1]; ST TYPE(k1[i],FDRUM[kdr[i],2])
      end else
      if i = 2 then FRN[k1[i]]:= RNDRUM[kdr[i]] else
      begin FCN1[k1[i]]:= CNDRUM[kdr[i],1];
        FCN2[k1[i]]:= CNDRUM[kdr[i],2]
      end; k1[i]:= k1[i] - 1; kdr[i]:= kdr[i] - 1; goto A
   end end SFD;
   k:= k1[1]; krn:= k1[2]; kcn:= k1[3]; ktps:= k1[4];
   i:= 1; SFD(100,FDRUM,FDRUM[1,1]);
   i:= 2; SFD(25,RNDRUM,entier(RNDRUM[1]));
   i:= 3; SFD(25,CNDRUM,entier(CNDRUM[1,1]));
   i:= - 2;
```

```
  A4: if kdr[4] ≠ 1 then
    begin if TPSDRUM[kdr[4]] = -1000 then
      begin k1[4]:= k1[4] - 1; i:= - 2 end else
      begin i:= i + 1; FTPS[k1[4],i]:= TPSDRUM[kdr[4]] end;
      kdr[4]:= kdr[4] - 1; goto A4
    end else if k1[4] > INDEX[index,4] then
    begin FROM DRUM(TPSDRUM,TPSDRUM[1]);
      kdr[4]:= 100; goto A4
  end end
end STORE TO DRUM;
```

2.8. The procedures *S, D, P, Q, POWER* and *INT POW*

The procedures of the title of this section are used to store a sum, a difference, a product, a quotient, a power and an integral power. In the declaration of these procedures use is made of some auxiliary procedures: *OPER ON TPS*, *ARITHMETIC*, *Sum*, *COMB*, *REPEATED PRODUCT* and *comm div*.

First the procedure *OPER ON TPS* is given:

```
integer procedure OPER ON TPS(oper,a,ta,da,b,tb,db);
value oper,a,ta,da,b,tb,db; integer oper,a,ta,da,b,tb,db;
begin integer x,y,i,degree,j;
  FIX; if ta ≠ tr pow series then da:=100 000 else
  if tb ≠ tr pow series then db:=100 000;
  degree:=if da < db then da else db;
  begin integer array coeff,coeffa,coeffb[0:degree];
    if da = 100 000 then
    begin for i:=1 step 1 until degree do coeffa[i]:=zero;
      coeffa[0]:=a
    end else COEFFICIENT(a,degree,x,coeffa);
    if db = 100 000 then
    begin for i:=1 step 1 until degree do coeffb[i]:=zero;
      coeffb[0]:=b
    end else COEFFICIENT(b,degree,y,coeffb);
    if oper = sum then
    begin for i:=0 step 1 until degree do
      coeff[i]:=SIMPLIFY(S(coeffa[i],coeffb[i]))
    end else if oper = product then
    begin for i:=0 step 1 until degree do
      coeff[i]:= SIMPLIFY(Sum(j,0,i,P(coeffa[j],coeffb[i-j])))
    end else if oper = quotient then
    begin for i:=0 step 1 until degree do coeff[i]:=
      SIMPLIFY(Q(D(coeffa[i],Sum(j,0,i-1,P(coeff[j],coeffb[i-j]))),coeffb[0]))
    end;
    x:=TPS(i,degree,if da = 100 000 then y else x,coeff[i]);
    ERASE BUT RETAIN(i,1,1,x); OPER ON TPS:=x
end end OPER ON TPS;
```

This procedure performs the elementary operations on truncated power-series. It is assumed that at least one of the parameters a and b is a truncated power-series. If a or b is not a truncated power-series, it is treated as a truncated power-series with *degree 100 000*, whose coefficients, except the zeroth, are *zero* whilst the zeroth coefficient is a or b itself.
It is assumed that the two truncated power-series are power series in the same variable (say x). It is, moreover, assumed that a and b are of the form:

$$a = \sum_{i=0}^{da} coeffa[i]\ x^i + O(x^{da+1})$$

$$b = \sum_{i=0}^{db} coeffb[i]\ x^i + O(x^{db+1})$$

A consequence of the last assumption is, that the *degree* of the resulting truncated power-series becomes equal to the minimum of da and db.

The following relations are used to calculate the coefficients *coeff*[i] of the resulting truncated power-series:

$a+b$: $\quad coeff[i] := coeffa[i] + coeffb[i];$

$a\times b$: $\quad coeff[i] := \sum_{j=0}^{i} coeffa[j]\times coeffb[i-j];$

a/b: $\quad coeff[i] := (coeffa[i] - \sum_{j=0}^{i-1} coeff[j]\times coeffb[i-j])/coeffb[0];$

To the obtained coefficients *coeff*[i],the *SIMPLIFY* operator (section 2.10) is applied.
It is remarked that intermediate results are erased in order to save storage space.
In general, the result is stored as a truncated power-series; it may, however, occur that all the coefficients *coeff*[i], $i > 0$, turn out to be *zero*, then there are two possibilities:

1. *coeff*[*0*] is not a truncated power-series and *OPER ON TPS*:= *coeff*[*0*];
2. *coeff*[*0*] is a truncated power-series and the result is stored as a truncated power-series with all coefficients, except *coeff*[*0*], equal to *zero*.

The reason for transforming a truncated power series into a constant *coeff*[*0*] is, that a constant does not need as much storage space as a truncated-power series, whose coefficients *coeff*[*i*], i = 1, 2, ... are *zero*.
We have chosen in favour of the efficiency instead of the mathematical more elegant way in which a truncated power-series with coefficients equal to zero remains a truncated power-series.

The reason for separating the two mentioned possibilities is that we want to be able to treat truncated power-series in several variables, such that the hierarchy of the variables is preserved.
Example: consider the truncated power-series:

$$p(x) = c_0 + c_1x + c_2x^2 + O(x^3),$$

in which each c_i is a truncated power-series in y:

$$c_i = c_{i,0} + c_{i,1}y + c_{i,2}y^2 + c_{i,3}y^3 + O(y^4).$$

If $q(x)$ is another truncated power-series in x with coefficients d_i, which are truncated power-series in y, then the formula-manipulation system will always deliver the correct result for the operations $p(x) + q(x)$, $p(x) \times q(x)$ and $p(x)/q(x)$.
If it turned out that $c_1 = c_2 = 0$, and $p(x)$ was set equal to c_0, instead of $c_0 + 0x + 0x^2 + O(x^3)$, then the result of e.g. $p(x) + q(x)$ would be erroneous:

$$p(x) + q(x) = c_{0,0} + c_{0,1}y + c_{0,2}y^2 + c_{0,3}y^3 + O(y^4) + d_0 + d_1x + d_2x^2 + O(x^3)$$

and *OPER ON TPS* would deliver:

$$c_{0,0} + d_0 + (c_{0,1} + d_1)y + (c_{0,2} + d_2)y^2 + O(y^3).$$

This is a consequence of the fact that *OPER ON TPS* does not check whether the two variables of the truncated power-series to be summed are equal.

It is the user who should prevent such a situation by consistently considering truncated power-series in the variable y as coefficients of truncated power-series in the variable x.

We now give some more auxiliary procedures:

```
procedure VAL OF NUM(f,r,i); value f; integer f; real r,i;
begin integer n,type; TYPE(f,type,n);
  if type = integer then begin VAL OF INT NUM(f,n); r:= n; i:= 0 end
  else if type = real then begin VAL OF REAL NUM(f,r); i:= 0 end
  else VAL OF COMP NUM(f,r,i)
end VAL OF NUM;

integer procedure ARITHMETIC(oper,a,b); value oper,a,b; integer oper,a,b;
begin real r,i,ra,ia,rb,ib;
  VAL OF NUM(a,ra,ia); VAL OF NUM(b,rb,ib);
  if oper = sum then begin r:= ra + rb; i:= ia + ib end else
  if oper = product then
  begin r:= ra × rb - ia × ib; i:= ra × ib + ia × rb end else
  begin r:= rb × rb + ib × ib; if abs(r) ≤ null × null then
    begin r:= 1/null; i:= 0 end else
    begin i:= (ia × rb - ra × ib)/r; r:= (ra × rb + ia × ib)/r end
  end; ARITHMETIC:= CN(r,i)
end ARITHMETIC;

integer procedure Sum(i,lb,ub,fi); value lb,ub; integer i,lb,ub,fi;
begin integer s; s:= zero; for i:= lb step 1 until ub do s:= S(s,fi); Sum:= s
end Sum;

integer procedure COMB(n,m); value n,m; integer n,m;
COMB:= if m = 0 then 1 else (COMB(n,m-1) × (n + 1 - m)) : m;

integer procedure REPEATED PRODUCT(f,n); value f,n; integer f,n;
begin integer a; a:= INT POW(f,n:2); REPEATED PRODUCT:= P(P(a,a),
  if (n:2)×2=n then one else f)
end REPEATED PRODUCT;
```

```
integer procedure comm div(a,b,f); integer a,b,f;
begin integer r,cd; cd:= COMMON DIVISOR(a,b);
  if cd ≠ one then
  begin a:= QUOTIENT(a,cd,r); b:= QUOTIENT(b,cd,r) end;
  comm div:= f
end comm div;
```

A short explanation of these procedures follows:

VAL OF NUM: the value of a number f is assigned to the real variables r and i (for the real and imaginary parts). No distinction is made between integral, real or complex numbers.

ARITHMETIC: becomes a number as the result of some calculation.

Sum: becomes the formula $\sum_{i=lb}^{ub} fi$.

COMB: becomes equal to the value of the combinatorial coefficient:

$$\binom{n}{m} = \frac{n!}{(n-m)!\,m!}$$

REPEATED PRODUCT: becomes equal to the formula $\prod_{i=1}^{n} f$. This procedure is only used if f is a number or a truncated power-series.

comm div: if there exists a common divisor of a and b, not equal to unity, then a and b are divided by this common divisor and they get other values. *comm div* becomes equal to f (which may depend on a and b).

Next the procedures *S*, *D*, *P*, *Q*, *POWER* and *INT POW* are reproduced:

```
integer procedure S(a,b); value a,b; integer a,b;
begin integer ta,la,ra,tb,lb,rb;
  ta:= TYPE(a,la,ra); tb:= TYPE(b,lb,rb);
  S:= if a = zero then b else if b = zero then a else
    if ta = number ∧ tb = number then ARITHMETIC(sum,a,b) else
```

```
    if expand then
      (if ta = tr pow series ∨ tb = tr pow series then
        OPER ON TPS(sum,a,ta,la,b,tb,lb) else
      if ta = quotient ∧ tb = quotient then
        P(Q(one,ra),comm div(ra,rb,Q(S(P(la,rb),P(lb,ra)),rb))) else
      if ta = quotient then Q(S(la,P(b,ra)),ra) else
      if tb = quotient then Q(S(P(a,rb),lb),rb) else
      STORE(a,sum,b))
    else STORE(a,sum,b)
end S;

integer procedure D(a,b); value a,b; integer a,b;
D:= if (TYPE(a,di,di) = number ∧ TYPE(b,di,di) = number) ∨
  a = zero ∨ b = zero ∨ expand then S(a,P(minone,b))
  else STORE(a,difference,b);

integer procedure P(a,b); value a,b; integer a,b;
begin integer ta,la,ra,tb,lb,rb;
  ta:= TYPE(a,la,ra); tb:= TYPE(b,lb,rb);
  P:= if a = zero ∨ b = zero then zero else if a = one
    then b else if b = one then a else
    if ta = number ∧ tb = number then ARITHMETIC(product,a,b) else
    if expand then
      (if ta = tr pow series ∨ tb = tr pow series then
        OPER ON TPS(product,a,ta,la,b,tb,lb) else
      if ta = quotient ∧ tb = quotient then
        comm div(la,rb,comm div(lb,ra,Q(P(la,lb),P(ra,rb)))) else
      if ta = quotient then comm div(ra,b,Q(P(la,b),ra)) else
      if tb = quotient then comm div(a,rb,Q(P(a,lb),rb)) else
      if ta = sum then S(P(la,b),P(ra,b)) else
      if tb = sum then S(P(a,lb),P(a,rb)) else
      STORE(a,product,b))
    else STORE(a,product,b)
end;
```

```
integer procedure Q(a,b); value a,b; integer a,b;
begin integer ta,la,ra,tb,lb,rb,a1; a:= a1:= SIMPLIFY(a);
  b:= SIMPLIFY(b); ta:= TYPE(a,la,ra); tb:= TYPE(b,lb,rb);
  if ta = number ∧ tb = number then
  begin Q:= ARITHMETIC(quotient,a,b); goto END end;
  Q:= if a = zero then zero else if b = one then a else
    if tb = number then P(Q(one,b),a) else if expand then
      (if ta = tr pow series ∨ tb = tr pow series then
        OPER ON TPS(quotient,a,ta,la,b,tb,lb) else
      if ta = quotient ∧ tb = quotient then
        comm div(la,lb,comm div(ra,rb,Q(P(la,rb),P(lb,ra)))) else
      if ta = quotient then comm div(la,b,Q(la,P(ra,b))) else
      if tb = quotient then comm div(a,lb,Q(P(a,rb),lb)) else
      comm div(a,b,if a = a1 ∨ (TYPE(a,la,ra) ≠ number ∧
      TYPE(b,lb,rb) ≠ number) then STORE(a,quotient,b) else Q(a,b)))
    else STORE(a,quotient,b);
END: end Q;

integer procedure POWER(a,b); value a,b; integer a,b;
POWER:= if (TYPE(a,di,di) = number ∧ TYPE(b,di,di) = number) ∨
  expand then EXP(P(b,LN(a))) else STORE(a,power,b);

integer procedure INT POW(a,n); value a,n; integer a,n;
begin integer i,t,la,ra; t:= TYPE(a,la,ra);
  INT POW:= if n = 0 then one else if n = 1 then a else
    if n < 0 then Q(one,INT POW(a,-n)) else
    if t = number then REPEATED PRODUCT(a,n) else
    if expand then
      (if t = tr pow series then REPEATED PRODUCT(a,n) else
      if t = sum then Sum(i,0,n,P(IN(COMB(n,i)),
      P(INT POW(la,n-i),INT POW(ra,i)))) else
      if t = product then P(INT POW(la,n),INT POW(ra,n)) else
      if t = quotient then Q(INT POW(la,n),INT POW(ra,n)) else
      if t = integral power then INT POW(ra,la × n) else
      STORE(n,integral power,a))
    else STORE(n,integral power,a)
end INT POW;
```

The construction of these procedures is in principle the same as the construction of the procedures *S* and *P* of the simple system described in chapter 1.

A complicating element in these procedures is the division, to which most of the following discussion is devoted.

First, some trivial remarks:

1. If the nonlocal Boolean variable *expand* has the value *false*, then the formulae are stored as they are written, except for a trivial simplification with the unit- and the zero elements *one* and *zero*.

2. Numerical calculations are carried out where possible. Thus e.g. "a + (1 + 2)" is stored as "a + 3", whereas "(a + 1) + 2" is stored as it stands due to the brackets around "a + 1"; note that if *expand* then a call of *SIMPLIFY* will deliver the result "a + 3".

3. Due to the construction of *IN*, *RN* and *CN*, due to the treatment of numbers as illustrated in the above remark and due to the construction of the procedures for storing functions, the result of storing, for example, the formula "(i × (1-i)/(1+i)) × a + ln(1) × b", where "i" is the imaginary unit and "a" and "b" algebraic variables, is simply "a", provided that the absolute precision *null* and the relative accuracy *rel acc* are not chosen too small.

4. In the following remarks it is assumed that *expand* has the value *true*.

5. $a - b$ is stored as $a + (-1) \times b$;
   a^b is stored as $\exp(b \times \ln(a))$.

6. If one of the parameters a and b (or both) of the procedures *S*, *D*, *P* or *Q* is a truncated power-series then the result of the calculation will also be a truncated power-series.

7. The procedure *INT POW* delivers,if a is a number or a truncated power-series,an efficiently formed product.
   If a is a sum, a product or a quotient then *INT POW* expands a.
   If a is the integral power "x^m" then the result is the integral power "$x^{m \times n}$".

Without division and without functions, simplification can easily be defined in the following way:

Two formulae "f" and "g" are called equivalent if
1. "f" and "g" are the same numbers, or
2. "f" and "g" are the same algebraic variables, or
3. the simplification of "f - g" yields 0.
Let "n" and "m" be numbers, let "f" and "g" be equivalent, then the simplification of "n × f + m × g" yields "(n + m) × f".

Since the ultimate test for equivalency is a test on the elementary constituants of a formula, i.e. the algebraic variables and the numbers, simplification should be carried out on the expanded formulae, i.e. a sum of products of algebraic variables and numbers.
For this expansion the procedure *P* defined at the end of chapter 1 (point 4 of the remarks) may be helpful.

It should be noticed that there exists no common agreement on the definition of simplification. This is not surprising since simplification strongly depends on the specific formula manipulations one wants to do.
A definition of simplification in a general system, with formulae built up by algebraic variables of which nothing is known does not leave much choice, however.
Factorization, for example, would be a nice facility and in some cases this can be programmed in an elementary way, e.g. "ax + bx = (a + b)x", in other cases, however, it can not be programmed in an elementary way as e.g. "$a^2 - b^2$".
On the other hand, the user who knows the form of the formulae beforehand, can build his own simplification procedure, which e.g. also factorizes.

Extension of the field of formulae with exponential functions offers no serious difficulties. We want e.g. to achieve that "$\sin^2(x+y) + \cos^2(y+x)$" is simplified into"1",which is a result of the following transformations:

exp(a) × exp(b) → exp(a + b)

and

$a \times \exp(f) - b \times \exp(g) \rightarrow 0$

if "f" is equivalent to "g" apart from a multiple of "$2\pi i$", and "a" is equivalent to "b".

Logarithmic functions are more troublesome; it is not evident whether "ln(a) + ln(b)" should be transformed into "ln(a × b)" or vice versa. Moreover, it is not evident whether "a × ln(g)" should be transformed into "$\ln(g^a)$", since this may lead to difficulties when "a = ln(f)". Finally, in view of the fact that the logarithm is a multivalued function, it seems doubtful to conclude, from the (analytic) equality of "f" and "g", that "ln(f)" and "ln(g)" are equivalent.
We decided, therefore, not to build in any simplification involving the special properties of logarithmic functions. A consequence is that the program recognizes that two logarithms "l_1" and "l_2" are equivalent only, if the ALGOL 60 variables l_1 and l_2, corresponding to "l_1" and "l_2" respectively, are equal; e.g. in the case that l_1 and l_2 have obtained values through the statements: *l[1]:= LN(x); l[2]:= l[1]*.
If, on the other hand, l_1 and l_2 have obtained values through: *l[1]:= LN(x); l[2]:= LN(x)*, then l_1 and l_2 are not recognized to be the same.

The effect of the implementation of division in the system on simplification allows simplification to have the following three properties:

1. The simplification of a formula "f" yields a quotient of a numerator "n" and a denominator "d".
2. "n" and "d" are formulae in which no division operator occurs (unless, possibly, within the argument of a function) and which are simplified according to the above definition.
3. Moreover, "n" and "d" do not contain common divisors which can be found in an elementary way other than numbers.
   This sentence may be replaced by: "n" and "d" are such that *COMMON DIVISOR*(n,d) = *one*.

Anticipating the description of the procedure *COMMON DIVISOR* in section 2.11, a few remarks are made.

COMMON DIVISOR calculates a common divisor of two formulae "f" and "g". The calculation is never successful if "f" or "g" contains exponential functions; in the sequel it will be assumed, therefore, that this is not the case.
The calculation is always successful if either "f" is a factor of "g" or "g" is a factor of "f". The calculation is not successful if "f" and "g" are e.g. of the form "f = a × x" and "g = b × x", where "a", "b" and "x" are different algebraic variables.
If only one algebraic variable occurs in both "f" and "g" then the calculation is always successful. Thus, *COMMON DIVISOR* will found out that, for example, "x^2 + 2 × x + 1" and "x^2 - 1" have a common divisor "2 × x + 2".

The second property of simplification requires procedures which not only apply the distributive law, but also rules of the following kind:

1. a/b + c → (a + c × b)/b
2. a + c/b → (a × b + c)/b
3. a/b × c → (a × c)/b
4. a × (b/c) → (a × b)/c
5. (a/b)/c → a/(b × c)
6. a/(b/c) → (a × c)/b

It is evident, however, that these rules can not be applied without precautions, since they may lead to a simplified formula which is more complicated than the original formula.
Example: let "a", "b" and "x" be algebraic variables.

The formulae:	would then be stored as:
"(a/x) + (b/x)"	"(a × x + b × x)/(x × x)"
"(a/(b × x)) × x"	"(a × x)/(b × x)"
"((a × x)/b)/x"	"(a × x)/(b × x)"

Simplifying the right-hand sides amounts to calculating the common divisor of the numerator and denominator. This calculation will, however, be unsuccessful.

Thus, a test is built in the procedures *S*, *P* and *Q* to prevent these situations and it is for this reason that the already discussed procedure *comm div* is introduced. The effect is that the three formulae above are stored as "(a + b)/x", "a/b" and "a/b", respectively.

Notice that the distributive law is applied in *P* only in case the actual values of the parameters *a* and/or *b* are no quotients. Otherwise, *P* would transform "(1/x) × (a + b)" into "((1/x) × a + (1/x) × b)", which is transformed into "a/x + b/x", which is transformed by *S* into "(1/x) × (a + b)", etc., etc.

It is also in order not to get a more complicated formula, that the parameters *a* and *b* of *Q* are simplified beforehand, which is not necessary for the parameters of *S* and *P*.

2.9. The functions

The procedures *EXP*, *LN*, *SIN*, *COS*, *ARCTAN* and *SQRT* are now reproduced:

```
integer procedure EXP(f); value f; integer f;
begin integer t,a,b; t:= TYPE(f,a,b);
  if t = function ∧ a = lnf then EXP:= b else
  if t = number then
  begin real r,i; VAL OF NUM(f,r,i); r:= exp(r);
    EXP:= CN(r × cos(i),r × sin(i))
  end else EXP:= STORE(expf,function,f)
end EXP;

integer procedure LN(f); value f; integer f;
begin integer t,a,b; t:= TYPE(f,a,b);
  if t = function ∧ a = expf then LN:= b else
  if t = number then
  begin real r,i; VAL OF NUM(f,r,i);
    LN:= if f = zero then RN(ln(null)) else CN(.5 × ln(r × r + i × i),
    if abs(r) ≤ null then sign(i) × 1.57079632679 else
    if r > null then arctan(i/r) else (if i < 0 then -1 else 1) ×
    1.57079632679 - arctan(r/i))
  end else LN:= STORE(lnf,function,f)
end LN;

integer procedure SIN(f); value f; integer f;
SIN:= if TYPE(f,di,di) = number ∨ expand then
  P(CN(0,-.5),D(EXP(P(im unit,f)),EXP(P(CN(0,-1),f)))) else
  STORE(sinf,function,f);

integer procedure COS(f); value f; integer f;
COS:= if TYPE(f,di,di) = number ∨ expand then
  P(RN(.5),S(EXP(P(im unit,f)),EXP(P(CN(0,-1),f)))) else
  STORE(cosf,function,f);
```

```
integer procedure ARCTAN(f); value f; integer f;
ARCTAN:= if TYPE(f,di,di) = number ∨ expand then
  P(CN(0,-.5),LN(Q(S(one,P(im unit,f)),D(one,P(im unit,f))))) else
  STORE(arctanf,function,f);

integer procedure SQRT(f); value f; integer f;
SQRT:= if TYPE(f,di,di) = number ∨ expand then
EXP(P(RN(.5),LN(f))) else STORE(sqrtf,function,f);
```

A few remarks are made:

1. If *expand* has the value *false*, then the procedures store the functions as they are written; otherwise, the functions are transformed into exponential and logarithmic functions.
2. If the parameter f is a number, then the result is another number.
3. The standard function *arctan(x)* delivers a result between $-\pi$ and $+\pi$.
4. The logarithm of a number will get an imaginary part y with $-\pi \leq y < \pi$.
5. The manner in which the elementary functions of complex numbers are calculated is rather bad numerically. A better, but more complicated, manner is described in chapter 4.

2.10. The simplification procedures

Already in section 2.8 simplification was extensively discussed. This section is devoted to a discussion of the procedures which perform the simplification.

The internally represented formula is, due to the construction of *S*, *P*, *Q* and *INT POW*, represented as a quotient of a numerator and a denominator, in which no quotients occur anymore (except, possibly, in the argument of a function).
The numerator and the denominator are represented as a sum of terms, and each term as a product of factors, i.e. integral powers of formulae which are neither a sum, nor a product, nor a quotient. The simplifying process has, therefore, as objects formulae, which have the structure as given above for the numerator and the denominator.

The chosen tree structure for the internal representation of formulae is very convenient for storing and extracting formulae; it is, however, very inconvenient for the simplification process, since we shall use the property that the terms of a sum or the factors of a product can freely be permuted in order to combine the terms or the factors with each other.

A second way of representing a formula "f" is by means of the integer arrays a and L. The array elements of a and L are such that

$$f = \sum_{i=1}^{L[0]} a[i,0,1] \times \prod_{j=1}^{L[i]} a[i,j,1] \uparrow a[i,j,2],$$

where $a[i,0,1]$ is a formula of type *number* and where the $a[i,j,1]$ are formulae not of type *integral power*, *sum*, *product* or *number*.

Remark:

In order to use the storage space efficiently, the simplifying procedures do not actually use an array a with three subscripts, but an array a with two subscripts.
The array elements $a[i,j,1]$ and $a[i,j,2]$, which will be used in the discussion, are actually stored as the number
$a[i,j,1] + a[i,j,2] \times 2\uparrow 13$

into one array element $a[i,j]$ (see also section 2.5).

The procedure $T(a,i,j,k)$ is used to extract the $a[i,j,1]$ and $a[i,j,2]$; in fact, $a[i,j,1] = T(a,i,j,1)$ and $a[i,j,2] = T(a,i,j,2)$.

Example: "$- 5 \times x^2 \times 2i \times y \times i \times x + i \times y \times i \times x^2$" is represented as:

$(a[i,j,1],\ a[i,j,2]) =$

i =	$j = 0$	$j = 1$	$j = 2$	$j = 3$	$L[i]$ =
1	(*one*,1)	(x,2)	(y,1)	(x,1)	3
2	(*minone*,1)	(y,1)	(x,2)		2

It is the task of the procedure *CONVERT*, which is now reproduced, to convert a formula from the tree structure representation into the second representation using the arrays a and L.

If the actual value of the Boolean parameter *bounds only* of *CONVERT* has the value *true*, then only the array bounds of a and L are calculated. Otherwise, the array elements of a and L are filled in.

```
Boolean procedure CONVERT(f,a,L,bounds only); value f; integer f;
integer array a,L; Boolean bounds only;
begin integer i,Li,Limax,num,n of num;

  procedure SUM(f); value f; integer f;
  begin integer f1,f2; if TYPE(f,f1,f2) = sum then
    begin SUM(f1); SUM(f2) end else
    begin Li:= 0; num:= one; i:= i + 1; n of num:= 0; PROD(f);
      if n of num ≥ 2 then CONVERT:= true;
      if bounds only then Limax:= if Limax < Li then Li else Limax
      else begin a[i,0]:= num; L[i]:= Li end
  end end SUM;
```

```
procedure PROD(f); value f; integer f;
begin integer type,f1,f2; type:= TYPE(f,f1,f2); if type = product then
  begin PROD(f1); PROD(f2) end else
  if type = number ∧ ¬ bounds only then
  begin n of num:= n of num + 1; num:= P(num,f) end else
  begin Li:= Li + 1; if ¬ bounds only then
    begin if type = integral power then a[i,Li]:= f2 + f1×t13
      else a[i,Li]:= f + t13
  end end
end PROD;
i:= Limax:= 0; CONVERT:= false; SUM(f); L[0]:= i;
if bounds only then L[1]:= Limax
end CONVERT;

integer procedure T(a,i,j,k); value i,j,k; integer i,j,k;
  integer array a; T:= if k = 1 then a[i,j] - (a[i,j]:t13)×t13 else
  a[i,j]:t13;
```

Note that the procedure *CONVERT* gets the value *true* or *false* dependent on whether two or more numbers could be combined into one number.
The Boolean procedure *SIMPL 2 REPR* is discussed now. It transforms a formula given in the arrays a and L. The result of this transformation, which might be called a simplification, is another formula, equivalent to the old one and also represented in the arrays a and L.
The form of the resulting formula is a standard form satisfying the following conditions:

1. If the formula is equal to *zero*, then $L[0] = 0$.
2. For $i = 1, \ldots, L[0]$ and $j = 1, \ldots, L[i] - 1$, $a[i,j,1]$ is not an exponential function.
   If $a[i,L[i],1]$ happens to be an exponential function, then its argument is simplified; moreover, this argument is not a sum with a number as summand.

In the following conditions it is assumed that the exponential functions are temporarily removed by replacing $L[i]$ by $L[i] - 1$ if $a[i,L[i],1]$ happens to be an exponential function.

3. For $i = 1, \ldots, L[0]$, and $j = 1, \ldots, L[i]$,

$$a[i,0,1] \neq zero, \tag{2.10.1}$$

$$a[i,j,2] > 0 \tag{2.10.2}$$

and for $j = 1, \ldots, L[i] - 1$,

$$a[i,j,1]^* < a[i,j+1,1]^*. \quad {}^{*)} \tag{2.10.3}$$

4. Let $length[i]$ be defined by

$$length[i] = \sum_{j=1}^{L[i]} a[i,j,2], \tag{2.10.4}$$

then, for $i = 1, \ldots, L[0] - 1$,

$$length[i] \geq length[i+1] \geq 0. \tag{2.10.5}$$

5. If for some i

$$length[i] = length[i+1],$$

then

$$\begin{aligned} \exists j_0 > 0: \{ & (a^*[i,j_0,1] < a^*[i+1,j_0,1] \vee \\ & a^*[i,j_0,1] = a^*[i+1,j_0,1] \wedge a[i,j_0,2] > a[i+1,j_0,2]) \\ & \wedge (j_0 > 0 \Longrightarrow \forall_j,\ 0 < j < j_0: \\ & a^*[i,j,1] = a^*[i+1,j,1] \wedge a[i,j,2] = a[i+1,j,2])\}. \end{aligned} \tag{2.10.6}$$

*) The meaning of $a[i,j,1]^*$ is explained in remark 1 of chapter 1 (p. 12).

If

$$a[i,j,k] = a[i+1,j,k] \qquad (2.10.7)$$

for $j = 1, \ldots, L[i]$ and $k = 1, 2$,
then, at least one of the original array elements: $a[i,L[i],1]$ or $a[i+1,L[i+1],1]$, was an exponential function before replacing $L[i]$ by $L[i] - 1$, or $L[i+1]$ by $L[i+1] - 1$; if both were exponential functions then their arguments are not equivalent, because, otherwise, the two terms, as given by the indexes i and $i+1$, would have been combined into one term.

Examples of formulae written in standard form are (let $x^* < y^* < z^*$):

$x \times y \times z \times e^{x+y+z}$

$x \times y^2 \times z^3$

$z^3 + y^2 + x$

$.5 \times x^2 \times y - .5 \times x^2 \times z + 7i$

$x \times y^2 \times z + x \times y \times z^2$

$x \times y \times e^z + x \times z \times e^y + y \times z \times e^x$

$x \times y \times z \times e^{x+y+z} + x \times y \times z \times e^{-x-y-z}$

$x \times y \times z \times e^{-x-y-z} + x \times y \times z \times e^{x+y+z}$

$e^{2x} + e^{2y}$

$e^{2y} + e^{2x}$

The last four examples show that the ordering of terms, in which exponential functions occur, is not fixed if the nonexponential parts are equal.

The transformations, carried out by the procedure *SIMPL 2 REPR* in order to get a standard form, can easily be traced in the procedure declaration reproduced now:

```
Boolean procedure SIMPL2REPR(a,L); integer array a,L;
begin integer k,i,j,i1,p,q,r,b,c,sexp; integer array exp,length[1:L[0]],
  s[1:2×L[0]]; SIMPL2REPR:= false;
  FIX; for i:= 1 step 1 until L[0] do
  begin sexp:= zero; for j:= 1 step 1 until L[i] do
    begin if TYPE(T(a,i,j,1),b,c) = function ∧ b = expf then   *)
      begin sexp:= S(sexp,P(IN(T(a,i,j,2)),c)); a[i,j]:= t13 - 1
    end end; exp[i]:= SIMPLIFY(- sexp);
    if exp[i] ≠ zero then
    begin if TYPE(exp[i],q,r) = number then
      begin a[i,0]:= P(a[i,0],EXP(exp[i])); exp[i]:= zero end else
      if TYPE(exp[i],q,r) = sum then
      begin if TYPE(r,b,c) = number then
        begin a[i,0]:= P(a[i,0],EXP(r)); exp[i]:= q end
  end end end; for i:= 1 step 1 until L[0] do
  begin p:= q:= 1; r:= L[i] - 1;
  A: for j:= p step q until r do
    begin if T(a,i,j,1) = T(a,i,j+1,1) then
      begin a[i,j]:= a[i,j] + T(a,i,j+1,2) × t13; a[i,j+1]:= t13 - 1;
        if q = -1 then begin i1:= j + 1; goto B end
      end else if T(a,i,j,1) > T(a,i,j+1,1) then
      begin k:= a[i,j]; a[i,j]:= a[i,j+1]; a[i,j+1]:= k;
        if q = 1 then begin p:= j - 1; q:= -1; r:= 1; i1:= j + 1; goto A end
      end else if q = -1 then goto B
    end; B: if q = -1 then begin p:= i1; q:= 1; r:= L[i] - 1; goto A end;
    for j:= 1 step 1 until L[i] do
    begin if a[i,j] = t13 - 1 then goto C end;
    goto D; C: L[i]:= j - 1; SIMPL2REPR:= true;
  D: end;
  for i:= 1 step 1 until L[0] - 1 do for j:= i + 1 step 1 until L[0] do
  begin if a[i,0] = zero ∨ a[j,0] = zero ∨ L[i] ≠ L[j] then goto EXIT;
    for k:= 1 step 1 until L[i] do
    begin if T(a,i,k,1) ≠ T(a,j,k,1) ∨ T(a,i,k,2) ≠ T(a,j,k,2) then goto EXIT end;
```

*) Note that *b* obtains a value by means of *TYPE*.

```
    if ¬ EQUAL(exp[i],exp[j]) then goto EXIT;
    a[i,0]:= S(a[i,0],a[j,0]); a[j,0]:= zero;
  EXIT: end;
  for i:= 1 step 1 until L[0] do
  begin s[i]:= i; length[i]:= 0; if a[i,0] = zero then L[i]:= 0;
    for j:= 1 step 1 until L[i] do length[i]:= length[i] + T(a,i,j,2)
  end;
  p:= q:= 1; r:= L[0] - 1;
AA: for i:= p step q until r do
  begin if length[s[i]] > length[s[i+1]] then goto OUT;
    if length[s[i]] < length[s[i+1]] then goto INTERCHANGE;
    j:= 0; for j:= j + 1 while j ≤ L[s[i]] ∧ j ≤ L[s[i+1]] do
    begin if T(a,s[i],j,1) < T(a,s[i+1],j,1) then goto OUT;
      if T(a,s[i],j,1) > T(a,s[i+1],j,1) then goto INTERCHANGE;
      if T(a,s[i],j,2) > T(a,s[i+1],j,2) then goto OUT;
      if T(a,s[i],j,2) < T(a,s[i+1],j,2) then goto INTERCHANGE
    end; if a[s[i],0] = zero ∧ a[s[i+1],0] ≠ zero then goto
    INTERCHANGE; goto OUT;
  INTERCHANGE: k:= s[i]; s[i]:= s[i+1]; s[i+1]:= k; if q = 1 then
    begin p:= i - 1; r:= 1; q:= -1; i1:= i + 1; goto AA end;
    goto CC;
  OUT: if q = -1 then goto BB;
  CC: end; BB: if q = -1 then
  begin p:= i1; q:= 1; r:= L[0] - 1; goto AA end;
  for i:= 1 step 1 until L[0] do
  begin if exp[i] ≠ zero ∧ a[i,0] ≠ zero then
    begin L[i]:= L[i] + 1; a[i,L[i]]:= EXP(exp[i]) + t13
  end end;
  p:= 0; for i:= 1 step 1 until L[0] do
  begin p:= i; if s[i] ≠ i ∧ a[s[i],0] ≠ zero then
    begin j:= -1; for j:= j + 1 while j ≤ L[s[i]] ∧ j ≤ L[i] do
      begin  k:= a[i,j]; a[i,j]:= a[s[i],j]; a[s[i],j]:= k end;
      for j:= L[s[i]] + 1 step 1 until L[i] do a[s[i],j]:= a[i,j];
```

```
    for j:= L[i] + 1 step 1 until L[s[i]] do a[i,j]:= a[s[i],j];
    k:= L[i]; L[i]:= L[s[i]]; L[s[i]]:= k;
    for j:= i + 1 step 1 until L[0] do
    begin if s[j] = i then begin s[j]:= s[i]; goto AB end end;
   AB: s[i]:= i
   end else if a[s[i],0] = zero then begin p:= p - 1; goto END end
  end; END: if p < L[0] then begin SIMPL2REPR:= true; L[0]:= p end;
  for i:= 1 step 1 until p do
  begin s[i]:= a[i,0]; s[i+p]:= if L[i] > 0 then T(a,i,L[i],1) else zero
  end; ERASE BUT RETAIN(i,1,2 × p,s[i]);
  for i:= 1 step 1 until p do
  begin a[i,0]:= s[i]; if L[i] > 0 then a[i,L[i]]:= a[i,L[i]] -
    T(a,i,L[i],1) + s[i+p]
  end
end SIMPL2REPR;
```

Remarks:

1. The procedure *ERASE BUT RETAIN* is used to restore the arguments of the exponentials and the newly found numerical factors *a[i,0,1]*, in an efficient way.

2. In order to simplify an exponential function, e.g. "exp(x + 2πi + y)" into "exp(x + y)", the procedure *SIMPL 2 REPR* has to know the numerical part of the argument such as "2πi" in the above example. Let this argument be the formula *f*, then the statement *g:= SIMPLIFY(−f)* delivers a formula in the standard form. If *g* is a sum with a number as summand, then this number is given by the rhs quantity of *g*. In the above example, *g* = (x + y) + 2πi.
   We remark, that the statement *g:= SIMPLIFY(f)* delivers a formula in the standard form, only if simplification was successful, i.e. two or more terms could be combined into one term, or two or more factors could be combined into one factor.

3. The procedure *SIMPL 2 REPR* is a Boolean procedure and it gets the value *true* or *false* dependent on whether the simplification was successful.

Next follows the reproduction of the procedure declaration of *SIMPLIFY*. Due to the construction of the procedure *Q*, *SIMPLIFY* does not have to simplify a quotient.

```
integer procedure SIMPLIFY(f); value f; integer f;
begin integer i,j,t,a,b; integer array A[0:1],B[1:1,1:1];
  t:= TYPE(abs(f),a,b); if t = quotient ∨ ¬ expand ∨ t = number ∨
  t = algebraic variable ∨ t = tr pow series ∨ (t = function ∧ a ≠ expf)
  then begin SIMPLIFY:= abs(f); goto END end;
  CONVERT(abs(f),B,A,true);
  begin integer array s[1:A[0],0:A[1]],L[0:A[0]];
    if CONVERT(abs(f),s,L,false) ∨ f < 0 ∨ SIMPL2REPR(s,L) then     *)
    begin t:= zero; for i:= 1 step 1 until L[0] do
      begin b:= s[i,0]; for j:= 1 step 1 until L[i] do
        b:= P(b,INT POW(T(s,i,j,1),T(s,i,j,2))); t:= S(t,b)
      end; SIMPLIFY:= t
    end else SIMPLIFY:= abs(f)
  end;
END: end SIMPLIFY;
```

We finally give the Boolean procedure *EQUAL*:

```
Boolean procedure EQUAL(f,g); value f,g; integer f,g;
if f = g then EQUAL:= true else
begin FIX; EQUAL:= SIMPLIFY(D(f,g)) = zero; ERASE
end EQUAL;
```

*) Note that *CONVERT* is executed before *SIMPL 2 REPR*.

2.11. The procedures *QUOTIENT* and *COMMON DIVISOR*

In this section we discuss the integral-division and the greatest-common-divisor algorithms.
We shall assume that the formulae to be considered do not contain exponential functions.

2.11.1. Definition of the integral-division process

QUOTIENT, as well as *COMMON DIVISOR*, operates on two formulae f and g occurring as their actual parameters. In the sequel we shall exclude the trivial case that both f and g are numbers. By means of the arrays F, LF, G and LG, these formulae can be represented in the standard form as defined in section 2.10.

$$f = \sum_{i=1}^{LF[0]} F[i,0,1] \times \prod_{j=1}^{LF[i]} F[i,j,1] \uparrow F[i,j,2],$$
$$g = \sum_{i=1}^{LG[0]} G[i,0,1] \times \prod_{j=1}^{LG[i]} G[i,j,1] \uparrow G[i,j,2]. \qquad (2.11.1)$$

We introduce the set of basic formulae "a_1", ..., "a_m", whose elements "a_i" are all the different formulae amongst

$$F[i,j,1],\ i = 1, \ldots, LF[0],\ j = 1, \ldots, LF[i],$$

and

$$G[i,j,1],\ i = 1, \ldots, LG[0],\ j = 1, \ldots, LG[i].$$

Obviously, we may rewrite the formulae f and g, using the formulae "a_i":

$$f = \sum_{i=1}^{LF[0]} F[i,0,1] \times \prod_{j=1}^{m} a_j \uparrow p_{i,j}\ ,\quad p_{i,j} \geq 0,$$
$$g = \sum_{i=1}^{LG[0]} G[i,0,1] \times \prod_{j=1}^{m} a_j \uparrow q_{i,j}\ ,\quad q_{i,j} \geq 0. \qquad (2.11.2)$$

The sum of all the exponentials $p_{i,j}$ and $q_{i,j}$ is denoted by P:

$$P = \sum_{j=1}^{m} \left(\sum_{i=1}^{LF[0]} p_{i,j} + \sum_{i=1}^{LG[0]} q_{i,j} \right). \qquad (2.11.3)$$

Consider an arbitrary formula "h"; let its standard form be given by:

$$h = \sum_{i=1}^{LH[0]} H[i,0,1] \times \prod_{j=1}^{LH[i]} H[i,j,1] \uparrow H[i,j,2]. \qquad (2.11.4)$$

Definition 1: The leading term of "h", denoted by "lt(h)", is given by:

if "h" = 0 then lt(h) = 0; otherwise,

$$lt(h) = H[1,0,1] \times \prod_{j=1}^{LH[1]} H[1,j,1] \uparrow H[1,j,2]. \qquad (2.11.5)$$

Definition 2: the length of "h", denoted by "l(h)", is given by:

if "h" is a number then l(h) = 0; otherwise,

$$l(h) = \sum_{j=1}^{LH[1]} H[1,j,2]. \qquad (2.11.6)$$

Definition 3: A stub of "h", denoted by "st(h)", is any formula, which is the sum of an arbitrary number and a linear combination of the terms of "h - lt(h)".

Example: Let $h = x^2 + x + 1$,

then $lt(h) = x^2$,

$l(h) = 2$,

$st(h) = n_1x + n_2$, where "n_1" and "n_2" are arbitrary numbers.

Definition 4: The formula "h" is called admissible, if it is a number or if its standard form (2.11.4) satisfies the following conditions:

1. For each $H[i,j,1]$, $i = 1, \ldots, LH[0]$, $j = 1, \ldots, LH[i]$, there exists a basic formula "a_k", such that $H[i,j,1] = a_k$.
2. $l(h) \le P$.

Definition 5: The leading-term powers of an admissible formula "h", denoted by "$p_i(h)$", i = 1, ..., m are defined by:

if "h" is a number, then $p_i(h) = 0$, i = 1, ..., m; otherwise, they are defined by:

$$lt(h) = H[1,0,1] \times \prod_{i=1}^{m} a_i \uparrow p_i(h), \quad 0 \le p_i(h) \le P. \qquad (2.11.7)$$

Theorem 1: If "h" is an admissible formula then:

$$l(h) = \sum_{i=1}^{m} p_i(h). \qquad (2.11.8)$$

Proof: If "h" is a number, then the theorem follows immediately; otherwise, it is observed that

$\prod_{i=1}^{m} a_i \uparrow p_i(h)$ is just another way of writing:

$$\prod_{j=1}^{LH[1]} H[1,j,1] \uparrow H[1,j,2];$$

and it follows: $l(h) = \sum_{j=1}^{LH[1]} H[1,j,2] = \sum_{i=1}^{m} p_i(h)$.

Theorem 2: If "h" is an arbitrary formula, then

$$l(h) \geq l(st(h)). \qquad (2.11.9)$$

Proof: Condition 4 for the standard form states:

$$length[1] \geq length[2] \geq \ldots \geq length[LH[0]] \geq 0.$$

Since $l(h) = length[1]$ and since, either $l(st(h)) = 0$, or $l(st(h)) = length[i]$, for some $i > 1$, the theorem follows.

Definition 6: The rank of an admissible formula "h", denoted by "R(h)", is given by:

$$R(h) = \sum_{i=1}^{m} 2^{2Q \times (A-a_i^*)} \times p_i(h), \qquad (2.11.10)$$

where Q is a positive integral number such that:

$$P < 2^Q, \qquad (2.11.11)$$

(P is defined in (2.11.3)), and A is defined by:

$$A = \max_{i=1,..,m} (a_i^*) + 1. \qquad (2.11.12)$$

Remark:

This definition needs some explanation: The a_i^* are the integral-numerical values of the internally represented formulae "a_i". Moreover, the simplification process of section 2.10 defines a particular hierarchy between the array elements of H; this hierarchy is determined by the values of the relations

$$H[i_1,j_1,1]^* < H[i_2,j_2,1]^*, \quad H[i_1,j_1,2] > H[i_2,j_2,2] \tag{2.11.13}$$

and

$$length[i_1] \geq length[i_2]. \tag{2.11.14}$$

The ordering of two terms "h_1" and "h_2", in the formula "h", is such that "h_1" preceeds "h_2" if $l(h_1) > l(h_2)$; if $l(h_1) = l(h_2)$, then the ordering is determined by means of (2.11.13). It will be shown in theorem 3 that, in the last case, $R(h_1) > R(h_2)$ if "h_1" preceeds "h_2".

Theorem 3: Let "h" be an admissible formula and "s" a stub of "h"; and let $l(h) = l(s)$. Then, either "h" and "s" are numbers, or

$$R(h) > R(s). \tag{2.11.15}$$

Proof: Suppose "h" is not a number, then "s" is neither a number and $l(h) > 0$.

Consider the leading term of "s", which is apart from a numerical factor,

$$lt(s) = \prod_{j=1}^{LH[i_0]} H[i_0,j,1] \uparrow H[i_0,j,2],$$

where $1 < i_0 \leq LH[0]$.

From

$$length[1] \geq \ldots \geq length[i_0] = length[1],$$

it follows

$$length[1] = \ldots = length[i_0]. \tag{2.11.16}$$

Consider first, the "lt(h)" and the second term "s_2" of "h", defined by:

$$s_2 = \mathrm{lt}(h - \mathrm{lt}(h)). \tag{2.11.17}$$

Let the first case (2.10.6) of condition 5 of the standard form be applicable ($j_0 = 1$); i.e.

$$H[1,1,1]^* < H[2,1,1]^*. \tag{2.11.18}$$

Let $a_{i_1} = H[1,1,1]$ and $a_{i_2} = H[2,1,1]$; then

$$A - a_{i_1}^* > A - a_{i_2}^* \tag{2.11.19}$$

and

$$R(\mathrm{lt}(h)) \geq 2^{2Q(A-a_{i_1}^*)} \geq 2^{2Q(A-a_{i_2}^*)+Q} \times 2^Q. \tag{2.11.20}$$

From condition 3, for the standard form, it follows

$$H[2,j,1]^* < H[2,j+1,1]^*, \; j = 1, \ldots, LH[2] - 1. \tag{2.11.21}$$

Let the product

$$\prod_{j=2}^{LH[2]} H[2,j,1] \uparrow H[2,j,2]$$

be rewritten into

$$\prod_{j=1}^{m} a_j \uparrow p_j \;, \quad p_j \geq 0;$$

then those formulae "a_j" for which $p_j > 0$, satisfy

$$a_{i_2}^* < a_j^*. \tag{2.11.22}$$

Hence,

$$\begin{aligned} R(\mathrm{lt}(h)) &> \{2^{2Q(A-a_{i_2}^*)} + 2^{2Q(A-a_{i_2}^*)-1} + \ldots + 1\} \times 2^Q \\ &> \{2^{2Q(A-a_{i_2}^*)} \times P + \sum_{\substack{j=1 \\ (p_j > 0)}}^{m} 2^{2Q(A-a_j^*)} \times P\} \\ &\geq R(s_2). \end{aligned}$$

Thus, if the first case of condition 4 is applicable, then

$$R(h) > R(s_2). \qquad (2.11.23)$$

If one of the other cases of condition 5 is applicable, then the above arguments can be repeated. For, if $j_0 > 1$, we may drop the factors $H[1,j,1]$ and $H[2,j,1]$, with $j < j_0$, since they give the same contribution in $R(h)$ and $R(s_2)$. Hence j_0 may be chosen equal to 1.
If $H[1,1,1] = H[2,1,1]$, then $H[1,1,2] > H[2,1,2]$; we may now drop from the two terms the factor $H[1,1,1] \uparrow H[2,1,2]$, since this factor gives also the same contribution in $R(h)$ and $R(s_2)$, and we are left with the known situation that $H^*[1,1,1] < H^*[2,1,1]$, where $H^*[2,1,1]$ is the old $H^*[2,2,1]$.

Consider now the following sequence of formulae "h_i":

$$h_0 = h \ , \quad h_i = h_{i-1} - lt(h_{i-1}) \ , \ i = 1, \ldots, i_0.$$

From (2.11.16) it follows that $l(h_i) = l(h_{i-1})$, $i = 1, \ldots, i_0$. By means of the same reasoning as above we can prove:

$$R(h_{i-1}) > R(h_i), \ i = 1, \ldots, i_0.$$

Hence,

$$R(h) > R(h_1) > \ldots > R(h_{i_0}) = R(s);$$

which proves theorem 3.

Definition 7: The partial integral-quotient of two formulae "h_1" and "h_2", denoted by "$piq(h_1,h_2)$",is defined as follows:
If either $h_1 = 0$ or $h_2 = 0$, then $piq(h_1,h_2) = 0$; if, on the other hand, $h_1 \neq 0$ and $h_2 \neq 0$, then we consider the quotient "q":

$$q = lt(h_1) \times (lt(h_2))^{-1}.$$

If "q" contains one or more factors of "$lt(h_2)$", with a negative exponential, then $piq(h_1,h_2) = 0$; otherwise, $piq(h_1,h_2) = q$.

Remarks:

1. If $piq(h_1,h_2) \neq 0$ then

$$l(piq(h_1,h_2)) = l(h_1) - l(h_2). \qquad (2.11.24)$$

2. If "h_1" and "h_2" are admissible, then

$piq(h_1,h_2) =$

if $h_1 = 0 \vee h_2 = 0 \vee (\exists i_1,\ 1 \leq i_1 \leq m,\ : p_{i_1}(h_1) < p_{i_1}(h_2))$

then 0,

$$\text{otherwise} \quad \frac{H_1[1,0,1]}{H_2[1,0,1]} \times \prod_{i=1}^{m} a_i \uparrow (p_i(h_1) - p_i(h_2)). \qquad (2.11.25)$$

3. If "h_1" and "h_2" are admissible and $piq(h_1,h_2) \neq 0$, then

$$R(piq(h_1,h_2)) = R(h_1) - R(h_2). \qquad (2.11.26)$$

Definition 8: The integral-quotient sequences $\{q_n\}$ and $\{r_n\}$, $n = 0,1,2,\ldots$, for two formulae "h_1" and "h_2", are defined by:

$$q_0 = 1, \qquad r_0 = h_1,$$
$$q_n = piq(r_{n-1},h_2),\ r_n = r_{n-1} - q_n \times h_2,\ n > 0. \qquad (2.11.27)$$

Theorem 4: The integral-quotient sequences of definition 8 for two admissible formulae "h_1" and "h_2", have the property:

if $q_n = 0$, then $r_n = r_{n-1}$ $(n > 0)$;

if $q_n \neq 0$, $n > 0$, then,

either $l(r_n) < l(r_{n-1})$,

or $l(r_n) = l(r_{n-1})$ and in this case,

either $R(r_n) < R(r_{n-1})$,

or $R(r_n) = R(r_{n-1})$ and then $l(r_n) = 0$.

Proof: From (2.11.27) it follows immediately that $q_n = 0$ implies $r_n = r_{n-1}$. Assume now that $q_n \neq 0$. From (2.11.27) it follows that

$$lt(r_n) = lt(st(r_{n-1})) \vee lt(r_n) = q_n \times lt(st(h_2)).$$

Note that we are not interested in numerical factors; these may be taken into account by the stub of "r_{n-1}" or of "h_1".

From (2.11.24) and theorem 2 it follows that

$$l(r_n) \leq l(r_{n-1}). \qquad (2.11.28)$$

From this it follows that all "r_n" are admissible. We now assume that $l(r_n) = l(r_{n-1})$. Using (2.11.26) and theorem 3 it follows that, if $lt(r_n) = lt(st(r_{n-1}))$, either "$r_n$" and "$r_{n-1}$" are numbers or $R(r_n) < R(r_{n-1})$; if, on the other hand, $lt(r_n) = q_n \times lt(st(h_2))$ then $R(r_n) = R(r_{n-1}) - R(h_2) + R(st(h_2))$; hence, either "$h_2$" and "$st(h_2)$" are numbers, or $R(r_n) < R(r_{n-1})$. But, if "h_2" is a number then $q_n = 0$ for $n \geq 1$; therefore, this case does not need to be considered.

Theorem 5: The integral-quotient sequences $\{q_n\}$ and $\{r_n\}$, $n = 0,1,2,\ldots$, for two admissible formulae "h_1" and "h_2" have the property that there exists an integral number M such that $q_{M+1} = 0$ and $q_M \neq 0$.

Proof: From theorem 4 it follows that either $l(r_n)$ decreases or, if it stays constant, then $R(r_n)$ decreases, or if $R(r_n)$ stays constant then "r_n" is a number and then $q_{n+2} = 0$.

From this observation it follows that an upper bound for M is given by the number

$$l(h_1) \times B,$$

where

$$B = \sum_{i=1}^{m} 2^{2Q \times (A-a_i^*)} P. \qquad (2.11.29)$$

Definition 9: The integral quotient and the remainder of two admissible formulae "h_1" and "h_2", which are denoted by "$iq(h_1,h_2)$" and "$rem(h_1,h_2)$", respectively, are defined as follows:

Let $\{q_n\}$ and $\{r_n\}$, $n = 0,1,2,\ldots$, be the integral-quotient sequences of "h_1" and "h_2"; let the integral number M be such that $q_{M+1} = 0$ and $q_M \neq 0$, then

$$iq(h_1,h_2) = \sum_{n=1}^{M} q_n \quad \text{and} \quad rem(h_1,h_2) = r_M. \tag{2.11.30}$$

Theorem 6. If "h_1" and "h_2" are admissible formulae, then

$$h_1 = iq(h_1,h_2) \times h_2 + rem(h_1,h_2). \tag{2.11.31}$$

Proof:

$$\begin{aligned} rem(h_1,h_2) &= r_{M-1} - q_M \times h_2 = \\ &= r_{M-2} - (q_M + q_{M-1}) \times h_2 = \\ &\text{-----------------------} \\ &= h_1 - \left(\sum_{n=1}^{M} q_n\right) \times h_2. \end{aligned}$$

Theorem 7: If "h_3" and "h_2" are admissible formulae; if $h_2 \neq 0$ and $h_1 = h_3 \times h_2$, where "h_1" is also admissible, then

$$iq(h_1,h_2) = h_3$$

and

$$rem(h_1,h_2) = 0.$$

Proof: Consider the integral-quotient sequences $\{q_n\}$ and $\{r_n\}$, $n = 0,1,2,\ldots$, for "h_1" and "h_2"; then

$$q_0 = 1, \qquad r_0 = h_1,$$

and from $lt(h_3 \times h_2) = lt(h_3) \times lt(h_2)$, it follows

$$\begin{aligned} q_1 &= piq(h_1,h_2) = \\ &= lt(h_3), \end{aligned}$$

and

$$r_1 = h_1 - q_1 \times h_2 = \{h_3 - lt(h_3)\} \times h_2.$$

If $r_1 = 0$, then $q_2 = 0$ and $rem(h_1,h_2) = r_1 = 0$.
If $r_1 \neq 0$, then the quantities q_2 and r_2 are considered:

$$q_2 = lt(h_3 - lt(h_3))$$

and

$$r_2 = \{h_3 - lt(h_3) - lt(h_3 - lt(h_3))\} \times h_2.$$

If $r_2 = 0$ then we have again $\text{rem}(h_1,h_2) = 0$.
If $r_2 \neq 0$ then we consider r_3;
this process is repeated until $q_{n+1} = 0$.
$q_{n+1} = 0$ implies

$$\text{lt}(h_3 - \text{lt}(h_3) - \text{lt}(h_3 - \text{lt}(h_3)) \ldots) = 0,$$

hence, $$r_n = \{h_3 - \text{lt}(h_3) - \ldots\} \times h_2 = 0.$$

Therefore, $\text{rem}(h_1,h_2) = 0$.
And from theorem 6 we conclude

$$\text{iq}(h_1,h_2) = h_3.$$

We are now ready to define the effect of a call:

$$q := QUOTIENT(g,f,r),$$

namely:

$$q := \text{iq}(g,f) \quad \text{and} \quad r := \text{rem}(g,f).$$

2.11.2. The procedure *QUOTIENT*

The procedure *QUOTIENT* is now reproduced.

```
integer procedure QUOTIENT(g,f,remainder); value g,f;
integer g,f,remainder;
begin integer i,j,k,lf,lg,quotient; Boolean first; integer array A[0:1],
  B[1:1,1:1]; FIX; CONVERT(f,B,A,true);
  begin integer array F[1:A[0],0:A[1]],LF[0:A[0]];
```

```
integer procedure QUOT(g1); value g1; integer g1;
begin integer factor; CONVERT(g1,B,A,true);
  begin integer array G[1:A[0],0:A[1]],LG[0:A[0]],GG[1:A[1]];
    CONVERT(g1,G,LG,false); SIMPL2REPR(G,LG);
    if LG[0] = 0 then goto ZERO;
    if first then
    begin first:= false; for i:= 1 step 1 until LG[0] do
      begin if TYPE(T(G,i,LG[i],1),j,k) = function ∧
        j = expf then goto UNDEFINED
      end
    end;
    lg:= 0; for i:= 1 step 1 until LG[1] do
    begin lg:= lg + T(G,1,i,2); GG[i]:= T(G,1,i,2) end;
    if lg < lf then goto ZERO; lg:= lf; k:= 1;
    for i:= 1 step 1 until LF[1] do
    begin for j:= k step 1 until LG[1] do
      begin if T(F,1,i,1) = T(G,1,j,1) then
        begin if T(F,1,i,2) ≤ T(G,1,j,2) then
          begin GG[j]:= T(G,1,j,2) - T(F,1,i,2);
            lg:= lg - T(F,1,i,2)
          end
          else goto ZERO; k:= j + 1; goto AA
        end
        else if T(F,1,i,1) < T(G,1,j,1) then goto ZERO
      end;
    AA: end; if lg > 0 then goto ZERO;
    factor:= Q(G[1,0],F[1,0]);
    for i:= 1 step 1 until LG[1] do
    factor:= P(factor,INT POW(T(G,1,i,1),GG[i]));
    goto NEXT STEP;
```

```
      ZERO: QUOT:= zero; remainder:= zero;
        for i:= 1 step 1 until LG[0] do
        begin k:= G[i,0]; for j:= 1 step 1 until LG[i] do
          k:= P(k,INT POW(T(G,i,j,1),T(G,i,j,2)));
          remainder:= S(remainder,k)
        end;
        goto OUT
      end;
    NEXT STEP: QUOT:= S(factor,QUOT(D(g1,P(factor,f))));
    OUT:
    end QUOT;

    if ¬ expand then goto UNDEFINED;
    CONVERT(f,F,LF,false); SIMPL2REPR(F,LF);
    if LF[0] = 0 ∨ LF[1] = 0 then goto UNDEFINED;
    for i:= 1 step 1 until LF[0] do
    begin if TYPE(T(F,i,LF[i],1),j,k) = function ∧ j = expf then    *)
      goto UNDEFINED
    end;
    lf:= 0; for i:= 1 step 1 until LF[1] do lf:= lf + T(F,1,i,2);
    first:= true; quotient:= QUOT(g);
    A[0]:= quotient; A[1]:= remainder;
    ERASE BUT RETAIN(i,0,1,A[i]);
    QUOTIENT:= A[0]; remainder:= A[1];
    goto END;
  UNDEFINED: QUOTIENT:= zero; remainder:= g; ERASE;
  END: end
end QUOTIENT;
```

*) Note that *j* obtains a value by means of *TYPE*.

The actions of *QUOTIENT* are briefly described:

1. The formula f is represented in the arrays F and LF; by virtue of a call of *SIMPL 2 REPR*, the array elements of F and LF satisfy the conditions for the standard form.
2. If *expand* has the value *false*, then the process leads to an end and *QUOTIENT* and *remainder* become equal to *zero* and g, respectively.
3. If there occur exponential functions in f, the process also terminates.
4. The length of f is calculated and assigned to lf.
5. The Boolean variable *first* becomes *true*.
6. The statement *quotient*:= *QUOT(g)* has as effect that *quotient*:= iq(g,f) and *remainder*:= rem(g,f).
7. By means of an "erase but retain statement", intermediate but uninteresting, results are erased.
8. *QUOTIENT* and *remainder* get their final values.

Before we describe the procedure *QUOT*, constituting the main part of *QUOTIENT*, we shall give another definition of the integral quotient of the formulae g and f, by means of the function "quot" defined as:

quot($g1$):= if piq($g1$,f) = *zero* then *zero* else

piq($g1$,f) + quot($g1$ - piq($g1$,f) $\times$ f).

Then the integral quotient is defined as "quot(g)".

It is easily seen that this definition is equivalent with definition 9. The effect of *QUOT* can now be described:

QUOT(g1) = quot($g1$).

The actions of *QUOT* are:

1. The formula $g1$ is represented in the arrays G and LG; by virtue of a call of *SIMPL 2 REPR*, the array elements of G and LG satisfy the conditions for the standard form.
2. If the nonlocal Boolean variable *first* has the value *true*, it is investigated whether there occur exponential functions in the actual parameter $g1$ (which is the formula g of *QUOTIENT*). Moreover, *first* gets the value *false*.

3. The length of *g1* is calculated and assigned to *lg*.
4. It is investigated whether piq(*g1*,*f*) = *zero*; if so, the process terminates by a jump to the label *ZERO*, where *QUOT* and *remainder* get the values *zero* and r_M, respectively. (r_M is defined in definition 9).
5. If piq(*g1*,*f*) ≠ *zero*, then the process continues, after a jump to the label *NEXT STEP*, by means of the statement

   QUOT:= S(factor,QUOT(D(g1,P(factor,f)))),

   in which *factor* = piq(*g1*,*f*).

2.11.3. The common-divisor process

Before the common-divisor process can be discussed, it is necessary to give some theorems and definitions.

Theorem 8: Let "h_1" and "h_2" be admissible formulae; then $h_3 = \text{rem}(h_1,h_2)$ is also an admissible formula; moreover, $l(h_3) \le l(h_1)$ and if $l(h_3) = l(h_1)$ then $R(h_3) \le R(h_1)$; if, furthermore, $R(h_3) = R(h_1)$ then $h_3 = h_1$ or $h_3 = 0$.

Proof: Consider the integral-quotient sequences $\{q_n\}$ and $\{r_n\}$, $n = 0,1,2,\ldots$, for "h_1" and "h_2".

$$q_0 = 1, \qquad r_0 = h_1,$$
$$q_n = \text{piq}(r_{n-1},h_2)\ , \quad r_n = r_{n-1} - q_n \times h_2,\ n = 1,\ \ldots,\ M,$$
$$q_{M+1} = 0.$$

From theorem 4 it follows that

$$l(h_3) = l(r_M) \le \ldots \le l(h_1);$$

hence, "h_3" is an admissible formula.

Let $l(h_3) = l(h_1)$.
If $M = 0$, then $h_3 = h_1$; we therefore assume that $M > 0$.
From theorem 4 and

$$l(h_3) = l(r_M) = \ldots = l(h_1).$$

it follows that either $R(h_3) < R(h_1)$, or $R(h_3) = R(h_1)$ and "h_3", "r_M", ..., "r_1" and "h_1" are numbers.

But, if "r_1" and "h_1" are numbers, then either $r_1 = 0$ (if "h_2" is also a number) or $r_1 = h_1$; if $r_1 = 0$ then $q_2 = 0$ and $h_3 = 0$; if $r_1 = h_1$ then $q_1 = 0$ and $h_3 = h_1$.

This proves the theorem.

Definition 10: The common-divisor sequences $\{Q_n\}$ and $\{f_n\}$, $n = 1,2,3,\ldots$, for two admissible formulae "h_1" and "h_2" are given as:

$$f_1 = h_1 \ , \quad f_2 = h_2, \qquad (2.11.32)$$

$$Q_n = iq(f_n, f_{n+1}), \ f_{n+2} = rem(f_n, f_{n+1}).$$

Remark: Theorem 8 asserts that this definition is meaningful.

Theorem 9: The common divisor sequences $\{Q_n\}$ and $\{f_n\}$, $n = 1,2,3,\ldots$, for two admissible formulae "h_1" and "h_2" have the property that there exists an integral number $N \geq 0$, such that

$$f_{n+2} = f_n \quad \text{and} \quad Q_n = 0 \text{ for } n \geq N.$$

Proof: Theorem 8 gives:

Either $l(f_{n+2}) < l(f_n)$,

or $l(f_{n+2}) = l(f_n)$ and in this case,

either $R(f_{n+2}) < R(f_n)$,

or $R(f_{n+2}) = R(f_n)$ and in this case,

either $f_{n+2} = f_n$, or $f_{n+2} = 0$.

Let n_0 be such that

$$l(f_{n_0+2}) = l(f_{n_0}) \wedge R(f_{n_0+2}) = R(f_{n_0}) \wedge$$

$$l(f_{n_0+3}) = l(f_{n_0+1}) \wedge R(f_{n_0+3}) = R(f_{n_0+1}).$$

The existence of n_0 follows from

$$n_0 \leq \{l(h_1) + l(h_2)\} \times B,$$

where B is defined in (2.11.29).

Hence, either $f_{n_0+2} = f_{n_0}$ or $f_{n_0+2} = 0$; and either $f_{n_0+3} = f_{n_0+1}$ or $f_{n_0+3} = 0$.

Let us consider the case $f_{n_0+2} = 0$; then $f_{n_0+3} = \text{rem}(f_{n_0+1}, 0) = f_{n_0+1}$ and $f_{n_0+4} = \text{rem}(0, f_{n_0+3}) = 0$.

Therefore, N may be chosen equal to n_0+1 in this case.

In the same way we have in the case $f_{n_0+3} = 0$, that N may be chosen equal to n_0+2.

We, finally, consider the case that

$$f_{n_0+2} = f_{n_0} \wedge f_{n_0+3} = f_{n_0+1}.$$

Evidently, $f_{n_0+4} = \text{rem}(f_{n_0}, f_{n_0+1}) = f_{n_0+2}$

and $f_{n_0+5} = \text{rem}(f_{n_0+1}, f_{n_0+2}) = f_{n_0+3}$;

hence, N may be chosen equal to n_0.

The statement that $Q_n = 0$ if $f_{n+2} = f_n$, is an immediate consequence of the definition of f_{n+2}.

We have thus proved theorem 9.

Definition 11: The common divisor of two admissible formulae "h_1" and "h_2", denoted by "$\text{gcd}(h_1,h_2)$", is defined by means of:

Let $\{Q_n\}$ and $\{f_n\}$, $n = 1,2,3,\ldots$, be the common-divisor sequences for "h_1" and "h_2"; let N be the smallest integer ≥ 1 with the property that either,

$$Q_N = 0 \wedge Q_{N-1} = 0 \wedge N \geq 2, \tag{2.11.33}$$

or,

$$f_{N+2} = 0. \tag{2.11.34}$$

If $Q_N = 0 \wedge Q_{N-1} = 0$, or f_{N+1} is a number then

$$\text{gcd}(h_1,h_2) = 1;$$

otherwise,

$$\text{gcd}(h_1,h_2) = f_{N+1}.$$

Remark: If $f_{N+2} = 0$, then it is not possible that $Q_N = 0$, unless $N = 1$; for, if $Q_N = 0$, then $f_N = f_{N+2} + Q_N \times f_{N+1} = f_{N+2} = 0$ and N would not be the smallest integral number for which $f_{n+2} = 0$. In the case that $N = 1$, only condition (2.11.34) is verified.

Theorem 10: Let "h_1" and "h_2" be admissible formulae; let $h_3 = \gcd(h_1,h_2) \neq 1$; then

$$\mathrm{rem}(h_1,h_3) = \mathrm{rem}(h_2,h_3) = 0.$$

Moreover, any admissible formula "h_4", for which,

$$\mathrm{rem}(h_1,h_4) = 0 \wedge \mathrm{rem}(h_2,h_4) = 0,$$

satisfies:

$$\mathrm{rem}(h_3,h_4) = 0.$$

Proof: Consider the common-divisor sequences $\{Q_n\}$ and $\{f_n\}$, $n = 1,2,3,\ldots$, for "h_1" and "h_2".

Then, from $\gcd(h_1,h_2) \neq 1$, it follows that there exists an $N \geq 1$ such that

$$f_{N+2} = 0 \quad \text{and} \quad f_{N+1} = h_3.$$

Hence,

$$\begin{aligned}
f_N &= Q_N \times f_{N+1} &&= q_N \times h_3, \\
f_{N-1} &= Q_{N-1} \times f_N + f_{N+1} &&= q_{N-1} \times h_3, \\
&\text{-----------------------------------} \\
h_2 = f_2 &= Q_2 \times f_3 + f_4 &&= q_2 \times h_3, \\
h_1 = f_1 &= Q_1 \times f_2 + f_3 &&= q_1 \times h_3,
\end{aligned}$$

where, if $q_{N+2} = 0$ and $q_{N+1} = 1$, $q_{n-1} = Q_{n-1}q_n + q_{n+1}$, for $n = N,N-1,\ldots,2$.

From theorem 7 we have, therefore,

$$\mathrm{rem}(h_1,h_3) = \mathrm{rem}(h_2,h_3) = 0.$$

If $h_1 = a \times h_4$ and $h_2 = b \times h_4$, then

$$f_3 = f_1 - Q_1 \times f_2 = (a - Q_1 \times b) \times h_4 = \bar{q}_3 \times h_4$$

$$f_4 = f_2 - Q_2 \times f_3 = (b - Q_2 \times \bar{q}_3) \times h_4 = \bar{q}_4 \times h_4$$

$$h_3 = f_{N+1} = f_{N-1} - Q_{N-1} \times f_N = (\bar{q}_{N-1} - Q_{N-1} \times \bar{q}_N) \times h_4 = \bar{q}_{N+1} \times h_4$$

where $\bar{q}_{n+1} = \bar{q}_{n-1} - Q_{n-1} \times \bar{q}_n$, $n = 2, 3, \ldots, N$ and again we have from theorem 7 that

$$\text{rem}(h_3, h_4) = 0.$$

Remarks:

1. The condition $\gcd(h_1,h_2) \neq 1$ is necessary in theorem 10. An example, where $\gcd(h_1,h_2) = 1$ and where "h_4" ($\neq 1$) exists with the properties $\text{rem}(h_1,h_4) = 0$ and $\text{rem}(h_2,h_4) = 0$, is given by:

   $$h_1 = x^2 - y^2,$$

   $$h_2 = x^2 - 2xy + y^2,$$

   $$h_4 = x - y.$$

2. From theorem 10 we may conclude that if a common divisor is found, unequal to 1, then it is the greatest; if, on the other hand, the common divisor is equal to 1 then it is not necessarily the greatest; i.e. there may be another common divisor, not divisible by the first one.

We are now ready to introduce the procedure *COMMON DIVISOR*.

A call: *h:= COMMON DIVISOR(f,g)*

has as effect:

$$h := \gcd(f,g).$$

```
integer procedure COMMON DIVISOR(f,g); value f,g; integer f,g;
begin integer gcd,q,r; Boolean s,s1;
  procedure GCD(f1,f2,f3); value f1,f2; integer f1,f2,f3;
  begin integer f4; q:= QUOTIENT(f1,f2,f3); s:= q ≠ zero;
    if ¬ s1 ∧ ¬ s then goto UNDEFINED;
    if f3 = zero then
    begin gcd:= f2; goto if TYPE(gcd,di,di) = number then
      UNDEFINED else OK
    end; s1:= s; GCD(f2,f3,f4)
  end;
  FIX; s1:= true; GCD(f,g,r);
OK: ERASE BUT RETAIN(q,1,1,gcd); COMMON DIVISOR:= gcd; goto END;
UNDEFINED: COMMON DIVISOR:= one; ERASE;
END: end COMMON DIVISOR;
```

COMMON DIVISOR is now briefly explained:

1. The two Boolean variables *s* and *s1* are used to test whether $Q_n = 0 \wedge Q_{n-1} = 0$;
2. the procedure *GCD* has to execute the calculations as given by (2.11.32); its parameters *f1*, *f2* and *f3* correspond to f_n, f_{n+1} and f_{n+2}; by means of a recursive call, $GCD(f_2, f_3, f_4)$, the process is repeated until either condition (2.11.33) or condition (2.11.34) is satisfied.

Remarks:

1. *COMMON DIVISOR* delivers the gcd(f,g), apart from a numerical factor (all numbers are considerd as "unities").
2. If f and g are polynomials in one algebraic variable, then *COMMON DIVISOR* delivers the greatest common-divisor such as it is defined usually; see e.g. Van der Waerden [26], p. 65.
3. If f and g are polynomials in two or more algebraic variables, then *COMMON DIVISOR* finds the greatest common-divisor if either f is a factor of g or g is a factor of f.

4. If f and g are polynomials in two or more algebraic variables, but f is not a factor of g and g is not a factor of f, then *COMMON DIVISOR* may not be successful in finding the greatest common-divisor, if this differs from "1". For example, the greatest common-divisor "$x+y$" of "x^2-y^2" and "$x^2+2xy+y^2$" will not be found; instead, *COMMON DIVISOR* finds "1" as gcd.

Final remark: In the beginning of this section we excluded the case that f or g contains exponential functions.
This is necessary since, otherwise, the processes, as defined in this section, can not be shown to terminate.
(One should try to find the common divisor of e.g.

$$e^{2x} - e^{2y} \quad \text{and} \quad e^{y} + e^{x},$$

by means of definition 10).

2.12. Supplementary equipment

First, the procedure *DER* is reproduced which calculates the derivative of a formula *f* with respect to the algebraic variable *x*.
The formula *f* may be given in expanded as well as in not-expanded form.

```
integer procedure DER(f,x); value f,x; integer f,x;
begin integer t,a,b; t:= TYPE(f,a,b);
  if f = x then DER:= one else
  if t = sum then DER:= S(DER(a,x),DER(b,x)) else
  if t = difference then DER:= D(DER(a,x),DER(b,x)) else
  if t = product then DER:= S(P(DER(a,x),b),P(a,DER(b,x))) else
  if t = quotient then
  begin integer da,db; da:= DER(a,x); db:= DER(b,x);
    DER:= if db = zero then Q(da,b) else Q(D(P(da,b),P(a,db)),P(b,b))
  end else
  if t = power then DER:= P(f,DER(P(b,LN(a)),x)) else
  if t = integral power then
  begin integer d; d:= DER(b,x); DER:= if d = zero then zero else
    P(IN(a),P(INT POW(b,a-1),d))
  end else
  if t = tr pow series then
  begin integer i,y; integer array coeff[0:a]; COEFFICIENT(f,a,y,coeff);
    DER:= S(TPS(i,a,y,DER(coeff[i],x)), if x ≠ y ∨ a = 0 then zero else
      TPS(i,a-1,y,P(IN(i+1),coeff[i+1])))
  end else
  if t = function then
  begin integer d; d:= DER(b,x);
    DER:= if d = zero then d else
      if a = expf then P(f,d) else
      if a = lnf then P(INT POW(b,-1),d) else
      if a = sinf then P(COS(b),d) else
      if a = cosf then P(min one,P(SIN(b),d)) else
      if a = arctanf then P(Q(one,S(one,P(b,b))),d) else
      P(Q(RN(.5),f),d)
  end else DER:= zero;
END: end DER;
```

The procedure to be given next, is *COPY*, which is used in the complex conjugation process and in the substitution process. Note that *COPY* builds up a new formula only, if this is necessary.

```
integer procedure COPY(f,F SPECIAL); value f; integer f;
Boolean procedure F SPECIAL;
begin integer t,a,b,a1,b1; if F SPECIAL(f) then
  begin COPY:= f; goto END end; t:= TYPE(f,a,b);
  if t = sum ∨ t = difference ∨ t = product ∨ t = quotient
  ∨ t = power then
  begin a1:= COPY(a,F SPECIAL); b1:= COPY(b,F SPECIAL);
    COPY:= if a = a1 ∧ b = b1 then f else
    if t = sum then S(a1,b1) else if t = difference
    then D(a1,b1) else if t = product then P(a1,b1) else
    if t = quotient then Q(a1,b1) else POWER(a1,b1)
  end else if t = function ∨ t = integral power then
  begin b1:= COPY(b,F SPECIAL); COPY:= if b1 = b then f else
    if t = integral power then INT POW(b1,a) else if a = expf
    then EXP(b1) else if a = lnf then LN(b1) else if a = sinf
    then SIN(b1) else if a = cosf then COS(b1) else if a =
    arctanf then ARCTAN(b1) else SQRT(b1)
  end else
  if t = tr pow series then
  begin integer i,x,x1; integer array co,co1[0:a]; Boolean B;
    COEFFICIENT(f,a,x,co); B:= true;
    for i:= 0 step 1 until a do
    begin  co1[i]:= COPY(co[i],F SPECIAL); B:= B ∧ co1[i] = co[i] end;
    x1:= COPY(x,F SPECIAL); COPY:= if B ∧ x1 = x then f else
    TPS(i,a,x1,co1[i])
  end else COPY:= f;
END: end COPY;
```

The next procedure to be reproduced is *CC*, which becomes equal to the complex conjugate of a formula *f*. It is assumed that all algebraic variables are real algebraic-variables.
Thus, if one wants to use a complex variable "z", one should use "x + iy" instead.
If one insists on using complex variables within the general system, then the procedure *CHANGE* should be changed in such a way that it defines the complex conjugate not only of a number but also of an algebraic variable. This implies that there should exist,besides the algebraic variables, their complex conjugates which may be given by the lhs or rhs quantities of their internal representation.

```
integer procedure CC(f); value f; integer f;
begin Boolean procedure CHANGE(g); integer g;
  begin integer a,b; if TYPE(g,a,b) = number ∧ a = complex then   *)
    begin real r,i; VAL OF NUM(g,r,i); g:= CN(r,-i); CHANGE:= true
    end else CHANGE:= false
  end;
  CC:= COPY(f,CHANGE)
end CC;
```

Finally, the procedure *SUBSTITUTE* is reproduced. *SUBSTITUTE* becomes equal to a formula *f* in which all constituants, given by *argument i*, *i* = *lb*, ..., *ub*, are changed into the formulae *value i*, *i* = *lb*, ..., *ub*. If, in particular, *f* happens to be a truncated power-series in "x" and "x" occurs as one of the arguments, then *f* is transformed into an extended sum in which "x" is replaced by the value of the corresponding argument.

*) Note that *a* obtains a value by means of *TYPE*.

Suppose that the truncated power-series "p(x)" is an approximation of the function "f(x)", then substitution of an algebraic variable "a" into "x", results in a truncated power series in the variable "a", which is an approximation of "f(a)"; substitution of a truncated power-series "q(y)" into "x" results in a truncated power-series in the variable "y", which is an approximation of "f(q(y))"; and substitution of a formula "g", which is neither an algebraic variable nor a truncated power-series, results in a formula, which is not a truncated power-series; this formula is an approximation of "f(g)".
Thus, if "p(x)" has numbers as coefficients, then a substitution of a number "c" into "x", results in a number which is an approximation of "f(c)".

The substitution process may be called parallel; i.e. the result is independent of the order of the variables to be substituted.

Example: let $f = x^2/y$, $arg[0] = x$, $val[0] = y$,
$arg[1] = y$ and $val[1] = x$,
then
SUBSTITUTE(f,i,0,1,arg[i],val[i])
results in y^2/x, and
SUBSTITUTE(f,i,0,1,arg[1-i],val[i-1])
results also in y^2/x.
On the other hand,
SUBSTITUTE(SUBSTITUTE(f,i,0,0,x,y),j,0,0,y,x)
results in x^2/x, which becomes x, if the system is in the expanding state, and
SUBSTITUTE(SUBSTITUTE(f,i,0,0,y,x),j,0,0,x,y)
results in y^2/y, which becomes y, if the system is in the expanding state.

```
integer procedure SUBSTITUTE(f,i,lb,ub,argument i,value i);
value f,lb,ub; integer f,i,lb,ub,argument i,value i;
begin integer array argument,value[lb:ub];
  Boolean procedure SUBST(g); integer g;
  begin integer a,b,i,j; if TYPE(g,a,b) = tr pow series then
    begin integer x; integer array coeff[0:a];
      COEFFICIENT(g,a,x,coeff);
      for i:= lb step 1 until ub do
      begin if x = argument[i] then
        begin if TYPE(value[i],j,j) = algebraic variable then g:=
          TPS(j,a,value[i],SUBSTITUTE(coeff[j],b,lb,ub,argument[b],
          value[b])) else
          begin g:= zero; for b:= a step -1 until 0 do
            g:= S(P(g,value[i]),SUBSTITUTE(coeff[b],j,lb,ub,
              argument[j],value[j]))
          end; SUBST:= true; goto END
    end end end; for i:= lb step 1 until ub do
    begin if g = argument[i] then
      begin g:= value[i]; SUBST:= true; goto END end
    end; SUBST:= false;
  END: end;
  for i:= lb step 1 until ub do
  begin argument[i]:= argument i; value[i]:= value i end;
  SUBSTITUTE:= COPY(f,SUBST)
end SUBSTITUTE;
```

2.13. Output

By means of a simple actual program (see section 2.2), the procedures *OUTPUT* and *OUTPUT VARIABLE* will be introduced.
An output procedure which gives much more elegant results, i.e. less brackets and less plus and minus signs, is given in chapter 3.
The reason why the output procedure is not given as part of the general system is, that it is strongly dependent on the particular formula manipulations to be performed.

The actual program as reproduced in this section solves the following problem:

let $s_1 = 1 + a_1x + a_2x^2 + \dots + a_5x^5 + O(x^6)$;

let $s_3 = 1 + \ln(s_1) = 1 + c_1x + \dots + c_5x^5 + O(x^6)$;

Calculate the c_i in terms of the a_i.
The calculation is performed along the following steps:

Step 1: Calculate the first five derivatives $d_1, \dots, d_5$ of $f(s_1) = 1 + \ln(s_1)$, with respect to s_1.

Step 2: Substitute $s_1 = 1$ into the d_i.

Step 3: Calculate the coefficients b_i in $g(y) = 1 + \ln(1 + y) = 1 + b_1y + \dots + b_5y^5 + O(y^6)$ using:

$$b_i = d_i(s_1 = 1)/i!.$$

Step 4: Calculate c_i by substituting $y = s_1 - 1$ into $g(y)$: $g(s_1 - 1) = 1 + \ln(s_1)$.

```
INITIALIZE;
comment RPR 290466/06;
ACTUAL PROGRAM:
begin integer f,g,x,y,i,j,fact; integer array a[1:5],b,d[0:5],s[1:4];
```

```
procedure PR(s); string s; PRINTTEXT(s);

procedure OUTPUT(f); value f; integer f;
begin integer t,a,b; t:= TYPE(f,a,b); if ¬ expand then PR(‹(›);
  if f = zero then PR(‹0›) else if f = one then PR(‹1›) else
  if t = sum ∨ t = difference ∨ t = product ∨ t = quotient ∨
  t = power then
  begin if t = quotient ∧ expand then PR(‹(›); OUTPUT(a);
    if t = sum then PR(‹ + ›) else if t = difference then PR(‹ - ›)
    else if t = product then PR(‹ × ›) else if t = quotient then
    begin if expand then PR(‹)/(›) else PR(‹ / ›) end else
    PR(‹ ↑ ›); OUTPUT(b); if t = quotient ∧ expand then PR(‹)›)
  end else
  if t = function then
  begin if a = expf then PR(‹exp(›) else if a = lnf then PR(‹ln(›)
    else if a = sinf then PR(‹sin(›) else if a = cosf then PR(‹cos(›)
    else if a = arctanf then PR(‹arctan(›) else PR(‹sqrt(›);
    OUTPUT(b); PR(‹)›)
  end else
  if t = integral power then
  begin PR(‹(›); OUTPUT(b); PR(‹)↑›); ABSFIXT(2,0,a) end else
  if t = number then
  begin integer i; real ra,ia; PR(‹(›);
    if a = integer then
    begin VAL OF INT NUM(f,i); FIXT(entier(ln(abs(i))×.4343
      + 1),0,i)
    end else
    if a = real then
    begin VAL OF REAL NUM(f,ra); FLOT(12,3,ra) end else
    begin VAL OF COMP NUM(f,ra,ia); FLOT(12,3,ra); PR(‹,›);
      FLOT(12,3,ia)
    end; PR(‹)›)
  end else
```

```
  if t = tr pow series then
  begin integer i,x; integer array coeff[0:a];
    COEFFICIENT(f,a,x,coeff); for i:= 0 step 1 until a do
    begin PR(‹(›); OUTPUT(coeff[i]); PR(‹) × (›); OUTPUT(x);
      PR(‹)↑›); ABSFIXT(2,0,i); if i < a then PR(‹ + ›)
    end
  end else OUTPUT VARIABLE(f);
  if ¬ expand then PR(‹)›)
end OUTPUT;

procedure OUTPUT VARIABLE(f); value f; integer f;
  comment The structure of the following procedure body could be
    made more simple. This structure, however, indicates how the
    lhs and rhs quantities of an algebraic variable can be used
    in a case where a large number of algebraic variables occur;
begin integer i,lhs,rhs; switch CASE:= X,Y,S1;
  procedure A(g,s); integer g; string s;
  if f = 0 then
  begin i:= i + 1; g:= STORE(1,algebraic variable,i) end
  else begin PR(s); goto END end;
  if f = 0 then
  begin for i:= 1,2,3,4,5 do a[i]:= STORE(2,algebraic variable,i);
    i:= 0; goto CASE[1]
  end else
  begin TYPE(f,lhs,rhs); if lhs = 1 then goto CASE[rhs] else
    begin PR(‹a[›); ABSFIXT(1,0,rhs); PR(‹]›); goto END end
  end;
X: A(x,‹x›);
Y: A(y,‹y›);
S1: A(s[1],‹s[1]›);
END: end OUTPUT VARIABLE;
```

```
BEGIN OF CALCULATION:
  NLCR; PR(⊰results of calculation RPR 290466/06⊱);
  OUTPUT VARIABLE(0); FIX;
STEP 1: NLCR; PR(⊰s[1] = ⊱);
  FIX; OUTPUT(TPS(i,5,x,if i = 0 then one else a[i])); ERASE;
  FIX; d[0]:= S(one,LN(s[1]));
  NLCR; PR(⊰f(s[1]) = ⊱); OUTPUT(d[0]);
  for i:= 1,2,3,4,5 do d[i]:= DER(d[i-1],s[1]);
STEP 2: for i:= 0,1,2,3,4,5 do d[i]:= SUBSTITUTE(d[i],j,1,1,s[1],one);
STEP 3: fact:= 1; b[0]:= d[0]; b[1]:= d[1];
  for i:= 2,3,4,5 do
  begin fact:= fact × i; b[i]:= Q(d[i],IN(fact)) end;
  ERASE BUT RETAIN(i,0,5,b[i]);
  FIX; g:= TPS(i,5,y,b[i]); NLCR; PR(⊰g(y) = ⊱); OUTPUT(g);
  s[1]:= TPS(i,5,x,if i = 0 then one else a[i]);
  comment The easiest way to perform step 4 would be to use the
    statement: s[3]:= SUBSTITUTE(g,i,1,1,y,D(s[1],one)),
    the following statements are, however, more efficient with
    respect to storage space. The first statement serves to erase
    the truncated power-series g and s[1];
  ERASE;
STEP 4: s[2]:= TPS(i,5,x,if i = 0 then zero else a[i]);
  s[4]:= one; s[3]:= b[0];
  for i:= 1,2,3,4,5 do
  begin FIX; s[4]:= P(s[4],s[2]);
    s[3]:= S(s[3],P(b[i],s[4]));
    ERASE BUT RETAIN(j,3,4,s[j])
  end;
  NLCR; PR(⊰s[3] = ⊱); OUTPUT(s[3])
end; comment the next two ends correspond to the two begins of the
general system;
end end
```

The input tape consists of:

8192 500 0 100 5 $_{10}$-10 $_{10}$-10

The lay-out of the following results should be compared with the lay-out of the same results of Example 6 of section 3.2.7.

results of calculation RPR 290466/06

s[1] = (1) × (x)↑ 0 + (a[1]) × (x)↑ 1 + (a[2]) × (x)↑ 2 +
(a[3]) × (x)↑ 3 + (a[4]) × (x)↑ 4 + (a[5]) × (x)↑ 5

f(s[1]) = 1 + ln(s[1])

g(y) = (1) × (y)↑ 0 + (1) × (y)↑ 1 + ((-.500000000000$_{10}$- 0)) ×
(y)↑ 2 + ((+.333333333333$_{10}$- 0)) × (y)↑ 3 +
((-.250000000000$_{10}$- 0)) × (y)↑ 4 +
((+.200000000000$_{10}$- 0)) × (y)↑ 5

s[3] = (1) × (x)↑ 0 + (a[1]) × (x)↑ 1 + (a[2] +
(-.500000000000$_{10}$- 0) × (a[1])↑ 2) × (x)↑ 2 +
(a[3] + (-1) × a[1] × a[2] + (+.333333333333$_{10}$- 0) ×
(a[1])↑ 3) × (x)↑ 3 + (a[4] + (-1) × a[1] × a[3]
+ (-.500000000000$_{10}$- 0) × (a[2])↑ 2 + (a[1])↑ 2 ×
a[2] + (-.250000000000$_{10}$- 0) × (a[1])↑ 4) × (x)↑ 4
+ (a[5] + (-1) × a[1] × a[4] + (-1) × a[2] × a[3]
+ (a[1])↑ 2 × a[3] + a[1] × (a[2])↑ 2 + (-1) ×
(a[1])↑ 3 × a[2] + (+.200000000000$_{10}$- 0) ×
(a[1])↑ 5) × (x)↑ 5

2.14. Garbage collection

In this section we investigate the consequences of building a formula-manipulation system in ALGOL 60 which implies a garbage collection in order to use the memory space efficiently.

At an arbitrary moment during the formula-manipulation process it may occur that one of the arrays *F*, *FRN*, *FCN1*, *FCN2* and *FTPS*, as described in section 2.4, is completely filled and it becomes necessary for the system to allocate new storage space, if available. In order to do this it has to recognize all the formulae which may be erased. The user has, therefore, to supply this information to the system. This may be done in two ways:

1. Each time the user encounters an erasable formula, he signals this to the system, which attaches some flag to the internal representation of this formula.
   At the moment that the storage space has become exhausted, the system goes through all flagged formulae and their subformulae trying to allocate the new storage space in the places occupied by these formulae; the system has then to solve the following problem: is a subformula of some flagged formula also a subformula of a nonflagged formula, i.e. a formula which may not be erased. This problem cannot be solved by the system unless the user gives more information to the system.

2. During the course of the computations, the user builds up a list R of formulae which may not be erased. If he encounters an erasable formula he removes this formula from R.
   At the moment the system discovers that the storage space has become exhausted, it goes through the internal representations of all formulae of R and their subformulae and attaches flags to these formulae. Then, evidently, all space occupied by nonflagged formulae may be reused for the storage of new formulae.

We concentrate on this second approach.
Again it turns out that there are two possibilities:

1. The formulae of R are restored by the system in a compact way, such that the storage space left is easily localized by the values of the pointers of the arrays *F*, *FRN*, *FCN1*, *FCN2* and *FTPS*.
2. The formulae of R are not restored.

The first approach has the consequence that the integer variables corresponding to the formulae should get new values, namely those values which refer to the newly stored formulae. Hence, the system has to know which integer variable corresponds to which formula. This means that the user should not only provide the system with a list of nonerasable formulae but also with a list of the names of these formulae in which the new locations of the formulae can be stored.
From this it follows that all the formula manipulations should be done via an auxiliary array, called e.g. *LIST*. If the integer variable corresponding to a formula is f, then the value of f should refer to some place, for example, *LIST*[*value of f*] which should refer to the location where the internal representation of that formula is stored.
Evidently, a disadvantage of this approach is that we need a large amount of storage space for the array *LIST*.
Note that as far as this point is concerned a LISP [9] system has no troubles since the formulae are not represented by ordinary variables of integral type but by special variables of "list" type. A double administration is then not necessary.
Let us take the double administration for granted and investigate the way of programming.
Of course, we still want to be able to use expressions like

"*S(S(S(x,S(y,x)),S(x,y)),S(x,S(x,S(y,y))))*".

During the evaluation of this expression, certain subformulae are built up, which should be placed on *LIST*. After these subformulae have been used in forming other formulae they should be removed from *LIST* and the last formula should be put on *LIST*.
Thus the procedure *S* should have the following form:

```
integer procedure S(a,b); value a,b; integer a,b;
begin integer a1,b1; a1:= LIST[a]; b1:= LIST[b];
      REMOVE FROM LIST(a); REMOVE FROM LIST(b);
      S:= SET ON LIST(
          if a1 = LIST[zero] then b1 else
          if b1 = LIST[zero] then a1 else
          STORE(a1,sum,b1))
end S;
```

where *REMOVE FROM LIST* and *SET ON LIST* have obvious meanings.
The question arises whether it is possible to write now e.g. *S(x,S(y,x))*, if *x* and *y* are algebraic variables. The answer is in the negative, since the execution of *S(y,x)* has the effect that both *x* and *y* are removed from *LIST* such that the execution of *S(x,...)* becomes meaningless.
This can only be done correctly if the algebraic variables, and not only these but also the identifiers of formulae, are "packed in" another procedure, say, *TAKE*. *TAKE(x)* should have the effect that the value of *LIST[x]* is placed again at the top of *LIST*. *REMOVE FROM LIST* should then remove the topelement from *LIST*; and we have to change in the procedure *S*:

    REMOVE FROM LIST(a) into *REMOVE FROM LIST*
and *REMOVE FROM LIST(b)* into *REMOVE FROM LIST*

With these procedures we may now write
    S(TAKE(x),S(TAKE(y),TAKE(x)))

We conclude that the first approach, i.e. the system restores the non-erasable formulae in a compact form, results in:
1. a need for more storage space,
2. a complicated way of programming.

Let us now investigate the second approach, where the nonerasable formulae are not restored by the system.

If the formulae to be retained are not restored in a compact form, then one may use a free-list technique.
Due to the occurrence of common subformulae, one should then also use reference counters and decide how many bits these reference counters may occupy.
This seems to be a complicated problem: too many bits is a waste of storage space; too few bits may lead to the problem that one has to copy formulae which are referenced often; (this will in particular be the case for algebraic variables,thereby leading to problems in the simplifying processes).

We decided,therefore,to maintain the simple techniques as described in section 2.7.

Writing the formula manipulation system in the new language ALGOL 68 will not lead to problems concerning storage allocation, since the concept "structure" involves built-in garbage collection.

A study concerning the subject of this chapter is undertaken in depth in [28] as a preparatory study for the language ABC ALGOL, i.e. ALGOL 60 augmented with the type: <u>formula</u>. ABC standing for Algebraïsche Bewerkingen met behulp van de Computer (Dutch for Algebraic Manipulation by means of the Computer).

Chapter 3

THE FORMULA PROGRAM

(A convenient input/output system)

3.1. Introduction

This chapter is written primarily for the user who wants to manipulate formulae without studying the details of chapter 2. In this chapter, the syntax and semantics of a so-called formula program are described. One may define formula manipulations by means of a formula program. The possible formula manipulations are:

1. All the manipulations of the general system, i.e. simplification, power-series manipulation, substitution, etc. (see section 2.1).
2. Solving linear algebraic equations. (This may be used simultaneously with power-series manipulations, in order to calculate series expansions.)
3. Outputting formulae; one may specify two different forms of output:
   a. The normal form, e.g. "a + b × c"; the obtained output may be used for building another formula program.
   b. The "operator" form, e.g. "S(z,P(u,v))"; the obtained output may be used to construct a complex-arithmetic ALGOL 60 program (see chapter 4). The algebraic variables may especially be declared to be of real type; they are output in another way than algebraic variables which are not declared to be of real type. If the formula to be output is e.g."a + a × z × b + b",where "a" and "b" are algebraic variables, declared to be of real type, and "z" an algebraic variable, not declared to be of real type, then the form of the output after an output command is: "SR(a + b,PR(a × b,z))"; thus, the real variables are taken together. Note that the procedures "SR" and "PR" of chapter 4 calculate a sum and a product, respectively,of a real number and a complex number.

4. Differentiation, i.e. the same differentiation as described in chapter 2 with one modification: one may specify the derivatives of a set of formulae.
   For example, suppose one has the formula "f"; if its derivative with respect to "x" is specified to be "fprime", then, after a call of the differentiation operator DER, the derivative of, say, "$f + x^2$" with respect to "x" will be "fprime + 2x".

Section 3.2 contains the definition of a formula program and the instructions for preparing the input tape containing this formula program; moreover, some examples are given.
Section 3.3 contains the "formula-program processor", or "processor", which reads and executes the formulae programs occurring on the input tape. This processor is an ALGOL 60 program containing the general formula-manipulation system described in chapter 2.

3.2. The definition of a formula program

In this section we give the syntax and semantics of a formula program as it may occur on input tape, defining the formula manipulations to be performed.

3.2.1. The syntax of a formula

The main constituents of a formula program are formulae; therefore, we define firstly the syntax of a formula. The metalinguistic variables: letter, digit, integer, and unsigned number are used as defined in [11].

Remark: the definition of formula as given below is only valid for this chapter.

<formula>::= <term>|+<term>|-<term>|<formula>+<term>|<formula>-<term>

<term>::= <factor>|<term>×<factor>|<term>/<factor>

<factor>::= <primary>|<factor>↑<primary>

<primary>::= i|<real number>|(<formula>)|<formula identifier>|
<algebraic variable>|<truncated power-series>|<elementary-function designator>|<derivative>|<simplified formula>|<complex conjugate>|
<result of substitution>|<integral quotient>|<common divisor>

<real number>::= <unsigned number>

<algebraic variable>::= <variable identifier>

<formula identifier>::= <identifier>

<variable identifier>::= <identifier>

<identifier>::= <letter>|<identifier><letter>|<identifier><digit>

<truncated power-series>::= TPS(<algebraic variable>,<formula list>)

<formula list>::= <formula>|<formula list>,<formula>

<elementary-function designator>::= <elementary-function identifier>
(<formula>)

<elementary-function identifier>::= exp|ln|sin|cos|arctan|sqrt

<derivative>::= DER(<formula>,<algebraic variable>)

<simplified formula>::= SIMPL(<formula>)

<complex conjugate>::= CC(<formula>)

<result of substitution>::= SUBST(<formula>,<list of pairs>)

<list of pairs>::= <algebraic variable>,<formula>|<list of pairs>,
<algebraic variable>,<formula>

<integral quotient>::= QUOT(<formula>,<formula>,<remainder identifier>)

<remainder identifier>::= <formula identifier>

<common divisor>::= COMM DIV(<formula>,<formula>)

3.2.2. The semantics of a formula

3.2.2.1. A formula

The form of a formula resembles closely the form of an arithmetic expression in ALGOL 60.
The differences are:
A formula may contain the imaginary unit "i".
A formula may contain truncated power-series and results of substitutions with arbitrary numbers of parameters.
A formula may not contain if clauses and arbitrary function designators.

3.2.2.2. An identifier

An identifier may be built up with no more than 8 letters and/or digits.
Examples: a
a1
a sub 50

Whether an identifier is the identifier of an algebraic variable or is a formula identifier, depends on its first occurrence in the formula program. This will be explained in more detail in section 3.2.4.14.

3.2.2.3. A truncated power-series

Use of truncated power-series (t.p.s.) is only effective if the processor is in the expanding state (see section 3.2.4.5).
A t.p.s. has a variable "x", defined by the algebraic variable, and coefficients "c_i", i = 0, 1, ..., n, defined by the formula list.
The degree of a t.p.s. is equal to n.
If "p_1" and "p_2" are t.p.s.'s, then the result of "$p_1 + p_2$", "$p_1 - p_2$", "$p_1 \times p_2$" and "p_1/p_2" is, in general, another t.p.s.,"p_3", whose degree is equal to the lesser of the degrees of "p_1" and "p_2". The variable of "p_3", if "p_3" is a t.p.s., is equal to the variable of "p_1"; one obtains, therefore, meaningful results only if "p_1" and "p_3" have the same variable.
If "f" is an arbitrary formula, but not a t.p.s., and m is an integral number, then "$p_1 + f$", "$f + p_1$", "$p_1 - f$", "$f - p_1$", "$p_1 \times f$", "$f \times p_1$", "p_1/f", "f/p_1" and "$p_1 \uparrow m$" are, in general, also t.p.s.'s.

If the coefficients "c_i", i = 1, ..., n, of a t.p.s. "p" turn out to be zero, and if, furthermore, "c_0" is not a t.p.s., then "p" is transformed into "c_0" (therefore the above "p_3" is not necessarily a t.p.s.).
The reason for this, mathematically rather incorrect, transformation is that "c_0" occupies less storage space then "$c_0 + 0x + \ldots + 0x^n$".
The reason for requiring "c_0" not to be a t.p.s., becomes clear in the following.

For calculations concerning power series in two or more variables, say "x" and "y", one should consistently use t.p.s.'s in one variable ("x") with as coefficients t.p.s.'s in the other variable ("y"); i.e. one should introduce a certain hierarchy in the variables (which would not be preserved if the above extra condition: ""c_0" is not a t.p.s." was ignored).
Example: "$a00 + a10x + a01y + a20x^2 + a11xy + a02y^2$" should be programmed as:

```
"TPS(x,TPS(y,a00,a01,a02),
       TPS(y,a10,a11),
       TPS(y,a20))".
```

Note that "TPS(y,a20)" might be changed into "a20",yielding the same result.

The above form is to be preferred above the form:

```
"TPS(x,TPS(y,a00,a01,a02),
       TPS(y,a10,a11,0),
       TPS(y,a20,0,0));
```

for, in the last case, the calculations of the meaningless coefficients of "xy^2" will also be performed; moreover, these coefficients will occupy storage space.

3.2.2.4. An elementary-function designator

Let "F(x)" be an elementary-function designator. If "x" is a number then "F(x)" is stored as a number.

If the processor is in the expanding state (see section 3.2.4.5), then the following transformations are carried out:

$\sin(x) \rightarrow \{\exp(ix) - \exp(-ix)\}/(2i)$,

$\cos(x) \rightarrow \{\exp(ix) + \exp(-ix)\}/2$,

$\arctan(x) \rightarrow \ln\{(1 + ix)/(1 - ix)\}/(2i)$,

$\mathrm{sqrt}(x) \rightarrow \exp\{\ln(x)/2\}$.

A call of the simplify operator "SIMPL" or a call of "OUTPUT R",in the expanding state of the processor, involves a simplifying process for the exponential functions.

Therefore, for example, "$\sin^2(x + y) + \cos^2(y + x)$" is simplified into "1".

3.2.2.5. A derivative

The derivative of a formula "f" with respect to the variable "x" is defined as "DER(f,x)".

In combination with a previous "special derivative statement" (see section 3.2.4.10),one can specify the derivatives of certain formulae (even algebraic variables) beforehand.

Example: If the derivative of the algebraic variable "y" has been specified to be "yprime", then

"DER(y ↑ 2 + x ↑ 2, x)" has as result:

"2 × y × yprime + 2 × x".

3.2.2.6. <u>A simplified formula</u>

Simplification is only possible if the processor is in the expanding state (see section 3.2.4.5).
Automatic simplification is performed by the processor for quotients and truncated power-series.
Moreover, a formula to be output by an "OUTPUT R" command is automatically simplified beforehand (this can only be avoided by changing the state of the processor into the not-expanding state).
The use of "SIMPL" is therefore limited to those cases in which the formula to be simplified will not be output immediately afterwards and is not a quotient nor a truncated power-series.

Sections 2.10 and 2.11 describe in a detailed way what simplification means and how it is performed. Roughly stated, simplification means that equal terms in a sum and equal factors in a product are combined into one term and one factor, respectively. Moreover, common factors of the numerator and the denominator of a quotient are, in general, divided out. Finally, trigonometric simplification is performed. Simplification does not mean that a sum is transformed into a product of factors as in $a^2 - b^2 \rightarrow (a + b) \times (a - b)$.

3.2.2.7. <u>A complex conjugate</u>

The complex conjugate of a formula "f" is defined as "CC(f)". The formula which is stored for "CC(f)" is a copy of "f", in which each complex number is changed into its complex conjugate. Note that algebraic variables are not changed into complex-conjugate algebraic variables.
<u>Example</u>: "CC(x + <u>i</u> × y)" is stored as "x − <u>i</u> × y".

3.2.2.8. <u>A result of substitution</u>

Let "f" be a formula and "x_1", ..., "x_n" be some algebraic variables with which "f" is built up. To obtain a new formula,"g", which is equal to "f" in which each "x_i" is changed into a formula,"y_i", i = 1, ..., n,

one may use the "SUBST" operator as follows:

"g:= SUBST(f, x_1, y_1, ..., x_n, y_n)".

The order of the pairs "x_i,y_i" does not affect the result (the substitution process might be called "parallel").
Hence, if "f = x + y + z" and "a = y + z", where "x", "y" and "z" are algebraic variables, then

"g1:= SUBST(f, x, a, y, b, z, c)" has as effect that
"g1 = y + z + b + c";

"g2:= SUBST(f, z, c, y, b, x, a)" has as effect that
"g2 = y + z + b + c";

"g3:= SUBST(SUBST(f, x, a), y, b, z, c)" has as effect that
"g3 = b + c + b + c";

and "g4:= SUBST(SUBST(f, y, b, z, c), x, a)" has as effect that
"g4 = y + z + b + c".

If the formula "f" is a truncated power-series, then the following may happen:
Let

$$f = c_0 + c_1 u + \dots + c_n u^n + O(u^{n+1}).$$

If "u" is amongst the algebraic variables, say, for some i, "u = x_i", then, if "y_i" is an algebraic variable, the result of the substitution is a new truncated power-series with variable "y_i"; if "y_i" is not an algebraic variable, then the result of the substitution is a sum:

"$c_0 + c_1 \times y_i + \dots + c_n \times y_i^n$".

If "u" is not amongst the algebraic variables "x_i", i = 1, ..., n, then the result of the substitution is again a truncated power-series in "u". Note that the substitutions are also performed in the coefficients "c_i".

3.2.2.9. <u>An integral quotient and a common divisor</u>

We refer to section 2.11 for the meaning of integral quotient and common divisor.

<u>Examples</u>: "g:= QUOT(x ↑ 2 + x + 1, x - 1, remainder)" results in "g = x + 2" and "remainder = 3".
"COMM DIV(x ↑ 2 - x + 1, x ↑ 2 - 1)" results in the formula ".5 × x - .5".

3.2.3. The syntax of a formula program

We now proceed with the syntax of a formula program.

<formula program>::= <heading><compound formula-statement>; END;

<heading>::= <any sequence of symbols not containing (> (<numb of formulae>,<numb of real numbers>,<numb of complex numbers>, <numb of truncated power-series>,<maximal degree of truncated power-series>, <absolute accuracy>,<relative accuracy>, <numb of identifiers>,<numb of special derivatives>)

<numb of formulae>::= <integer>

<numb of real numbers>::= <integer>

<numb of complex numbers>::= <integer>

<numb of truncated power-series>::= <integer>

<maximal degree of truncated power-series>::= <integer>

<absolute accuracy>::= <number>

<relative accuracy>::= <number>

<numb of identifiers>::= <integer>

<numb of special derivatives>::= <integer>

<compound formula-statement>::= <formula statement>|<compound formula-statement>;<formula statement>

<formula statement>::= <formula block>|NLCR|EXPAND|NOT EXP|<formula assignment-statement>|<print-string statement>|<output statement>|<special-derivative statement>|<declaration to be of real type statement>|<coefficients of power series statement>|<solve linear equations statement>

<formula block>::= FIX; <compound formula-statement>;<formula-block end>

<formula-block end>::= ERASE|<erase but retain statement>

<erase but retain statement>::= ER B RET(<formula-identifier list>)

<formula-identifier list>::= <formula identifier>|<formula-identifier list>,<formula identifier>

<formula assignment-statement>::= <formula identifier>:= <formula>

<print-string statement>::= PR STRING(<any sequence of symbols,except an odd number of consecutive symbols)>)

<output statement>::= OUTPUT R(<any sequence of symbols not containing : or ?>:= <formula>)|OUTPUT R(?:= <formula>)|OUTPUT C(<any sequence of symbols not containing : or ?>:= <formula>)|OUTPUT C(?:= <formula>)

<special-derivative statement>::= SPEC DER(<algebraic variable>,<list of derivative pairs>)|SPEC DER(<algebraic variable>)

<list of derivative pairs>::= <formula>,<formula>|<list of derivative pairs>,<formula>,<formula>

<declaration to be of real type statement>::= REAL(<algebraic-variable list>)

<algebraic-variable list>::= <algebraic variable>|<algebraic-variable list>,<algebraic variable>

<coefficients of power series statement>::= COEFF(<formula>,<coefficient-identifier list>)

<coefficient-identifier list>::= <formula-identifier list>

<solve linear equations statement>::= SOL LIN EQ(<order>,<algebraic-variable list>,<formula list>)

<order>::= <integer>

3.2.4. The semantics of a formula program

3.2.4.1. The heading

In the heading of the formula program,one may first give some informal information such as a description of the purpose of the formula program and an identification number.
Between the brackets (and) a set of numbers have to be given,the first seven of which are read by the general formula-manipulation system of chapter 2 and the last two of which are read by the formula-program processor.

The first five numbers serve to allocate storage space for the formulae, real numbers, complex numbers and truncated power-series.
The cost in X8 machine words is:

  numb of formulae + numb of formulae ÷ 6
+ 2 × numb of real numbers
+ 4 × numb of complex numbers
+ (1 + numb of truncated power-series) × (2 + maximal degree of truncated power-series).

The first four numbers should be non-negative and should not exceed 2^{13}.
The "absolute accuracy" is used in transforming a complex number into a real number and for rounding a real number to zero.
The "relative accuracy" is used in transforming a real number into an integral number.
An upper bound for the number of identifiers to be used should be given as "numb of identifiers"; the cost in X8 machine words is: 3 × numb of identifiers.
An upper bound for the number of special derivatives to be used should be given as "numb of special derivatives"; the cost in X8 machine words is: 2 × numb of special derivatives.

3.2.4.2. A compound formula-statement

Example: "f:= a + b; NLCR; g:= a - b"

3.2.4.3. A formula block

If the "formula-block end" is "ERASE", then all the formulae which are built up during the execution of the formula block are erased (including the algebraic variables).
Not only the formulae are erased but also the formula identifiers to which the formulae were assigned. (Note that the identifiers themselves remain stored, but they lose their meaning.)

Example 1: "FIX; f:= a + b; ERASE"
The internally stored formulae "a", "b", and "a + b" are erased and the identifiers "a", "b", and "f" have lost their meaning.

Example 2: "FIX; a:= 1; f:= a + b; ERASE"
The internally stored formulae "b" and "1 + b" are erased and the identifiers "b" and "f" have lost their meaning. That the formula "1" is not erased and, therefore, the formula identifier "a" is also not erased, follows from the fact that the processor protects the formulae: "0", "1", "-1" and "i" against erasure; they are treated as basic formulae.
The same situation holds for:
Example 3: "a:= 2; FIX; f:= a + b; ERASE"
Again, the internally stored formulae "b" and "2 + b" are, together with the identifiers "f" and "b", erased, but the formula "2" and the identifier "a" are not erased.

We now treat the case that the "formula-block end" is an "erase but retain statement", e.g. "ER B RET(f,g)".

The remarks, as given for ERASE, may now be repeated, except that the formulae given by the formula identifiers "f" and "g" are not erased nor are the algebraic variables occurring in "f" and "g" erased.

Example 4: "FIX; f:= a + b; h:= c + d; g:= a + c; ER B RET(f,g)"
The formula "c + d" is erased, whereas the formulae "a + b" and "a + c" are not.
Again, identifiers corresponding to erased formulae or erased algebraic variables, as "h" and "d", lose their meaning.

Remark: An occurrence of

"<formula-block end>; END;"

in the formula program may be replaced by "END;".

3.2.4.4. NLCR

This statement has as effect a"new line carriage return" command sent to the printer and the punching of an nlcr symbol in the output tape.

3.2.4.5. EXPAND, NOT EXP

The processor is in the expanding state after a call of "EXPAND",

or in the not-expanding state after a call of "NOT EXP".
In the not-expanding state, the processor performs some elementary transformations, i.e.:

1. A sum, difference, product, quotient or power of two numbers is transformed into a number; e.g. "1 + 2" is transformed into "3".
2. The integral power or elementary function of a number is transformed into a number; e.g. "sin(0)" is transformed into "0".
3. The transformations concerning the zero- and unit element (e.g. $0 + a \rightarrow a$, $1 \times a \rightarrow a$, $0 \times a \rightarrow 0$ etc.).

If the processor is in the expanding state, then,besides the above transformations, the following transformations are performed:

4. $(a + b) \times c \rightarrow a \times c + b \times c$, $a \times (b + c) \rightarrow a \times b + a \times c$ and the quotient transformations as e.g. $a + b/x \rightarrow (a \times x + b)/x$ (see section 2.8);
5. the transformations as given in section 3.2.2.4.

In order to perform simplifications, either automatically (quotients and truncated power-series),or by means of the "OUTPUT R" or the "SIMPL" operator, it is necessary that the processor be in the expanding state.

3.2.4.6. <u>A formula assignment-statement</u>

A "formula assignment-statement" can be used to assign a formula to a formula identifier.
After the assignment of a formula, say "a + b", to the formula identifier "f", each occurrence of "f" in another formula, later on,leads to a substitution of "a + b" for "f".
<u>Example</u>: "f:= a + b; g:= f + c", has as effect that "g" is the formula "(a + b) + c"; if "g" were output, it is output as "a + b + c" (provided "a", "b" and "c" are not themselves formula identifiers but, instead, algebraic variables).

3.2.4.7. <u>A print-string statement</u>

By means of a print-string statement, it is possible to output a sequence of symbols. If one wants to output the symbol ")", one should place two consecutive symbols ")" in the string.

Examples: "PR STRING(a + b + c)" leads to the output "a + b + c"
"PR STRING(a + sin(b)) + cos(c)))" leads to the output
"a + sin(b) + cos(c)".

3.2.4.8. The output statement involving OUTPUT R

Examples: Let "a" and "b" be algebraic variables. Then
"OUTPUT R(f:= a + b)" leads to the output: "f:= a + b;".
"OUTPUT R(?:= a + b)" leads to the output: "a + b".
"EXPAND; OUTPUT R(?:= a + b - a)" leads to the output: "b".
"NOT EXP; OUTPUT R(?:= a + b - a)" leads to the output "a + b - a".
In general, one wants the output of a formula and, moreover, some text giving information about the formula. By means of the "OUTPUT R" command both text and the formula to be output can be specified in such a way that the result which is output may be used to form an ALGOL 60 program or a formula program.
If one does not want the surrounding text, then the output statement may be used in the form "OUTPUT R(?:= ...)".
If the "OUTPUT R" command is given when the processor is in the expanding state, then firstly the formula to be output is simplified and then optimal output is produced; i.e., the output does not contain superfluous symbols.
If the "OUTPUT R" command is given when the processor is not in the expanding state, then the output is produced immediately from the given formula, without a simplification beforehand.

3.2.4.9. The output statement involving OUTPUT C

Examples: Let "a", "b", "x" and "y" be algebraic variables; let "a" and "b" be declared to be of real type; then
"OUTPUT C(f:= x + y)" leads to the output: "ASSIGN(f,S(x,y));".
"OUTPUT C(f:= a + x + y)" leads to the output: "ASSIGN(f,SR(a,S(x,y)));".
"OUTPUT C(?:= a + x + b + y)" leads to the output: "SR(a + b, S(x,y))".
"OUTPUT C(?:= a + b × x × a + b)" leads to the output:
"SR(a + b, PR(b × a,x))".
"OUTPUT C(?:= 2 + i + x × 2 × i)" leads to the output:
"S(CN(2,1), P(x,CN(0,2)))".
"OUTPUT C(?:= 2 + 2 × a)" leads to the output: "RN(2 + 2 × a)".

If the "OUTPUT C" command is given in the form "OUTPUT C(?:= ...)", then the formula occurring at the place of the dots is output in a special complex form, adapted for using the output to construct a complex-arithmetic ALGOL 60 program in the sense of chapter 4.
A formula may contain real parts, which are as much as possible combined with each other, in order to obtain an efficient complex-arithmetic ALGOL 60 program.
This recombination is illustrated by the third and the fourth example, where "a" and "b" have been declared to be of real type.
Note that the procedures *SR* and *PR* of chapter 4, calculate a sum or product, respectively, of a real and a complex number.

If the "OUTPUT C" command is given in the form "OUTPUT C(text:= ...)", then the output in the complex form of the formula is preceded by:
1. the symbols "ASSIGN(",
2. the symbols of the text, followed by the symbol ","
and it is closed by the symbols ");".
The output obtained is then a complex assignment-statement in the sense of chapter 4.

The reason for introducing the possibility of output in the complex form is that writing a complex-arithmetic ALGOL 60 program may be cumbersome, since an expression involving complex numbers can not be written in the usual form; instead, each +, −, ×, / and ↑ symbol has to be treated as an operator involving the procedures *S*, *D*, *P*, *Q* and *POWER*, declared in chapter 4.
One may now construct a formula program, in which the complex-arithmetic expression is programmed in the usual form; the output of this program may then be used to construct the complex-arithmetic ALGOL 60 program.

Remark: The processor does not provide for "new line carriage return" symbols in the output of a formula.

3.2.4.10. A special-derivative statement

The derivatives with respect to a variable (given by the algebraic variable) of certain formulae (given by their formula identifiers), may be specified by means of a "special-derivative statement".

Example: "SPEC DER(x, f, fprime, g, cos(x))", has as effect that if, later on, the derivative of "f" is calculated, then "fprime" is substituted for this derivative; the same holds for "g" and "cos(x)". Thus, "DER(f ↑ 2 + g ↑ 2, x)" leads to the formula:
"2 × f × fprime + 2 × g × cos(x)".

The result of "SPEC DER(x, f, fprime, fprime, fdprime)" is that a call of "DER(DER(f,x), x)", later on, has as result "fdprime".

The effect of "SPEC DER(x, f, fprime, g, DER(f,x) + f)" is that a call "DER(g,x)", later on, has as result "fprime + f".

A next call of "SPEC DER" cancels the effect of all preceding calls of "SPEC DER".

3.2.4.11. A declaration to be of real type statement

This statement enables one to specify certain algebraic variables to be of real type, such that an output statement, in the "OUTPUT C" form, has as effect that these algebraic variables are treated as real variables, in contrast to algebraic variables which are not declared to be of real type.
Example: "REAL(a,b); OUTPUT C(?:= a + b)" has as effect that the symbols "RN(a + b)" are output; whereas "OUTPUT(?:= a + b)" alone, would output the symbols "S(a,b)".

Through the execution of a "formula-block end" the effect of a "declaration to be of real type statement" can be cancelled.

3.2.4.12. A coefficients of power series statement

This statement enables one to give names to the coefficients of a truncated power-series.
Example: "f:= TPS(x, c0, c1, c2); COEFF(f ↑ 2, coeff0, coeff1, coeff2)" leads to the assignment of the formulae

"c0 ↑ 2", "2 × c0 × c1" and "c1 ↑ 2 + 2 × c0 × c2"

to the formula identifiers "coeff0", "coeff1" and "coeff2", respectively. If "f" does not happen to be a truncated power-series (e.g. "c1" and "c2" are equal to zero), then "coeff1" and "coeff2" get the values zero.

3.2.4.13. A solve linear equations statement

A "solve linear equations statement" may be used to solve n equations, obtained by equating n linear formulae in n unknowns to zero.
The value of n is given by the absolute value of the parameter "order".
If the equations are singular, then the processor signals this fact by outputting the error message "singular system".

If the equations are nonsingular, then the solutions are calculated by means of successive elimination. These solutions are then output; moreover, if the value of "order" is positive, then the algebraic variables given in the "algebraic variable list" become formula identifiers to which the solutions are assigned. If the value of "order" is negative, then the algebraic variables remain algebraic variables, but, in general, less memory space is used than in the other case.

If the equations are linear and nonsingular, then a solution is always found (provided that there is enough memory space). In some special cases, the solution will also be found if the equations are nonlinear. It is then necessary that the ordering of the algebraic variables and the ordering of the equations be correlated with each other in a special way. (Note that for linear equations the orderings may be given arbitrarily.) Before defining this correlation, we shall describe the elimination process:
Let $x_1, \dots, x_n$ be the algebraic variables; let $f_1, \dots, f_n$ be the left-hand sides of the equations. In general, f_i is a function of $x_1, \dots, x_n$.
Take an x_i for which $\frac{\partial f_1}{\partial x_i}$ is not identically zero; if such an x_i does not exist, then the equations are called singular.
We assume that f_1 is a linear function of x_i; i.e.

$$f_1 = A + B \times x_i$$

in which $A = f_1(x_1, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)$

and $B = \frac{\partial f_1}{\partial x_i}$, independent of x_i.

From $f_1 = 0$, we calculate $x_i = -A/B$, which is a function of $x_1, \ldots, x_{p-1}, x_{p+1}, \ldots, x_n$. Substituting this x_i into $f_2, f_3, \ldots, f_n$ gives n-1 equations in n-1 variables. After these equations have been solved, x_i can be calculated.

It is now easily seen that the treatment of a set of nonlinear equations might be successful if the variables and the equations can be arranged in such a way that the condition: "there exists an x_i such that f_1 is linear in x_1" is satisfied during each step of the elimination process. Many nonlinear equations, resulting from truncated-power-series manipulations, have the form:
for $i = 1, \ldots, n$:
f_i is linear in x_i, does not depend on $x_{i+1}, \ldots, x_n$ but is nonlinear in $x_1, \ldots, x_{i-1}$.
Since f_1 depends only on x_1 in a linear manner, x_1 can be calculated and substituted into $f_2, \ldots, f_n$. Since, then, f_2 depends only on x_2 in a linear manner, x_2 can be calculated and substituted into $f_3, \ldots, f_n$; etc.
Example: let $y = 1 + y_1 x + y_2 x^2 + y_3 x^3 + O(x^4)$;
let $z = 1 + z_1 x + z_2 x^2 + z_3 x^3 + O(x^4)$; if $y = \sqrt{z}$, or $y^2 - z = 0$, then y_n may be expressed in terms of z_n.
From

$$y^2 - z = (2y_1 - z_1)x + (2y_2 + y_1^2 - z_2)x^2 + (2y_3 + 2y_1 y_2 - z_3)x^3 + O(x^4),$$

one obtains the following equations in the variables y_1, y_2 and y_3:

$$2y_1 - z_1 = 0$$

$$2y_2 + y_1^2 - z_2 = 0$$

$$2y_3 + 2y_1 y_2 - z_3 = 0$$

which have, indeed, the above form; and can, therefore, be treated by means of a "solve linear equations statement". The resulting formula program reads:

```
"<heading> y:= TPS(x, 1, y1, y2, y3);
z:= TPS(x, 1, z1, z2, z3);
COEFF(y ↑ 2 - z, c0, c1, c2, c3);
SOL LIN EQ(3, y1, y2, y3, c1, c2, c3);
END;"
```

The output of this formula program is:

```
"y1:= 1/2 × z1;
 y2:= -1/8 × z1 ↑ 2 + 1/2 × z2;
 y3:= 1/16 × z1 ↑ 3 - 1/4 × z1 × z2 + 1/2 × z3;"
```

As illustrated in this example, the output defined by a "solve linear equations statement" consists of assignment statements; they may be used for an ALGOL 60 program or for another formula program.

Choosing the value of "order" to be negative may be very effective with regard to the memory space, since, in the final stage of the elimination process, the processor erases all already output results. For example, if f_n and x_n form the last pair, then x_n is calculated and f_n is erased (f_n has, in general, become a lengthy formula since $x_1, \ldots, x_{n-1}$ have been substituted into it); next, for i = 1, ..., n-1 the x_i are calculated by substituting x_n into it, and after they have been output, they are immediately erased.

If the results of a "solve linear equations statement" have to be used in other computations, it might be desirable to make two formula programs: the first one for solving the equations with "order" negative and the second one for the remainder of the calculations. Note that it is possible, by means of "print string statements", to make a formula program which generates another formula program.

3.2.4.14. Formula identifier vs. algebraic variable

In this section we explain the difference between a formula identifier and an algebraic variable.

Consider an identifier and its first occurrence in a formula program or its first occurrence after the identifier is erased through a "formula-block end". The identifier is a formula identifier if this occurrence is:

at the left-hand side of a formula assignment-statement or as coefficient in a "coefficients of power series statement", or as the remainder identifier in an integral quotient, or as an algebraic variable in the "algebraic-variable list" of a "solve linear equations statement" in which the parameter "order" has a positive value.
In the last case, the identifier becomes a formula identifier after the execution of the "solve linear equations statement"; during the execution it is an algebraic variable.

If the first occurrence of the identifier is at another place in the formula program, then the identifier is an algebraic variable.

It is possible that a formula identifier changes into an algebraic variable or vice versa (as in a "solve linear equations statement").

Example:

"(10, 0, 0, 0, 0, $_{10}$-10, $_{10}$-10, 3, 0)
FIX; f:= a + b; OUTPUT R(the formula f:= f); ERASE;
a:= 1 + f; b:= 1 - f;
OUTPUT R(the formula a:= a);
OUTPUT R(the formula b:= b);
END;"

This formula program gives the output:

"the formula f:= a + b;
the formula a:= f + 1;
the formula b:= - f + 1;"

3.2.5. Execution of a formula program

The formula statements in a formula program are executed in the order in which they are read by the processor.
Each formula statement which has been read is executed immediately before the reading of the next formula statement. Some errors in the formula program are detected and signalled, after which the execution is discontinued.
The output resulting after a discontinuation has the following structure:

<epilogue>
line number = <integer> execution time = <integer>

The "line number" gives the number of lines of the formula program from the first line after the heading until the line on which the execution has been discontinued.
The "execution time" gives the time, in seconds, during which the formula program was under execution.
The possible detected errors are given now, with the text defining the error:

<table>
<tr><th>text:</th><th>error:</th></tr>
<tr><td>| not followed by ^</td><td>The symbol ↑ is punched wrongly.</td></tr>
<tr><td>_ not followed by i</td><td>The symbol $\underline{i}$ is punched wrongly.</td></tr>
<tr><td>id too long</td><td>Identifier too long (more than 8 symbols).</td></tr>
<tr><td>wrong symbol in id</td><td>A symbol of an identifier is neither a letter nor a digit.</td></tr>
<tr><td>too much id</td><td>There are more identifiers than given by "numb of identifiers" in the heading.</td></tr>
<tr><td>) missing</td><td>There is no ")" for a corresponding "(".</td></tr>
<tr><td>degree too large</td><td>A truncated power series has a degree greater than "maximal degree of truncated power-series" as given in the heading.</td></tr>
<tr><td>heading not closed with $\underline{)}$</td><td>Evident.</td></tr>
<tr><td>statement not closed with ;</td><td>Evident.</td></tr>
<tr><td>wrong assignment statement</td><td>The symbol ":=" does not occur in a formula assignment statement.</td></tr>
<tr><td>degree of tr power series too small</td><td>In a "coefficients of power series statement", there are more formula identifiers than there are coefficients.</td></tr>
<tr><td>FIX missing</td><td>A formula block does not begin with "FIX;".</td></tr>
<tr><td>singular system</td><td>The system of equations of a "solve linear equations statement" is singular.</td></tr>
<tr><td>arg in subst not alg var</td><td>Evident.</td></tr>
<tr><td>degree of TPS not appropriate</td><td>Evident.</td></tr>
<tr><td>. not followed by unsigned integer</td><td>Evident.</td></tr>
<tr><td>$_{10}$ not followed by integer</td><td>Evident.</td></tr>
</table>

The execution of a formula program may also be terminated due to a lack of storage space. If this storage space concerns the space for formulae, real and complex numbers, and truncated power-series, then the "epilogue" has the form:

array too small <n1><n2><n3><n4><n5>

in which "n1", "n2", "n3" and "n4" are numbers defining the current values of the pointers of the arrays in which the formulae, real numbers, complex numbers and truncated power-series, respectively, are stored; and in which "n5" is a number defining the number of times "FIX" is called in the formula program as well as in the processor; this number should not exceed 25.
(One might try to run the formula program again with other numbers in the heading.)

Finally, the execution of a formula program may be terminated due to a lack of free storage space, not under control of the processor; e.g. the storage space for recursive procedure calls.

It is remarked that the Electrologica X8 computer of the Mathematical Centre has, in its present state, a total storage space, which is left for the storage of formulae and as working space, consisting of about 18000 machine words.

The "epilogue" of a formula program, which has been executed until the "END", is: "ready".

3.2.6. Discussion

The considerations which led to the particular formula-program processor, as described in this chapter, were:

1. The user should be able to express himself conveniently.
2. The formula-program processor should occupy little memory space in order to make it applicable for small computers.

These considerations contradict each other; if the user wants more programming tools, he can extend the formula-program processor at will, using the general system of chapter 2.

3.2.7. Examples

In this section we give some examples, in which the possibilities of making formula programs are demonstrated.

Example 1 demonstrates:

a) simplification of formulae with trigonometric functions;
b) the fact that algebraic variables do not necessarily lose their meaning by an ERASE statement;
c) complex conjugation;
d) the fact that real formulae do not have to look like real formulae;
e) the fact that infinity may be output as $.1_{10}11$;
f) the calculation of a common divisor, which does not necessarily have to be output as a nice formula;
g) an "erase but retain statement";
h) some substitutions involving truncated power-series.

```
Formula program RPR 130667/03
(8192,  250,  250,  50,  3,  10-10,  10-10,  11,  0)
FIX; EXPAND;
  NLCR; PR STRING(Results formula program RPR 130667/03,);
  NLCR; PR STRING(in which we demonstrate several possibilities );
  PR STRING(of formula manipulations.);

  NLCR;
   PR STRING(The following four examples are taken from the Handbook );
  NLCR; PR STRING(for mathematical functions and tables [1]);
  NLCR; PR STRING(formula 4.3.27: sin(3x)) - 3sin(x)) + 4sin(x))↑3 = );
  OUTPUT R(?:= sin(3 × x) - 3 × sin(x) + 4 × sin(x) ↑ 3);
ERASE; FIX;
  NLCR; PR STRING(formula 4.3.28: cos(3x)) + 3cos(x)) - 4cos(x))↑3 = );
  OUTPUT R(?:= cos(3 × x) + 3 × cos(x) - 4 × cos(x) ↑ 3);
ERASE; FIX;
```

```
  NLCR; PR STRING
  (formula 4.3.29: sin(4x)) - 8 cos(x))↑3 sin(x)) + 4 cos(x)) sin(x)) = );
  OUTPUT R(?:= sin(4 × x) - 8 × cos(x) ↑ 3 × sin(x) +
  4 × cos(x) × sin(x));
ERASE; FIX;
  NLCR; PR STRING(formula 4.3.30: cos(4x)) - 8 cos(x))↑4 + 8 cos(x))↑2 = );
  OUTPUT R(?:= cos(4 × x) - 8 × cos(x) ↑ 4 + 8 × cos(x) ↑ 2 );
ERASE; FIX;
  pi:= 3.14159265359; a:= SIMPL(cos(pi/2 - x) - sin(x));
  OUTPUT R(cos(pi/2 - x) - sin(x) := a);
ERASE; FIX; OUTPUT R(
a:= a);
  PR STRING( Note that a has not lost its meaning since it is equal
to zero, a formula not erased by ERASE.); NLCR;
  PR STRING(We therefore set a equal to b.); a:= b; NLCR;
ERASE; FIX;
  PR STRING(Now a is erased and is output as: ); OUTPUT R(?:= a);
  NLCR;

  f:= i /(x ↑ 2 + (1 + i) × x + i) + (2/(1 + i))/(x↑2 - 1);
  OUTPUT R(f:= f);
  OUTPUT R(complex conjugate of f:= CC(f));
  g:= (f + CC(f))/2; h:= (f - CC(f))/(2 × i);
  OUTPUT R(real part of f:= g);
  OUTPUT R(imag part of f:= h);
  NLCR; PR STRING(The two last results do not seem to be real,);
  NLCR; PR STRING(but for x = 2, 3 and 4 the real part equals:);
  NLCR; OUTPUT R(?:= SUBST(g,x,2)); PR STRING(          );
  OUTPUT R(?:= SUBST(g,x,3)); PR STRING(          );
  OUTPUT R(?:= SUBST(g,x,4)); NLCR;
  PR STRING(and for x = 2, 3 and 4 the imag part equals:);
  NLCR; OUTPUT R(?:= SUBST(h,x,2)); PR STRING(          );
  OUTPUT R(?:= SUBST(h,x,3)); PR STRING(          );
```

```
  OUTPUT R(?:= SUBST(h,x,4));
  NLCR; PR STRING(For x = 1 the real part equals );
  OUTPUT R(?:= SUBST(g,x,1));
  NLCR; PR STRING(and the imaginary part equals );
  OUTPUT R(?:= SUBST(h,x,1));

  g:= 1/f; NLCR; OUTPUT R(The inverse g of f:= g);
  h:= COMM DIV(g,x ↑ 2 + 1);
  OUTPUT R(The gcd of g and x ↑ 2 + 1 := h);
  NLCR; PR STRING(If the noninteresting number -1 + i is divided out );
  PR STRING(one obtains as gcd: ); NLCR;
  OUTPUT R(?:= SIMPL(h/(-1 + i)));
  k:= QUOT(g,h,r);
  OUTPUT R(The integral quotient of g and h:= k);
  OUTPUT R(whilst the remainder:= r);
ER B RET(g,h);
  OUTPUT R(After the 'erase but retain statement' g and h
  remain unchanged
  g:= g); OUTPUT R(h:= h);
FIX; f:= TPS(x,a,b,c,d); OUTPUT R(f:= f);
  g:= SUBST(f,x,x);
  OUTPUT R(g is f as truncated power-series
g:= g); g:= SUBST(f,a,d, b,c, c,b, d,a);
  OUTPUT R(g becomes the truncated power-series f with other
coefficients:= g); g:= SUBST(f,x,TPS(a,0,1,2,3));
  OUTPUT R(g becomes a truncated power-series in a
g:= g);
ERASE;
END;
```

Results formula program RPR 130667/03,
in which we demonstrate several possibilities of formula manipulations.
The following four examples are taken from the Handbook
for mathematical functions and tables [1]
formula 4.3.27: sin(3x) – 3sin(x) + 4sin(x)∧3 = 0
formula 4.3.28: cos(3x) + 3cos(x) – 4cos(x)∧3 = 0
formula 4.3.29: sin(4x) – 8 cos(x)∧3 sin(x) + 4 cos(x) sin(x) = 0
formula 4.3.30: cos(4x) – 8 cos(x)∧4 + 8 cos(x)∧2 = 1
cos(pi/2 – x) – sin(x) := 0;

a:= 0; Note that a has not lost its meaning since it is equal
to zero, a formula not erased by ERASE.
We therefore set a equal to b.
Now a is erased and is output as: a

f:= 1/(x∧2+(i–1)×x–i);
complex conjugate of f:= 1/(x∧2–(1+i)×x+i);
real part of f:= –i/2×x/(–i/2×x∧3+i/2×x∧2–i/2×x+i/2);
imag part of f:= i/2/(–i/2×x∧3+i/2×x∧2–i/2×x+i/2);
The two last results do not seem to be real,
but for x = 2, 3 and 4 the real part equals:
2/5 3/20 4/51
and for x = 2, 3 and 4 the imag part equals:
–1/5 –1/20 –1/51
For x = 1 the real part equals .1$_{10}$11
and the imaginary part equals .1$_{10}$11

The inverse g of f:= x∧2+(i–1)×x–i;
The gcd of g and x ∧ 2 + 1 := (i–1)×x–1–i;
If the noninteresting number –1 + i is divided out one obtains as gcd:
x+i

```
The integral quotient of g and h:= (-1/2-i/2)×x+1/2+i/2;
whilst the remainder:= 0;
After the 'erase but retain statement' g and h
  remain unchanged
  g:= x↑2+(i-1)×x-i;
h:= (i-1)×x-1-i;
f:= a+b×x+c×x↑2+d×x↑3+O(x↑4);
g is f as truncated power-series
g:= a+b×x+c×x↑2+d×x↑3+O(x↑4);
g becomes the truncated power-series f with other
coefficients:= d+c×x+b×x↑2+a×x↑3+O(x↑4);
g becomes a truncated power-series in a
g:= a+b×a+(2×b+c)×a↑2+(3×b+4×c+d)×a↑3+O(a↑4);
ready
line number = 76 execution time = 79 sec
```

Example 2 demonstrates:

a) the two different output statements and the effect of a "declaration to be of real type statement";

b) some "solve linear equations statements".

```
Formula program RPR 100367/01
(1000,  500,   100,   0,    0,    ⏨-10,  ⏨-10,  10,    0)
NLCR; PR STRING(Results formula program RPR 100367/01);
EXPAND; FIX; OUTPUT C(a:= b + .1⏨-1 × c); ERASE;
FIX; OUTPUT R(a:= b + .1⏨-1 × c); ERASE;
FIX; REAL(a,b);
  OUTPUT C(z:= a + y + b); OUTPUT C(z:= a × y × b);
  NLCR; OUTPUT C(?:= a + a × y × b + b × 3.14 + i × 3.14);
ERASE;
FIX; SOL LIN EQ(1,x,a × x - b); ERASE;
FIX; SOL LIN EQ(2,x,y,a × x + b × y - e,c × x + d × y - f); ERASE;
FIX; SOL LIN EQ(2,x,y,a × x - e, d × y - f); ERASE;
FIX; SOL LIN EQ(-2,x,y,b × y - e, c × x - f); ERASE;
FIX; SOL LIN EQ(-3,x,y,z, a × y + b × z - c,
                          a × x + b × y - d,
                          a × z + b × x - f);
ERASE;
FIX; SOL LIN EQ(3,x,y,z, a × y + b × z - c,
                         a × x + b × y - d,
                         a × z + b × x - f);
ER B RET(x); NLCR; OUTPUT R(x:= x);
FIX; SOL LIN EQ(2,x,y,a × x - e, c × x - f);
END;
```

Results formula program RPR 100367/01

```
ASSIGN(a,S(b,PR(1/100,c)));
a:= b+1/100×c;
ASSIGN(z,SR(a+b,y));
ASSIGN(z,PR(a×b,y));
SR(a+b×157/50,S(PR(a×b,y),CN(0,157/50)))
x:= b/a;
x:= (-b×f+e×d)/(a×d-b×c);
y:= (a×f-e×c)/(a×d-b×c);
x:= e/a;
y:= f/d;
y:= e/b;
x:= f/c;
y:= (a↑2×c-a×b×f+b↑2×d)/(a↑3+b↑3);
x:= (a↑2×d-a×b×c+b↑2×f)/(a↑3+b↑3);
z:= (a↑2×f-a×b×d+b↑2×c)/(a↑3+b↑3);
y:= (a↑2×c-a×b×f+b↑2×d)/(a↑3+b↑3);
x:= (a↑2×d-a×b×c+b↑2×f)/(a↑3+b↑3);
z:= (a↑2×f-a×b×d+b↑2×c)/(a↑3+b↑3);

x:= (a↑2×d-a×b×c+b↑2×f)/(a↑3+b↑3);
singular system
line number = 20 execution time = 156 sec
```

Example 3 demonstrates a complicated substitution operation.

```
Formula program RPR 150667/01
(2048,  500,   0,     0,     0,    ₁₀-10,  ₁₀-10,   7,      0)
NLCR; PR STRING(Results formula program RPR 150667/01);
NLCR; PR STRING(this example is due to L. Meertens);
f:= a × b × c × d;
OUTPUT R(f0:= f);
g:= 4 / (4 × f - SUBST(f,a,a + 3, b,b - 1, c,c - 1, d,d - 1)
               - SUBST(f,a,a - 1, b,b + 3, c,c - 1, d,d - 1)
               - SUBST(f,a,a - 1, b,b - 1, c,c + 3, d,d - 1)
               - SUBST(f,a,a - 1, b,b - 1, c,c - 1, d,d + 3));
OUTPUT R(g1:= g);
END;
```

```
Results formula program RPR 150667/01
this example is due to L. Meertens
f0:= a×b×c×d;
g1:= 4/(4×a×b+4×a×c+4×a×d+4×b×c+4×b×d+4×c×d-8×a-8×b-8×c-8×d+12);
ready
line number = 10 execution time = 19 sec
```

Example 4 demonstrates:

a) the use of a "special-derivative statement";

b) the use of complex-output statements for preparing an ALGOL 60 complex-arithmetic program.

```
Formula program for the damping of waves by surface-active materials
RPR 230367/01
(1000,  50,  100,  0,  0,  ₁₀-10,  ₁₀-10,  25,  4)
NLCR;
PR STRING(comment results from formula program RPR 230367/01;);
FIX; NOT EXP; REAL(omega,nu);
  OUTPUT C(m:= sqrt(k↑2 + i × omega/nu));
  OUTPUT C(mprime:= k/sqrt(k↑2 + i × omega/nu));
ERASE; FIX;
  SPEC DER(k,m,mprime); REAL(rho,omega,sigma,nu,pi,g,eta);
  F1:= i × rho × g × k/omega + i × rho × omega + 2 × eta × k↑2 +
    i × k↑3 × sigma/omega;
  F2:= - rho × g × k/omega + (- 2 × i) × eta × k × m -
    k↑3 × sigma/omega;
  F3:= (- 2 × i) × k↑2 × eta + k↑3 × eps/omega;
  F4:= (2 × k↑2 + i × omega/nu) × eta - i × k↑2 × m × eps/omega;
  OUTPUT C(F1:= F1); OUTPUT C(F2:= F2);
  OUTPUT C(F3:= F3); OUTPUT C(F4:= F4);
  FIX; OUTPUT C(F1p:= DER(F1,k)); ERASE;
  FIX; OUTPUT C(F2p:= DER(F2,k)); ERASE;
  FIX; OUTPUT C(F3p:= DER(F3,k)); ERASE;
  FIX; OUTPUT C(F4p:= DER(F4,k)); ERASE;
ERASE; FIX;
  SPEC DER(k,F1,F1p,F2,F2p,F3,F3p,F4,F4p);
  delta:= F1 × F4 - F2 × F3; OUTPUT C(delta:= delta);
  OUTPUT C(der:= DER(delta,k));
END;
```

```
comment results from formula program RPR 230367/01;
ASSIGN(m,SQRT(S(INT POW(k,2),Q(PR(omega,iu),RN(nu)))));
ASSIGN(mprime,Q(k,SQRT(S(INT POW(k,2),Q(PR(omega,iu),RN(nu))))));
ASSIGN(F1,S(Q(PR(rho×g,P(iu,k)),RN(omega)),S(PR(rho×omega,iu),
        S(PR(2×eta,INT POW(k,2)),Q(PR(sigma,P(iu,INT POW(k,3))),      *)
        RN(omega))))));
ASSIGN(F2,D(S(PR((-1),Q(PR(rho×g,k),RN(omega))),PR(eta,P(CN(0,-2),
        P(k,m)))),Q(PR(sigma,INT POW(k,3)),RN(omega))));
ASSIGN(F3,S(PR(eta,P(CN(0,-2),INT POW(k,2))),Q(P(INT POW(k,3),
        eps),RN(omega))));
ASSIGN(F4,D(PR(eta,S(PR(2,INT POW(k,2)),Q(PR(omega,iu),RN(nu)))),
        Q(P(iu,P(INT POW(k,2),P(m,eps))),RN(omega))));
ASSIGN(F1p,S(Q(PR(rho×g,iu),RN(omega)),S(PR(2×eta×2,k),Q(PR(3×sigma,
        P(iu,INT POW(k,2))),RN(omega)))));
ASSIGN(F2p,D(SR((-1)×rho×g/omega,S(PR(eta,P(CN(0,-2),m)),PR(eta,
        P(CN(0,-2),P(k,mprime))))),Q(PR(3×sigma,INT POW(k,2)),
        RN(omega))));
ASSIGN(F3p,S(PR(2×eta,P(CN(0,-2),k)),Q(PR(3,P(INT POW(k,2),eps)),
        RN(omega))));
ASSIGN(F4p,D(PR(2×2×eta,k),Q(P(S(PR(2,P(iu,P(k,m))),P(iu,P(INT POW(
        k,2),mprime))),eps),RN(omega))));
ASSIGN(delta,D(P(F1,F4),P(F2,F3)));
ASSIGN(der,D(S(P(F1p,F4),P(F1,F4p)),S(P(F2p,F3),P(F2,F3p))));
ready
line number = 24 execution time = 10 sec
```

*) The processor does not provide for new-line-carriage-return symbols in the output of formulae; therefore, they are punched by hand, followed by a tabulator symbol.

Example 5 demonstrates a simultaneous use of truncated-power-series manipulation and solving linear equations in order to calculate Taylor-series coefficients.

Let

$$f(x) = (g(x))^{\frac{1}{2}}$$

and

$$g(x) = 1 + g_1x + g_2x^2 + g_3x^3 + g_4x^4 + g_5x^5 + O(x^6)$$

$$f(x) = 1 + f_1x + f_2x^2 + f_3x^3 + f_4x^4 + f_5x^5 + O(x^6).$$

Then the results of the following formula program are the formulae for the coefficients f_i as functions of the coefficients g_i.
One should consult the Handbook [1] 3.6.18 for checking the results.

```
Formula program RPR 100367/02
(500,  100,  100,  20,  5,  ₁₀-10,  ₁₀-10,  25,  0)
NLCR; PR STRING(Results formula program RPR 100367/02);
g:= TPS(x,1,g1,g2,g3,g4,g5);
f:= TPS(x,1,f1,f2,f3,f4,f5);
COEFF(f↑2 - g,c0,c1,c2,c3,c4,c5);
SOL LIN EQ(5,f1,f2,f3,f4,f5, c1,c2,c3,c4,c5);
END;
```

```
Results formula program RPR 100367/02
f1:= 1/2×g1;
f2:= -1/8×g1↑2+1/2×g2;
f3:= 1/16×g1↑3-1/4×g1×g2+1/2×g3;
f4:= -5/128×g1↑4+3/16×g1↑2×g2-1/4×g1×g3-1/8×g2↑2+1/2×g4;
f5:= 7/256×g1↑5-5/32×g1↑3×g2+3/16×g1↑2×g3+3/16×g1×g2↑2-
      1/4×g1×g4-1/4×g2×g3+1/2×g5;
ready
line number = 6 execution time = 18 sec
```

Example 6

The problem of section 2.13 is treated again; i.e. determine the Taylor coefficients of

$s_3 = 1 + \ln(s_1)$,

if $s_1 = 1 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + O(x^6)$.

```
Formula program RPR 011167/01
(8192,  500,   0,    100,   5,    ⏨-10,  ⏨-10,  16,    0)
PR STRING(
Results formula program RPR 011167/01);
EXPAND; FIX; s1:= TPS(x,1,a1,a2,a3,a4,a5); OUTPUT R(s[1]:= s1);
ERASE; FIX; d0:= 1 + ln(s1); OUTPUT R(f(s[1]):= d0);
  d1:= DER(d0,s1); d2:= DER(d1,s1); d3:= DER(d2,s1);
  d4:= DER(d3,s1); d5:= DER(d4,s1);
  g:= SUBST(TPS(y,1,d1,d2/2,d3/6,d4/24,d5/120),s1,1);
ER B RET(g); OUTPUT R(g(y):= g);
s3:= SUBST(g,y,TPS(x,0,a1,a2,a3,a4,a5));
OUTPUT R(s[3]:= s3);
END;
```

```
Results formula program RPR 011167/01
s[1]:= 1+a1×x+a2×x↑2+a3×x↑3+a4×x↑4+a5×x↑5+O(x↑6);
f(s[1]):= ln(s1)+1;
g(y):= 1+1×y+(-1/2)×y↑2+1/3×y↑3+(-1/4)×y↑4+1/5×y↑5+O(y↑6);
s[3]:= 1+a1×x+(-1/2×a1↑2+a2)×x↑2+(1/3×a1↑3-a1×a2+a3)×x↑3+
        (-1/4×a1↑4+a1↑2×a2-a1×a3-1/2×a2↑2+a4)×x↑4+
        (1/5×a1↑5-a1↑3×a2+a1↑2×a3+a1×a2↑2-a1×a4-a2×a3+a5)×x↑5+O(x↑6);
ready
line number = 11 execution time = 29 sec
```

Example 7

The following problem, due to Professor N.G. de Bruijn, has been communicated to us by drs. B.J.M. Morselt (both of the Technological University, Eindhoven).

Let $A(u,v) = \sum_{n=0}^{\infty} \sum_{m=0}^{\infty} a_{n,m} u^n v^m$;

let $B(u,v) = \sum_{n=0}^{\infty} \sum_{m=0}^{\infty} b_{n,m} u^n v^m$.

From

$$F = \sum_{k=0}^{\infty} \frac{t^k}{k!} ((A\ (kz,z))^k - (B\ (tz,z))^k) = 0,$$

determine the $b_{n,m}$ as functions of the $a_{n,m}$. That this is possible has been shown by de Bruijn.

The following formula programs calculate $b_{0,1}$, $b_{1,0}$, $b_{0,2}$, $b_{1,1}$, $b_{2,0}$, $b_{0,3}$, $b_{1,2}$, $b_{2,1}$, $b_{0,4}$, $b_{1,3}$, $b_{2,2}$, $b_{0,5}$, $b_{1,4}$ and $b_{2,3}$.

The first program (RPR 231067/01) calculates the first three terms of F as a double truncated power-series in z and t, and produces a second program (RPR 231067/02). This second program calculates the desired coefficients of B.

The reason for the separation of the calculation process is that during the first stage of the calculation process, i.e. the determination of the coefficients of F, quite a number of truncated power-series are needed, whereas they are not needed for the calculation of the coefficients of B.

It is remarked that the calculation of four terms of F led to an exhaustion of the storage space.

```
Formula program RPR 231067/01 The de Bruijn-problem
(8192,  500,   0,    750,   5,    ⏨-10,  ⏨-10,   65,    0)
PR STRING(
Formula program RPR 231067/02 The de Bruijn-problem
(8192,  500,   0,     0,    0,    ⏨-10,  ⏨-10,   40,    0))
EXPAND; PR STRING(
Results RPR 231067/02 The de Bruijn-problem));); 
EXPAND; FIX;
A:= TPS(u,TPS(v,a00,a01,a02,a03,a04,a05),
         TPS(v,a10,a11,a12,a13,a14),
         TPS(v,a20,a21,a22,a23),
         TPS(v,a30,a31,a32),
         TPS(v,a40,a41),a50);
B:= TPS(u,TPS(v,a00,b01,b02,b03,b04,b05),
         TPS(v,b10,b11,b12,b13,b14),
         TPS(v,b20,b21,b22,b23),
         TPS(v,b30,b31,b32),
         TPS(v,b40,b41),b50);
A:= SUBST(A,u,TPS(z,0,k,0,0,0,0),v,TPS(z,0,1,0,0,0,0));
B:= SUBST(B,u,TPS(z,0,TPS(t,0,1,0,0),0,0,0,0),v,TPS(z,0,1,0,0,0,0));
ER B RET(A,B);
T1:= TPS(z,TPS(t,0,1,0,0),0,0,0,0,0);
T2:= TPS(z,TPS(t,0,0,1,0),0,0,0,0,0);
T3:= TPS(z,TPS(t,0,0,0,1),0,0,0,0,0);
F:= T1 × (SUBST(A,k,1) - B) + T2 × (SUBST(A,k,2)↑2 - B↑2)/2 +
    T3 × (SUBST(A,k,3)↑3 - B↑3)/6;
COEFF(F,c0,c1,c2,c3,c4,c5);
```

```
COEFF(c1,C0,C1,C2);
PR STRING(
b01:= SUBST(); OUTPUT R(?:= C1); PR STRING(
  ,b01,0)); OUTPUT R(b01:= b01));
b10:= SUBST(); OUTPUT R(?:= C2); PR STRING(
  ,b10,0)); OUTPUT R(b10:= b10));)；

COEFF(c2,C0,C1,C2,C3);
PR STRING(
b02:= SUBST(); OUTPUT R(?:= C1); PR STRING(
  ,b02,0)); OUTPUT R(b02:= b02));
b11:= SUBST(); OUTPUT R(?:= C2); PR STRING(
  ,b11,0)); OUTPUT R(b11:= b11));
b20:= SUBST(); OUTPUT R(?:= C3); PR STRING(
  ,b20,0)); OUTPUT R(b20:= b20));)；

COEFF(c3,C0,C1,C2,C3);
PR STRING(
b03:= SUBST(); OUTPUT R(?:= C1); PR STRING(
  ,b03,0)); OUTPUT R(b03:= b03));
b12:= SUBST(); OUTPUT R(?:= C2); PR STRING(
  ,b12,0)); OUTPUT R(b12:= b12));
b21:= SUBST(); OUTPUT R(?:= C3); PR STRING(
  ,b21,0)); OUTPUT R(b21:= b21));)；
```

```
COEFF(c4,C0,C1,C2,C3);
PR STRING(
b04:= SUBST(); OUTPUT R(?:= C1); PR STRING(
   ,b04,0)); OUTPUT R(b04:= b04));
b13:= SUBST(); OUTPUT R(?:= C2); PR STRING(
   ,b13,0)); OUTPUT R(b13:= b13));
b22:= SUBST(); OUTPUT R(?:= C3); PR STRING(
   ,b22,0)); OUTPUT R(b22:= b22));););

COEFF(c5,C0,C1,C2,C3);
PR STRING(
b05:= SUBST(); OUTPUT R(?:= C1); PR STRING(
   ,b05,0)); OUTPUT R(b05:= b05));
b14:= SUBST(); OUTPUT R(?:= C2); PR STRING(
   ,b14,0)); OUTPUT R(b14:= b14));
b23:= SUBST(); OUTPUT R(?:= C3); PR STRING(
   ,b23,0)); OUTPUT R(b23:= b23));
END;);
END;
```

```
Formula program RPR 231067/02 The de Bruijn-problem
(8192,   500,    0,     0,     0,    ⏨-10,   ⏨-10,   40,    0)
EXPAND; PR STRING(
Results RPR 231067/02 The de Bruijn-problem);
b01:= SUBST(a10+a01-b01
  ,b01,0); OUTPUT R(b01:= b01);
b10:= SUBST(2×a00×a10+a00×a01-a00×b01-b10
  ,b10,0); OUTPUT R(b10:= b10);
b02:= SUBST(a20+a11+a02-b02
  ,b02,0); OUTPUT R(b02:= b02);
b11:= SUBST(4×a00×a20+2×a00×a11+a00×a02-a00×b02+2×a10⋀2+
        2×a10×a01+1/2×a01⋀2-1/2×b01⋀2-b11
  ,b11,0); OUTPUT R(b11:= b11);
b20:= SUBST(9/2×a00⋀2×a20+3/2×a00⋀2×a11+1/2×a00⋀2×a02-1/2×
        a00⋀2×b02+9/2×a00×a10⋀2+3×a00×a10×a01+1/2×a00×a01⋀2-
        1/2×a00×b01⋀2-a00×b11-b01×b10-b20
  ,b20,0); OUTPUT R(b20:= b20);
b03:= SUBST(a30+a21+a12+a03-b03
  ,b03,0); OUTPUT R(b03:= b03);
b12:= SUBST(8×a00×a30+4×a00×a21+2×a00×a12+a00×a03-a00×b03+8×a10×
        a20+4×a10×a11+2×a10×a02+4×a01×a20+2×a01×a11+a01×a02-b01×
        b02-b12
  ,b12,0); OUTPUT R(b12:= b12);
b21:= SUBST(27/2×a00⋀2×a30+9/2×a00⋀2×a21+3/2×a00⋀2×a12+1/2×a00⋀2×
        a03-1/2×a00⋀2×b03+27×a00×a10×a20+9×a00×a10×a11+3×a00×a10×
        a02+9×a00×a01×a20+3×a00×a01×a11+a00×a01×a02-a00×b01×b02+
        9/2×a10⋀3+9/2×a10⋀2×a01+3/2×a10×a01⋀2+1/6×a01⋀3-1/6×b01⋀3-
        a00×b12-b01×b11-b10×b02-b21
  ,b21,0); OUTPUT R(b21:= b21);
b04:= SUBST(a40+a31+a22+a13+a04-b04
  ,b04,0); OUTPUT R(b04:= b04);
b13:= SUBST(16×a00×a40+8×a00×a31+4×a00×a22+2×a00×a13+a00×a04-a00×
        b04+16×a10×a30+8×a10×a21+4×a10×a12+2×a10×a03+8×a01×a30+4×
        a01×a21+2×a01×a12+a01×a03+8×a20⋀2+8×a20×a11+4×a20×a02+2×
```

```
        a11↑2+2×a11×a02+1/2×a02↑2-b01×b03-1/2×b02↑2-b13
   ,b13,0); OUTPUT R(b13:= b13);
 b22:= SUBST(81/2×a00↑2×a40+27/2×a00↑2×a31+9/2×a00↑2×a22+3/2×a00↑2×
        a13+1/2×a00↑2×a04-1/2×a00↑2×b04+81×a00×a10×a30+27×a00×a10×
        a21+9×a00×a10×a12+3×a00×a10×a03+27×a00×a01×a30+9×a00×a01×
        a21+3×a00×a01×a12+a00×a01×a03+81/2×a00×a20↑2+27×a00×a20×a11+
        9×a00×a20×a02+9/2×a00×a11↑2+3×a00×a11×a02+1/2×a00×a02↑2-a00×
        b01×b03-1/2×a00×b02↑2+81/2×a10↑2×a20+27/2×a10↑2×a11+9/2×
        a10↑2×a02+27×a10×a01×a20+9×a10×a01×a11+3×a10×a01×a02+9/2×
        a01↑2×a20+3/2×a01↑2×a11+1/2×a01↑2×a02-1/2×b01↑2×b02-a00×b13-
        b01×b12-b10×b03-b02×b11-b22
   ,b22,0); OUTPUT R(b22:= b22);
 b05:= SUBST(a50+a41+a32+a23+a14+a05-b05
   ,b05,0); OUTPUT R(b05:= b05);
 b14:= SUBST(32×a00×a50+16×a00×a41+8×a00×a32+4×a00×a23+2×a00×a14+
        a00×a05-a00×b05+32×a10×a40+16×a10×a31+8×a10×a22+4×a10×a13+
        2×a10×a04+16×a01×a40+8×a01×a31+4×a01×a22+2×a01×a13+a01×
        a04+32×a20×a30+16×a20×a21+8×a20×a12+4×a20×a03+16×a11×a30+
        8×a11×a21+4×a11×a12+2×a11×a03+8×a02×a30+4×a02×a21+2×a02×
        a12+a02×a03-b01×b04-b02×b03-b14
   ,b14,0); OUTPUT R(b14:= b14);
 b23:= SUBST(243/2×a00↑2×a50+81/2×a00↑2×a41+27/2×a00↑2×a32+9/2×a00↑2×
        a23+3/2×a00↑2×a14+1/2×a00↑2×a05-1/2×a00↑2×b05+243×a00×a10×
        a40+81×a00×a10×a31+27×a00×a10×a22+9×a00×a10×a13+3×a00×a10×
        a04+81×a00×a01×a40+27×a00×a01×a31+9×a00×a01×a22+3×a00×a01×
        a13+a00×a01×a04+243×a00×a20×a30+81×a00×a20×a21+27×a00×a20×
        a12+9×a00×a20×a03+81×a00×a11×a30+27×a00×a11×a21+9×a00×a11×
        a12+3×a00×a11×a03+27×a00×a02×a30+9×a00×a02×a21+3×a00×a02×
        a12+a00×a02×a03-a00×b01×b04-a00×b02×b03+243/2×a10↑2×a30+
        81/2×a10↑2×a21+27/2×a10↑2×a12+9/2×a10↑2×a03+81×a10×a01×a30+
        27×a10×a01×a21+9×a10×a01×a12+3×a10×a01×a03+243/2×a10×a20↑2+
        81×a10×a20×a11+27×a10×a20×a02+27/2×a10×a11↑2+9×a10×a11×a02+
        3/2×a10×a02↑2+27/2×a01↑2×a30+9/2×a01↑2×a21+3/2×a01↑2×a12+
```

```
        1/2×a01⋀2×a03+81/2×a01×a20⋀2+27×a01×a20×a11+9×a01×a20×a02+
        9/2×a01×a11⋀2+3×a01×a11×a02+1/2×a01×a02⋀2-1/2×b01⋀2×b03-
        1/2×b01×b02⋀2-a00×b14-b01×b13-b10×b04-b02×b12-b11×b03-b23
  ,b23,0); OUTPUT R(b23:= b23);
END;
ready
line number = 70 execution time = 666 sec
```

```
Results RPR 231067/02 The de Bruijn-problem
b01:= a10+a01;
b10:= a10×a00;
b02:= a20+a11+a02;
b11:= 3/2×a10⋀2+a10×a01+3×a00×a20+a00×a11;
b20:= 3/2×a10⋀2×a00+a00⋀2×a20;
b03:= a30+a21+a12+a03;
b12:= 7×a10×a20+3×a10×a11+a10×a02+3×a01×a20+a01×a11+7×a00×a30+
        3×a00×a21+a00×a12;
b21:= 17/6×a10⋀3+3/2×a10⋀2×a01+15×a10×a00×a20+3×a10×a00×a11+2×
        a01×a00×a20+6×a00⋀2×a30+a00⋀2×a21;
b04:= a40+a31+a22+a13+a04;
b13:= 15×a10×a30+7×a10×a21+3×a10×a12+a10×a03+7×a01×a30+3×a01×
        a21+a01×a12+15×a00×a40+7×a00×a31+3×a00×a22+a00×a13+15/2×
        a20⋀2+7×a20×a11+3×a20×a02+3/2×a11⋀2+a11×a02;
b22:= 63/2×a10⋀2×a20+17/2×a10⋀2×a11+3/2×a10⋀2×a02+15×a10×a01×
        a20+3×a10×a01×a11+57×a10×a00×a30+15×a10×a00×a21+3×a10×
        a00×a12+a01⋀2×a20+12×a01×a00×a30+2×a01×a00×a21+25×a00⋀2×
        a40+6×a00⋀2×a31+a00⋀2×a22+59/2×a00×a20⋀2+15×a00×a20×a11+
        2×a00×a20×a02+3/2×a00×a11⋀2;
```

```
b05:= a50+a41+a32+a23+a14+a05;
b14:= 31×a10×a40+15×a10×a31+7×a10×a22+3×a10×a13+a10×a04+15×a01×
         a40+7×a01×a31+3×a01×a22+a01×a13+31×a00×a50+15×a00×a41+7×
         a00×a32+3×a00×a23+a00×a14+31×a20×a30+15×a20×a21+7×a20×
         a12+3×a20×a03+15×a11×a30+7×a11×a21+3×a11×a12+a11×a03+7×
         a02×a30+3×a02×a21+a02×a12;
b23:= 209/2×a10↑2×a30+63/2×a10↑2×a21+17/2×a10↑2×a12+3/2×a10↑2×
         a03+57×a10×a01×a30+15×a10×a01×a21+3×a10×a01×a12+195×a10×
         a00×a40+57×a10×a00×a31+15×a10×a00×a22+3×a10×a00×a13+213/2×
         a10×a20↑2+63×a10×a20×a11+15×a10×a20×a02+17/2×a10×a11↑2+
         3×a10×a11×a02+6×a01↑2×a30+a01↑2×a21+50×a01×a00×a40+12×
         a01×a00×a31+2×a01×a00×a22+59/2×a01×a20↑2+15×a01×a20×a11+
         2×a01×a20×a02+3/2×a01×a11↑2+90×a00↑2×a50+25×a00↑2×a41+
         6×a00↑2×a32+a00↑2×a23+201×a00×a20×a30+59×a00×a20×a21+15×
         a00×a20×a12+2×a00×a20×a03+57×a00×a11×a30+15×a00×a11×a21+
         3×a00×a11×a12+12×a00×a02×a30+2×a00×a02×a21;
ready
line number = 70 execution time = 347 sec
```

Example 8

The following problem is due to drs. F. Göbel (Mathematical Centre, Amsterdam):

Determine the Taylor coefficients of the functions f(x,y) and g(x,y) satisfying:

$$F(x,y) = \{\phi_1(x,y) - x\}\times f(x,y) + \phi_1(x,y)\times\{g(x,0) - f(0,y) - p\times r\times(1-x)\} = 0,$$

$$G(x,y) = \{\phi_2(x,y) - y\}\times g(x,y) + \phi_2(x,y)\times\{f(0,y) - g(x,0) - q\times r\times(1-y)\} = 0,$$

$$f(0,1) = g(1,0) = 0 \quad \text{and} \quad f(1,1) + g(1,1) = 1,$$

The functions ϕ_1 and ϕ_2 are given functions of x and y, the form of which will be given below.

p and q are given constants and r is a constant to be calculated.

The Taylor coefficients of f and g are to be calculated in the point x = 1, y = 1; therefore, we introduce the following variables: $\delta = 1 - x$ and $\varepsilon = 1 - y$.

f and g may be written as:

$$f(x,y) = \sum_{n=0}^{\infty} (\sum_{m=0}^{\infty} f_{nm11}\varepsilon^m)\delta^n,$$

$$g(x,y) = \sum_{n=0}^{\infty} (\sum_{m=0}^{\infty} g_{nm11}\varepsilon^m)\delta^n$$

and f(0,y) and g(x,0) may be written as:

$$f(0,y) = f_1(y) = \sum_{m=1}^{\infty} f_{0m01}\varepsilon^m,$$

$$g(x,0) = g_1(x) = \sum_{n=1}^{\infty} g_{n010}\delta^n.$$

Introducing $\tau = k_1\varepsilon + k_2\delta$, where k_1 and k_2 are given constants, we are able to specify the form of the functions $\phi_1(x,y)$ and $\phi_2(x,y)$:

$$\phi_1(x,y) = 1 + a\tau + b\tau^2 + \dots,$$

$$\phi_2(x,y) = 1 + A\tau + B\tau^2 + \dots,$$

where a, b, A and B are given constants.

Let the Taylor series for F and G be:

$$F(x,y) = \sum_{n=0}^{\infty} \left(\sum_{m=0}^{\infty} F_{nm}\varepsilon^m \right) \delta^n,$$

$$G(x,y) = \sum_{n=0}^{\infty} \left(\sum_{m=0}^{\infty} G_{nm}\varepsilon^m \right) \delta^n.$$

From: $F_{10} = F_{01} = G_{10} = G_{01} = 0$, we can determine:
g_{0011}, f_{0101}, g_{1010} and r.

From: $F_{20} = F_{11} = F_{02} = G_{20} = G_{11} = G_{02} = 0$, we can determine:
f_{1011}, f_{0111}, f_{0201}, g_{0111}, g_{1011} and g_{2010}.

From: $F_{30} = F_{21} = \dots = G_{12} = G_{03} = 0$, we can determine:
f_{2011}, f_{1111}, f_{0211}, f_{0301}, g_{0211}, g_{1111}, g_{2011} and g_{3010}.

In theory, we can repeat this process as far as we wish; in practice, however, it was not possible to calculate the "third order" coefficients f_{2011}, ..., g_{3010}, due to memory limitations.
The following formula programs calculate, of the "second order" coefficients, only the f_{0201} and g_{2010}.

Let us now discuss the calculation process.
It can easily be seen that the six equations determining f_{1011}, ..., g_{2010} have a form which makes them suitable for a direct computation (instead of using the elaborate "solve linear equations statement").

For, F_{20} is a linear function of f_{1011} and g_{2010},
F_{11} is a linear function of f_{0111} and f_{1011},
F_{02} is a linear function of f_{0201} and f_{0111},
G_{02} is a linear function of g_{0111} and f_{0201},

G_{11} is a linear function of g_{1011} and g_{0111},
and G_{20} is a linear function of g_{2010} and g_{1011}.

If g_{2010} is considered to be a constant in the first three equations, then f_{1011}, f_{0111} and f_{0201} can easily be calculated in terms of g_{2010}. If f_{0201} is considered to be a constant in the last three equations, then g_{0111}, g_{1011} and g_{2010} can easily be calculated in terms of f_{0201}. We thus obtain two equations for f_{0201} and g_{2010}, viz:

$$f_{0201} = cf_1 \times g_{2010} + cf_2$$

and $$g_{2010} = cg_1 \times f_{0201} + cg_2$$

from which $$f_{0201} = (cf_2 + cf_1 \times cg_2)/(1 - cf_1 \times cg_1)$$

and $$g_{2010} = (cg_2 + cg_1 \times cf_2)/(1 - cf_1 \times cg_1).$$

The calculation process is divided into the following parts:

1. The calculation of the coefficients F_{nm} and G_{nm}.
2. a. The calculation of g_{0011}, f_{0101}, g_{1010} and r.

   b. The calculation of f_{1011}, f_{0111} and f_{0201} in terms of g_{2010}, for which the identifier G2010 is substituted.
   The calculation of g_{0111}, g_{1011} and g_{2010} in terms of f_{0201}, for which the identifier F0201 is substituted.

   c. The calculation of f_{0201} and g_{2010}.
   Note that in the calculations 2b and 2c, the already calculated g_{0011}, f_{0101}, g_{1010} and r are treated as if they were not calculated; i.e. they are treated as algebraic variables.
3. The final results for f_{0201} and g_{2010}, where the values of g_{0011}, f_{0101}, g_{1010} and r are substituted.

Part 1 is executed by formula program RPR 301067/01, which produces formula program RPR 301067/02, which executes part 2 and whichin turn produces formula program RPR 301067/03, which executes part 3.

Remarks

1. The reason for introducing some coefficients equal to zero in the truncated power-series of f and g, instead of lowering the degree of these power-series, is that f and g are multiplied in F with a truncated power-series of which the constant term is equal to zero.
2. In defining output by means of "PR STRING", it should be kept in mind that two succeeding right brackets are output as one bracket; hence,
   "PR STRING(... PR STRING(...))))...));););"
   produces the output:
   "... PR STRING(...))...);"
   which in turn produces the output:
   "...)..."
3. The reason for the extremely long computation times for RPR 301067/02 and /03 is the occurrence of quotients, due to which much time is wasted in the search for common factors.

Formula program RPR 301067/01 The Göbel-problem

```
(8192,  100,   0,    200,   2,   ₁₀-10,  ₁₀-10,  85,   0)
EXPAND;
x:= TPS(delta,1,-1,0); y:= TPS(delta,TPS(eps,1,-1,0),0,0);
tau:= TPS(delta,TPS(eps,0,k1,0),k2,0);
phi1:= 1 + a × tau + b × tau↑2;
phi2:= 1 + A × tau + B × tau↑2;
f0011:= 1 - g0011;
f:= TPS(delta,TPS(eps,f0011,f0111,0),
        TPS(eps,f1011,0),0);
g:= TPS(delta,TPS(eps,g0011,g0111,0),
        TPS(eps,g1011,0),0);
f1:= TPS(delta,TPS(eps,0,f0101,f0201),0,0);
F1:= TPS(delta,TPS(eps,0,f0101,F0201),0,0);
g1:= TPS(delta,0,g1010,g2010);
G1:= TPS(delta,0,g1010,G2010);
```

```
F:= (phil - x) × f + phil × (G1 - f1 + TPS(delta,0,-p × r,0));
G:= (phi2 - y) × g + phi2 × (F1 - g1 + TPS(delta,
        TPS(eps,0,-q × r,0),0,0));

COEFF(F,F0,F1,F2);
COEFF(F0,F00,F01,F02);
COEFF(F1,F10,F11);
COEFF(F2,F20);
COEFF(G,G0,G1,G2);
COEFF(G0,G00,G01,G02);
COEFF(G1,G10,G11);
COEFF(G2,G20);

NOT EXP;
PR STRING(
Formula program RPR 301067/02 The Göbel-problem
(8192,   100,   0,   0,   0,   10-10,   10-10,   40,   0))
PR STRING(
Formula program RPR 301067/03 The Göbel-problem
(8192,   100,   0,   0,   0,   10-10,   10-10,   40,   0))))
PR STRING(
Results RPR 301067/03 The Göbel-problem))));
EXPAND; FIX;));
EXPAND; FIX;
SOL LIN EQ(-4,g0011,f0101,g1010,r,);
NLCR; OUTPUT R(?:= F10); PR STRING(,);
NLCR; OUTPUT R(?:= F01); PR STRING(,);
NLCR; OUTPUT R(?:= G10); PR STRING(,);
NLCR; OUTPUT R(?:= G01); PR STRING());
ERASE; FIX; NLCR;); NLCR;
```

```
OUTPUT R(F:= F20); PR STRING(
f1011:= - SUBST(F,f1011,0))/DER(F,f1011));););
OUTPUT R(F:= F11); PR STRING(
f0111:= - SUBST(F,f0111,0))/DER(F,f0111));););
OUTPUT R(F:= F02); PR STRING(
f0201:= - SUBST(F,f0201,0))/DER(F,f0201));
);
OUTPUT R(G:= G02); PR STRING(
g0111:= - SUBST(G,g0111,0))/DER(G,g0111));););
OUTPUT R(G:= G11); PR STRING(
g1011:= - SUBST(G,g1011,0))/DER(G,g1011));););
OUTPUT R(G:= G20); PR STRING(
g2010:= - SUBST(G,g2010,0))/DER(G,g2010));
cf1:= DER(f0201,G2010));
cf2:= SUBST(f0201,G2010,0));
cg1:= DER(g2010,F0201));
cg2:= SUBST(g2010,F0201,0));
den:= 1 - cf1 × cg1; PR STRING(
FIX;
OUTPUT R(f0201:= ));
FIX; OUTPUT R(?:= (cf2 + cf1 × cg2))/den)); ERASE;
PR STRING())));
ERASE;
OUTPUT R(g2010:= ));
OUTPUT R(?:= (cg2 + cg1 × cf2))/den));
PR STRING())));
END;));
END;);
END;
```

```
Formula program RPR 301067/02 The Gobel-problem
(8192,   100,   0,   0,   0,   ₁₀-10,   ₁₀-10,   40,   0)
PR STRING(
Formula program RPR 301067/03 The Gobel-problem
(8192,   100,   0,   0,   0,   ₁₀-10,   ₁₀-10,   40,   0))
PR STRING(
Results RPR 301067/03 The Gobel-problem));
EXPAND; FIX;);
EXPAND; FIX;
SOL LIN EQ(-4,g0011,f0101,g1010,r,
1+(-1)×g0011+a×k2+a×k2×(-1)×g0011+g1010+(-1)×p×r,
a×k1+a×k1×(-1)×g0011+(-1)×f0101,
A×k2×g0011+(-1)×g1010,
A×k1×g0011+g0011+f0101+(-1)×q×r);
ERASE; FIX; NLCR;

F:= f1011+f1011×a×k2+b×k2∧2+b×k2∧2×(-1)×g0011+G2010+g1010×a×k2+
        (-1)×p×r×a×k2;
f1011:= - SUBST(F,f1011,0)/DER(F,f1011);
F:= f1011×a×k1+(-2)×k1×k2×b×g0011+2×k1×k2×b+k2×a×f0111+f0111+g1010×
        a×k1+(-1)×p×r×a×k1+a×k2×(-1)×f0101;
f0111:= - SUBST(F,f0111,0)/DER(F,f0111);
F:= a×k1×f0111+b×k1∧2+b×k1∧2×(-1)×g0011+(-1)×f0201+a×k1×(-1)×f0101;
f0201:= - SUBST(F,f0201,0)/DER(F,f0201);

G:= A×k1×g0111+g0111+B×k1∧2×g0011+F0201+A×k1×f0101+A×k1×(-1)×q×r;
g0111:= - SUBST(G,g0111,0)/DER(G,g0111);
G:= g1011×A×k1+g1011+A×k2×g0111+B×2×k1×k2×g0011+(-1)×g1010×A×k1+A×
        k2×f0101+A×k2×(-1)×q×r;
g1011:= - SUBST(G,g1011,0)/DER(G,g1011);
G:= g1011×A×k2+B×k2∧2×g0011+(-1)×g2010+(-1)×g1010×A×k2;
g2010:= - SUBST(G,g2010,0)/DER(G,g2010);
```

```
cf1:= DER(f0201,G2010);
cf2:= SUBST(f0201,G2010,0);
cg1:= DER(g2010,F0201);
cg2:= SUBST(g2010,F0201,0);
den:= 1 - cf1 × cg1; PR STRING(
FIX;
OUTPUT R(f0201:= );
FIX; OUTPUT R(?:= (cf2 + cf1 × cg2)/den); ERASE;
PR STRING());
ERASE;
OUTPUT R(g2010:= );
OUTPUT R(?:= (cg2 + cg1 × cf2)/den);
PR STRING());
END;);
END;
ready
line number = 74 execution time = 51 sec
```

```
Formula program RPR 301067/03 The Göbel-problem
(8192,  100,   0,   0,   0,   ⏨-10,  ⏨-10,  40,   0)
PR STRING(
Results RPR 301067/03 The Göbel-problem);
EXPAND; FIX;
g0011:= (-a×k2×q+a×p×k1-q)/(-a×k2×q+a×p×k1+k2×A×q-p×k1×A-p-q);
f0101:= (a×k2×k1×A×q-a×p×k1↑2×A-a×p×k1)/(-a×k2×q+a×p×k1+k2×A×q-p×
        k1×A-p-q);
g1010:= (-a×k2↑2×A×q+a×k2×p×k1×A-k2×A×q)/(-a×k2×q+a×p×k1+k2×A×q-
        p×k1×A-p-q);
r:= (-a×k2-k1×A-1)/(-a×k2×q+a×p×k1+k2×A×q-p×k1×A-p-q);

FIX;
OUTPUT R(f0201:= (a↑2×k2↑2×r×k1↑2×A↑2×q+a↑2×p×r×k1↑4×A↑2-a↑2×
        k2↑2×k1↑2×f0101×A↑2-a↑2×k2×g1010×k1↑3×A↑2-a↑2×k2×k1↑3×
        f0101×A↑2-a↑2×g1010×k1↑4×A↑2+a↑2×k2↑2×g0011×k1↑2×B+2×
        a↑2×p×r×k1↑3×A-b×g0011×k1↑4×A↑2-a↑2×k2×g1010×k1↑2×A-2×
        a↑2×k2×k1↑2×f0101×A-2×a↑2×g1010×k1↑3×A-a×k1↑3×f0101×A↑2+
        b×k1↑4×A↑2+a↑2×p×r×k1↑2-2×b×g0011×k1↑3×A-a↑2×k2×k1×f0101-
        a↑2×g1010×k1↑2-2×a×k1↑2×f0101×A+2×b×k1↑3×A-b×g0011×k1↑2-
        a×k1×f0101+b×k1↑2)/(2×a↑2×k2↑2×k1×A+2×a×k2×k1↑2×A↑2+a↑2×
        k2↑2+4×a×k2×k1×A+k1↑2×A↑2+2×a×k2+2×k1×A+1));
ERASE;
OUTPUT R(g2010:= (a↑2×k2↑4×r×A↑2×q+a↑2×k2↑2×p×r×k1↑2×A↑2-a↑2×
        k2↑4×f0101×A↑2-a↑2×k2↑3×g1010×k1×A↑2-a↑2×k2↑3×k1×f0101×
        A↑2-a↑2×k2↑2×g1010×k1↑2×A↑2+a↑2×k2↑4×g0011×B+2×a×k2↑3×r×
        A↑2×q-k2↑2×b×g0011×k1↑2×A↑2-a↑2×k2↑3×g1010×A-2×a×k2↑3×f0101×
        A↑2-2×a×k2↑2×g1010×k1×A↑2-a×k2↑2×k1×f0101×A↑2+k2↑2×b×k1↑2×
        A↑2+2×a×k2↑3×g0011×B+k2↑2×r×A↑2×q-2×a×k2↑2×g1010×A-k2↑2×
        f0101×A↑2-k2×g1010×k1×A↑2+k2↑2×g0011×B-k2×g1010×A)/(2×a↑2×
        k2↑2×k1×A+2×a×k2×k1↑2×A↑2+a↑2×k2↑2+4×a×k2×k1×A+k1↑2×A↑2+
        2×a×k2+2×k1×A+1));
END;
ready
line number = 45 execution time = 1138 sec
```

```
Results RPR 301067/03 The Gobel-problem
f0201:= (-a↑3×k2↑3×q×k1↑2×B+a↑3×k2↑2×p×k1↑3×B+k2×q×k1↑4×A↑3×b-
        p×k1↑5×A↑3×b-a↑2×k2↑2×q×k1↑2×B+2×k2×q×k1↑3×A↑2×b-3×p×k1↑4×
        A↑2×b+k2×q×k1↑2×A×b-3×p×k1↑3×A×b-p×k1↑2×b)/(-2×a↑3×k2↑3×q×
        k1×A+2×a↑3×k2↑2×p×k1↑2×A+2×a↑2×k2↑3×q×k1×A↑2-2×a↑2×k2↑2×
        q×k1↑2×A↑2-2×a↑2×k2↑2×p×k1↑2×A↑2+2×a↑2×k2×p×k1↑3×A↑2+2×a×
        k2↑2×q×k1↑2×A↑3-2×a×k2×p×k1↑3×A↑3-a↑3×k2↑3×q+a↑3×k2↑2×p×
        k1+a↑2×k2↑3×q×A-6×a↑2×k2↑2×q×k1×A-3×a↑2×k2↑2×p×k1×A+4×a↑2×
        k2×p×k1↑2×A+4×a×k2↑2×q×k1×A↑2-3×a×k2×q×k1↑2×A↑2-6×a×k2×p×
        k1↑2×A↑2+a×p×k1↑3×A↑2+k2×q×k1↑2×A↑3-p×k1↑3×A↑3-3×a↑2×k2↑2×
        q-a↑2×k2↑2×p+2×a↑2×k2×p×k1+2×a×k2↑2×q×A-6×a×k2×q×k1×A-6×a×
        k2×p×k1×A+2×a×p×k1↑2×A+2×k2×q×k1×A↑2-q×k1↑2×A↑2-3×p×k1↑2×
        A↑2-3×a×k2×q-2×a×k2×p+a×p×k1+k2×q×A-2×q×k1×A-3×p×k1×A-q-p);
g2010:= (-a↑3×k2↑5×q×B+a↑3×k2↑4×p×k1×B+k2↑3×q×k1↑2×A↑3×b-k2↑2×p×k1↑3×
        A↑3×b-3×a↑2×k2↑4×q×B+2×a↑2×k2↑3×p×k1×B-k2↑2×p×k1↑2×A↑2×b-3×
        a×k2↑3×q×B+a×k2↑2×p×k1×B-k2↑2×q×B)/(-2×a↑3×k2↑3×q×k1×A+2×a↑3
        ×k2↑2×p×k1↑2×A+2×a↑2×k2↑3×q×k1×A↑2-2×a↑2×k2↑2×q×k1↑2×A↑2-2×
        a↑2×k2↑2×p×k1↑2×A↑2+2×a↑2×k2×p×k1↑3×A↑2+2×a×k2↑2×q×k1↑2×A↑3
        -2×a×k2×p×k1↑3×A↑3-a↑3×k2↑3×q+a↑3×k2↑2×p×k1+a↑2×k2↑3×q×A-
        6×a↑2×k2↑2×q×k1×A-3×a↑2×k2↑2×p×k1×A+4×a↑2×k2×p×k1↑2×A+4×a×
        k2↑2×q×k1×A↑2-3×a×k2×q×k1↑2×A↑2-6×a×k2×p×k1↑2×A↑2+a×p×k1↑3×
        A↑2+k2×q×k1↑2×A↑3-p×k1↑3×A↑3-3×a↑2×k2↑2×q-a↑2×k2↑2×p+2×a↑2×
        k2×p×k1+2×a×k2↑2×q×A-6×a×k2×q×k1×A-6×a×k2×p×k1×A+2×a×p×k1↑2
        ×A+2×k2×q×k1×A↑2-q×k1↑2×A↑2-3×p×k1↑2×A↑2-3×a×k2×q-2×a×k2×
        p+a×p×k1+k2×q×A-2×q×k1×A-3×p×k1×A-q-p);
ready
line number = 31 execution time = 5059 sec
```

3.3. The formula-program processor

In this section we describe and reproduce the formula-program processor. Remark: it is not a "sine qua non" to read this section in order to make a formula program.

The processor reads input tape; it performs formula manipulations by means of the general-formula-manipulation system of section 2, and it produces output.

The input tape has the following structure:

<input tape>::= <preparatory information>;<formula program>|<input tape> <formula program>

<preparatory information>::= <special symbols><special identifiers>

<special symbols>::=+-×/=↑:;<space symbol>()<tabulator symbol>i·$_{10}$? <nlcr symbol>

<special identifiers>::= <formula-statement identifiers>+<special formula-identifiers>

<formula-statement identifiers>::= REAL, SPEC DER, OUTPUT C, OUTPUT R, FIX, ERASE, ER B RET, NLCR, PR STRING, EXPAND, NOT EXP, COEFF, SOL LIN EQ, END

<special formula-identifiers>::= TPS, exp, ln, sin, cos, arctan, sqrt, DER, SIMPL, CC, SUBST, QUOT, COMM DIV

(Note that an nlcr symbol is a new-line-carriage-return symbol.)

The processor reads, first of all, the preparatory information and proceeds then to read and execute the formula programs which occur on the input tape.

3.3.1. The first part of the processor

The first part of the processor is now reproduced.

```
begin comment Formula program processor. Second corrected version.
    R 1050 RPR 280267/03/02;
  integer idp,s,next symbol,TIME,numb of id,numb of form st id,
```

```
  numb of spec id,numb of spec der,plus,minus,prod,quot,ttp,equal,hat,
  comma,colon,semicolon,space,left br,right br,tab,nlcr,bar,underlining,
  iu,imag unit,lower ten,point,question mark,line counter;
integer array SPEC ID[1:30,-1:1];

comment next follow the output procedures to be used in the
  formula-program processor;
procedure PR string(s); string s;
comment PUTEXT and PRINTTEXT are MC procedures for resp.
  punching and printing a string;
begin PUTEXT(s); PRINTTEXT(s) end;
procedure PR nlcr;
PR string(≮
≯);
procedure PR sym(s); value s; integer s;
comment PUSYM and PRSYM are the MC procedures for resp.
  punching and printing a symbol, whose internal representation
  is given by the actual value of s;
begin PUSYM(s); PRSYM(s) end;
procedure PR int num(i); value i; integer i;
begin integer n,m;
  if i = 0 then begin PR string(≮0≯); goto OUT end;
  if i < 0 then begin PR string(≮-≯); i:= -i end;
  m:= 1;
AGAIN: n:= i : m; if n < 10 then
  begin PR sym(n); if m = 1 then goto OUT;
    i:= i - n × m; m:= m/10
  end else m:= m × 10; goto AGAIN;
OUT: end PR int num;

comment next follows the input procedure RE sym, which becomes
  equal to the internal representation of the symbol read from
  input tape;
```

```
integer procedure RE sym;
begin integer i; RE sym:= i:= RESYM; comment RESYM is the MC
  procedure for reading a symbol from input tape;
  if i = nlcr then line counter:= line counter + 1
end RE sym;

procedure ERROR(B,s); Boolean B; string s;
if B then
begin PR nlcr; PR string(s); EXIT end ERROR;

procedure EXIT;
begin PR nlcr; PR string(⪡line number = ⪢); PR int num(line counter);
  PR string(⪡ execution time = ⪢); PR int num(time - TIME);
  PR string(⪡ sec⪢); comment the time is measured by the
  MC procedure time in seconds; goto START
end EXIT;

procedure NS;
begin again: next symbol:= RE sym; if next symbol = space ∨
  next symbol = tab ∨ next symbol = nlcr then goto again;
  if next symbol = bar then
  begin ERROR(RE sym ≠ hat,⪡|not followed by ∧⪢);
    next symbol:= ttp
  end else if next symbol = underlining then
  begin ERROR(RE sym ≠ iu,⪡_not followed by i⪢);
    next symbol:= imag unit
end end NS;

integer procedure ID(id list); integer array id list;
begin integer i,ns,p; integer array id[0:1]; id[0]:= id[1]:= 0; p:= -1;
A: p:= p + 1; ERROR(p > 7,⪡id too long⪢); ns:= next symbol + 1;
  ERROR(ns > 63,⪡wrong symbol in id⪢);
  id[p:4]:= id[p:4] + ns × 64↑(3-p+p:4×4); NS;
  for i:= plus,minus,prod,quot,colon,ttp,comma,semicolon,
  left br,right br do begin if next symbol = i then goto END ID end;
  goto A;
```

```
END ID: for p:= 1 step 1 until numb of spec id do
  begin if id[0] = SPEC ID[p,0] ∧ id[1] = SPEC ID[p,1] then
    begin ID:= p; goto OUT end
  end; for p:= numb of spec id + 1 step 1 until idp do
  begin if id[0] = id list[p,0] ∧ id[1] = id list[p,1] then
    begin ID:= p; goto OUT end
  end;
  ID:= idp:= idp + 1; ERROR(idp > numb of id,⊀too much id⊁);
  for p:= 0,1 do id list[idp,p]:= id[p]; id list[idp,-1]:= -1;
OUT: end ID;

plus:= RESYM; minus:= RESYM; prod:= RESYM; quot:= RESYM;
equal:= RESYM; bar:= RESYM; hat:= RESYM; comma:= RESYM;
colon:= RESYM; semicolon:= RESYM; space:= RESYM; left br:=
RESYM; right br:= RESYM; tab:= RESYM; underlining:= RESYM;
iu:= RESYM; point:= RESYM; lower ten:= RESYM; question mark:=
RESYM; nlcr:= RESYM; ttp:= 500; imag unit:= 600;

numb of id:= 30; idp:= line counter:= numb of spec id:= 0;
A: NS; ID(SPEC ID); if next symbol = plus then
  begin numb of form st id:= idp; goto A end;
  if next symbol ≠ semicolon then goto A; numb of spec id:= idp;
  s:= -1;
START: next symbol:= RESYM; if ¬(next symbol = left br ∧ s =
  underlining) then begin s:= next symbol; goto START end;
  idp:= numb of spec id; TIME:= time;
  NEW PAGE; STOPCODE; comment the MC procedure NEW PAGE
  has as effect that the next symbol to be output is printed on a new
  page. The MC procedure STOPCODE has as effect the punching
  of a stopcode symbol and a piece of blank tape;

 begin procedure PRINTTEXT(s); PR string(s);
   procedure NLCR; PR nlcr;
   procedure ABSFIXT(n,m,x); PR int num(x);
   comment at this place the general-formula-manipulation system
   of chapter 2 should be inserted;
```

The following list gives the meaning of the declared variables.

idp: is the pointer of the list of identifiers.

s: is only an auxiliary variable.

next symbol: gives the value of the internal representation of the last read symbol, which should be treated next.

TIME: gives the time (in seconds) at which the execution of a formula program has been started.

numb of id: gives the number of all the different identifiers which have been read from the beginning of the formula program, under execution, until the point just read; this number includes the special identifiers.

numb of form st id: gives the number of formula-statement identifiers.

numb of spec id: gives the number of special identifiers.

numb of spec der: is the pointer of the array in which explicit derivatives are stored; at the very beginning of the execution of the formula program, *numb of spec der* becomes equal to "numb of special derivatives" as given in the heading of the formula program.

plus, ..., *question mark*: are the values of the internal representations of the special symbols.

line counter: gives the number of lines of the formula program beginning immediately after the heading until the point just read.

The procedures *PR string*, *PR nlcr*, *PR sym* and *PR int num* are used to: print, by means of a line printer,and to punch on output tape, a string; an nlcr symbol; a symbol (whose internal representation is given by the value of *s*);and an integral number.

The procedure *RE sym* reads a symbol from input tape and becomes equal to the value of the internal representation of that symbol; an occurrence of an nlcr symbol leads to an increase of the *line counter*.

The procedures *ERROR* and *EXIT* have obvious meanings.

The procedure *NS* reads a symbol from input tape and assigns the value of its internal representation to the variable *next symbol*; lay-out symbols

are discarded; moreover, the symbols "↑" and "i̲" are recognized as the occurrence of "|" followed by "∧" and of "_" followed by "i".

The procedure *ID*, with parameter *id list*, is used to read an identifier, either special or not. The identifier read, which is, according to the requirements of section 3.2.2.2, composed of no more than 8 letters and/or digits, is stored into two machine words: the array elements *id*[*0*] and *id*[*1*].
Use is made of the fact that the values of the internal representation of the letters and digits lie between 0 and 62.
A difficulty arises from the fact that the digit "0" has as internal representation the value 0, while the absence of a letter or digit is recognized by the fact that a group of 6 bits have also the value 0; therefore, the values of the internal representations of letters and digits are augmented with 1.
After the storage of the identifier into *id*[*0*] and *id*[*1*], *ID* determines whether the identifier occurs in *SPEC ID*, or whether it occurs in *id list* (that is to say, occurs in the array which is substituted for *id list* as actual parameter); if neither possibility holds true, then the identifier is evidently a new one and is stored into *id list*.

During the reading of the special identifiers of the preparatory information from the input tape, the procedure *ID* is called with *SPEC ID* as actual parameter. During the reading of a formula program, *ID* is called with the array *identifier list*, whose length is determined by "numb of identifiers" occurring in the heading of the formula program, as actual parameter. The value of the procedure identifier *ID* is equal to the index at which the identifier read is stored; if this value is between 1 and *numb of spec id*, then the identifier is a special identifier; if, on the other hand, this value is greater than *numb of spec id*, then the identifier is either a formula identifier or an algebraic variable.
In order to know elsewhere (the procedure *primary*) in the processor, whether the identifier was a new one, *identifier list*[*idp*,*-1*] is set equal to -1 if this is indeed the case. That this is sufficient

follows from the fact that for an "old" identifier, with index k, *identifier list*[k,-7] is equal to the nonnegative index of a formula (or algebraic variable), as stored in the general-formula-manipulation system.

The remainder of the first part of the processor can now easily be traced:

The special symbols are read.

The special identifiers are read; and the variables *numb of form st id* and *numb of spec id* get their correct values (note that the special structure chosen enables one to delete or add special identifiers very easily; it is, moreover, very easy to change the special identifiers into possibly more convenient ones).

After the label *START*, the input tape is read until the opening bracket "(" of the heading of a formula program has been found, after which the general-formula-manipulation system reads the first seven numbers of the heading of the formula program.

3.3.2. The second part of the processor

The second part of the processor is considerably larger than the first part; we shall discuss and reproduce it,therefore,in several sections.

First some initial statements and some declarations are given:

```
INITIALIZE;
numb of id:= read + idp; numb of spec der:= read;
begin switch CASE:= Real,Spec der,Output c,Output r,Fix,Erase,
  Erase but retain,Nlcr,Pr string,Expand,Not expand,
  Tps coeff,Solve linear equations,End;
  integer array identifier list[idp:numb of id,-1:1],
  spec der[0:numb of spec der,1:2];

  integer procedure IDENTIFIER; IDENTIFIER:= ID(identifier list);
```

3.3.2.1. The input of a formula

In order to read a formula from input tape, the following procedures are declared.

```
Boolean procedure digit next symbol;
digit next symbol:= next symbol < 10;

integer procedure real number;
begin real r,e;
   real procedure unsigned integer;
   begin real i; i:=next symbol;
   A:NS; if digit next symbol then
     begin i:=i × 10 + next symbol; goto A end;
     unsigned integer:=i
   end unsigned integer;
   r:=1; e:=0;
   if next symbol ≠ lower ten then
   begin real l,r1,r2; r1:=r2:=0;
     if digit next symbol then r1:=unsigned integer;
     if next symbol = point then
     begin l:=10; NS; ERROR(¬ digit next symbol,
       ⊀. not followed by unsigned integer⊁);
     A:if digit next symbol then
       begin r2:=r2 + next symbol/l; l:=l × 10; NS; goto A end
     end;
     r:=r1 + r2
   end;
   if next symbol = lower ten then
   begin real sign; sign:=+1;
     NS; if next symbol = plus then NS else
     if next symbol = minus then begin NS; sign:=-1 end;
     ERROR( ¬ digit next symbol,⊀10 not followed by integer⊁);
     e:=sign × unsigned integer
   end;
   real number:=RN(r × 10↑e)
end real number;
```

```
integer procedure primary;
begin integer p,d,s; if next symbol = imag unit then
  begin NS; primary:= im unit end else
  if digit next symbol ∨ next symbol = point
    ∨ next symbol = lower ten then
  primary:= real number else if next symbol = left br then
  begin NS; primary:= formula; ERROR(next symbol ≠ right br,
    ⌈) missing⌉); NS
  end else
  begin switch CASE:= Tps,Exp,Ln,Sin,Cos,Arctan,Sqrt,
    Der,Simpl,Compl conj,Subst,Quot,Comm div;
    s:= IDENTIFIER; goto if s > numb of spec id then Identifier
    else CASE[s - numb of form st id];
  Tps:
    begin integer x,i; integer array coeff[0:FTPSm2];
      NS; x:= formula; i:= -1;
    A: i:= i + 1; NS; ERROR(i > FTPSm2,⌈degree too large⌉);
      coeff[i]:= formula; if next symbol ≠ right br then goto A;
      primary:= TPS(d,i,x,coeff[d]); NS
    end; goto END;
  Exp: primary:= EXP(primary); goto END;
  Ln: primary:= LN(primary); goto END;
  Sin: primary:= SIN(primary); goto END;
  Cos: primary:= COS(primary); goto END;
  Arctan: primary:= ARCTAN(primary); goto END;
  Sqrt: primary:= SQRT(primary); goto END;
  Der: NS; d:= formula; NS; primary:= DER(d,formula); NS; goto END;
  Simpl: primary:= SIMPLIFY(primary); goto END;
  Compl conj: primary:= CC(primary); goto END;
  Subst: NS; s:= formula;
    begin integer array arg,val[1:numb of id - numb of spec id]; p:= 0;
    A: NS; p:= p + 1; arg[p]:= formula; ERROR(TYPE(arg[p],di,di)
        ≠ algebraic variable,⌈arg in subst not alg var⌉);
      NS; val[p]:= formula; if next symbol ≠ right br then goto A;
      primary:= SUBSTITUTE(s,d,1,p,arg[d];val[d]); NS
    end; goto END;
```

```
  Quot: NS; p:= formula; NS; d:= formula; NS; s:= IDENTIFIER;
    primary:= QUOTIENT(p,d,identifier list[s,-1]); NS; goto END;
  Comm div: NS; p:= formula; NS; primary:= COMMON DIVISOR(p,formula);
    NS; goto END;
  Identifier: if identifier list[s,-1] ≠ -1 then primary:=
    identifier list[s,-1] else primary:= identifier list[s,-1]:=
    STORE(complex,algebraic variable,s);
  END: end end primary;

integer procedure factor;
begin integer e,a,b,p; p:= primary;
again: if next symbol = ttp then
  begin NS; e:= primary; if TYPE(e,a,b) = number ∧ a = integer then    *)
    begin VAL OF INT NUM(e,b); p:= INT POW(p,b) end
    else p:= POWER(p,e); goto again
  end; factor:= p
end factor;

integer procedure term;
begin integer f; f:= factor;
again: if next symbol = prod then begin NS; f:= P(f,factor); goto again end;
    if next symbol = quot then begin NS; f:= Q(f,factor); goto again end;
  term:= f
end term;

integer procedure formula;
begin integer t; if next symbol = minus then
  begin NS; t:= P(minone,term) end else
  if next symbol = plus then begin NS; t:= term end else t:= term;
again: if next symbol = plus then begin NS; t:= S(t,term); goto again end;
  if next symbol = minus then begin NS; t:= D(t,term); goto again end;
  formula:= t
end formula;
```

*) Note that *a* obtains a value by means of *TYPE*.

Of these procedures, *digit next symbol* and *real number* depend on the special internal representation of digits: the digits "0", "1", ..., "9" have as internal representation the values 0, 1, ..., 9, respectively.
A call of the procedures *real number*, *primary*, *factor*, *term* or *formula* has as effect that a real number, a primary, a factor, a term or a formula is read from input tape; the procedure identifiers become equal to the indices of the stored formulae in the general-formula-manipulation system. For a good understanding of these procedures it is necessary to recall the definition of formula in section 3.2.1.
The form of the procedures *factor*, *term* and *formula* was inspired by F.E.J. Kruseman Aretz [7].

3.3.2.2. <u>The output of a formula</u>

Several forms for the output of a formula are desired; they are described in sections 3.3.2.2.1, 3.3.2.2.2 and 3.3.2.2.3.

3.3.2.2.1. <u>The optimal form</u>

The simplified formula is output in standard form with as few symbols as possible. This form is produced by the procedure *OUTPUT R*. It is evident that this form is only of importance when the processor is in the expanding state; i.e. *expand* = <u>*true*</u>, for,otherwise, simplification is not possible.

The method by which the output is defined is: first, constructing the second representation of the formula (see section 2.10), and simplifying this representation by means of *SIMPL 2 REPR* such that the second representation has the standard form; and second, producing the output from the second representation. We remark that the construction of the second representation is necessary; simplification by means of *SIMPLIFY* only is insufficient, since then the formula is not necessarily put into the standard form.

The procedure *OUTPUT R* defines output only if the formula is a sum or a product; for other types of formulae, the output is produced by a call of *OUTPUT RC(f,2,0)*.
It may occur that there is no space left for the second representation,

or that the formula to be output is already simplified, and it is not desirable to use the above process for obtaining the output, then the process may be used as described in section 3.3.2.2.2.

3.3.2.2.2. <u>The ordinary form</u>

The formula is output immediately from its tree-structure representation. It is allowed that, e.g. "(a + (b × (-1)))" be output as "a + b × (-1)", but it is not allowed that it be output as it stands; i.e. superfluous brackets should be avoided. (Note that the optimal form for the output of the above formula is "a - b"; note, moreover, that the procedure *OUTPUT* of section 2.13 would produce "(a + (b × (-1)))" if *expand* = <u>*false*</u>.)
It is the task of the procedure *OUTPUT RC*, to produce this type of output.
Let us investigate the problem of defining output,with no superfluous brackets,more closely.
Let the type of a formula "f" be a sum, a difference, a product, a quotient, a power or an integral power.
Let "f" be the lhs or rhs of another formula "g", which is also a sum, ..., or an integral power.
Consider now the following table, in which the first column gives the type of "g" and the place of "f" in "g"; in which the first row gives the type of "f"; and in which the symbol "x" indicates that brackets should be placed around the output of "f". (The last column will be explained below.)

	+	−	×	/	↑	↑ n	type of the surrounding
f + ...							*plus*
... + f							*plus*
f − ...							*plus*
... − f	x	x					*product*
f × ...	x	x					*product*
... × f	x	x					*product*
f / ...	x	x					*product*
... / f	x	x	x	x			*power*
f ↑ ...	x	x	x	x			*power*
... ↑ f	x	x	x	x	x	x	*power* + 1
f ↑ n	x	x	x	x	x	x	*power*

table 3.1 Occurrence of brackets

For each possible occurrence of "f" in "g", we define the "type of the surrounding of f", using the last column of the table. If "f" does not occur as lhs or rhs of "g", but stands alone, or is the argument of a function, then the "type of the surrounding" is set equal to zero.
Using the fact that the values of the variables: *number*, *tr pow series*, *function*, *algebraic variable*, *sum*, *difference*, *product*, *quotient*, *power* and *integral power* are 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10, respectively, it is now very easy to determine when brackets should be placed around the output of "f"; namely then when the type of "f" is less than the "type of its surrounding".

Example:

Let f = a − b, g = f − c; then the "type of the surrounding of f" is *sum*; from *difference* ≥ *sum*, it follows that no brackets are necessary.
Let f = a/b, g = f ↑ c; then the "type of the surrounding of f" is *power*; from *quotient* < *power* it follows that brackets are needed.

The form of the procedure which produces the desired output is now exemplified by the following procedure *OP*, which produces output only if *f* is a sum, difference or algebraic variable. The actual value of its parameter *type* is the "type of the surrounding of *f*".

```
procedure OP(f,type); value f, type; integer f, type;
begin integer t, a, b;
   procedure LBR; if t < type then PR string(⌈(⌉);
   procedure RBR; if t < type then PR string(⌈)⌉);
   t:= TYPE(f,a,b);
   if t = sum then
   begin LBR; OP(a,sum); PR string(⌈+⌉); OP(b,sum); RBR end
   else if t = difference then
   begin LBR; OP(a,sum); PR string(⌈-⌉); OP(b,product); RBR end
   else OUTPUT VARIABLE(f)
end OP;
```

If *f* is some formula then a call *OP(f,0)* leads to the desired output, as can be verified easily.

3.3.2.2.3. The complex form

Finally, the form of the output may be chosen to be complex, for which the procedures *COUNT*, *REAL* and *OUTPUT C* are used.

"a + b" is output as "S(a,b)";
"a - b" is output as "D(a,b)";
"a × b" is output as "P(a,b)";
"a / b" is output as "Q(a,b)";
"a ↑ b" is output as "POWER(a,b)";
"a ↑ n" is output as "INT POW(a,n)" (n is an integer);
"x + a" is output as "SR(x,a)", if "x" is real;
"x × a" is output as "PR(x,a)", if "x" is real;
the elementary function symbols are output in capital letters;
a complex number "R + iI", unequal to the imaginary unit, is output as "CN(R,I)"; the imaginary unit is output as "iu";
a truncated power-series is output as a sum of products.

In order to combine real parts of a formula and to output them in a special form, namely enclosed by "RN(" and ")", the procedure *REAL* is used to determine whether a certain formula has real constituents only and is not composed of multi-valued functions (ln, arctan or sqrt). The procedure *COUNT* is used to count the number of terms in a sum or the number of factors in a product and to store them into an array. The procedure *OUTPUT C* produces the output of a formula if it is a sum or a product (only in these cases are real parts combined and reordered). If the formula is not a sum or a product, then *REAL* determines whether it is real, in which case the output, in ordinary form, is produced by a call of *OUTPUT RC(f,1,0)*; this output is enclosed between the symbols "RN(" and the symbol ")"; if the formula is not real, output is produced by a call of *OUTPUT RC(f,3,0)*.

In the above discussion we have seen that the procedure *OUTPUT RC* plays a key rôle.
If *OUTPUT R* is unable to produce output it calls *OUTPUT RC*; i.e. if ¬ *expand* then a call *OUTPUT RC(f,1,0)* is invoked; if *expand* and the formula is not a sum or product, then a call *OUTPUT RC(f,2,0)* is invoked.
If *OUTPUT C* is unable to produce output it also calls *OUTPUT RC* in the way mentioned above.
On the other hand, *OUTPUT RC* should invoke a call of *OUTPUT R* if its second parameter, *case*, is equal to 2 and a subformula to be output is a sum or a product, and if *case* is equal to 3, it should invoke a call of *OUTPUT C* for the output of a subformula.
Examples:

1. The optimal output of f = (a + b - 2 × a)/(y + x - 2 × y) is produced in the following way:
   1. *OUTPUT R(f)* calls *OUTPUT RC(f,2,0)*
   2. *OUTPUT RC(f,2,0)* produces the symbols "(", ")/(" and ")" while the space between these symbols is filled by two calls of *OUTPUT R*, namely *OUTPUT R*(a + b - 2 × a), which leads to the output "- a + b" and *OUTPUT R*(y + x - 2 × y), which leads to the output "x - y".

The complete output has,therefore,the form: "(- a + b)/(x - y)".

2. The ordinary output of f = (a + b - 2 × a)/(y + x - 2 × y) is "(a + b - 2 × a)/(y + x - 2 × y)" and is produced after a call of *OUTPUT RC(f,1,0)*.

3. The complex output of f = exp(a + b), if "a" and "b" are algebraic variables of real type, is produced in the following way:
   1. *OUTPUT C* recognizes that "f" is real and produces the symbols "RN(", calls *OUTPUT RC(f,1,0)* and produces the symbol ")".
   2. *OUTPUT RC(f,1,0)* produces the output "exp(a + b)".

   The result is,therefore, "RN(exp(a + b))".

4. The complex output of f = exp(a + b), if "a" and "b" are algebraic variables of complex and real type,respectively, is produced as follows:
   1. *OUTPUT C* recognizes that "f" is complex and calls *OUTPUT RC(f,3,1)*.
   2. *OUTPUT RC* produces the output "EXP(" and ")"; in between it calls *OUTPUT C*(a + b).
   3. *OUTPUT C* recognizes that "b" is a real, and "a" a complex algebraic variable; it produces the output "SR("; it calls *OUTPUT RC*(b,*1,sum*); it produces the output "," ; it calls *OUTPUT RC*(a,*1,sum*) and, finally,it produces the output ")".
   4. *OUTPUT RC*(b,*1,sum*) produces the output "b".
      OUTPUT RC(a,*1,sum*) produces the output "a".

   The result is,therefore,"EXP(SR(b,a))" which it should be, since the procedure SR of the complex-arithmetic system of chapter 4 adds a real number to a complex number.

The above examples show that the structure of the output procedures is far from simple.

3.3.2.2.4. The output of numbers

Before we proceed in reproducing the procedures discussed, we shall discuss the output of real and complex numbers.

A positive real number r may be output in two different ways:

1. as a rational: "p/q", where p and q are integers;
2. as a real number in floating-point notation, i.e.
   a) the decimal point;
   b) at most twelve digits, the first one unequal to zero;
   c) the symbol "$_{10}$";
   d) the exponent of 10 in at most three digits.

The rational p/q is calculated as follows:

Let $A_{-1} = 0,\ A_0 = 1,$

$$B_{-1} = 1,\ B_0 = 0,$$

$$r_0 = r,$$

$$\left.\begin{aligned} a_n &= \text{entier}(r_{n-1} + .5) \\ r_n &= 1/(r_{n-1} - a_n) \\ A_n &= a_n A_{n-1} + A_{n-2} \\ B_n &= a_n B_{n-1} + B_{n-2} \end{aligned}\right\} \text{ for } n = 1, 2, 3, \ldots;$$

then

$$r = a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{a_4 + \cdots}}}$$

and

$$\lim_{n\to\infty} \frac{A_n}{B_n} = r.$$

The process is discontinued if

$$|A_n/B_n - r| \le \mathit{rel\ acc} \times r,$$

where *rel acc* is the relative accuracy of the general-formula-manipulation system, and the output is formed as "p/q", where $p = |A_n|$ and $q = |B_n|$. The process is discontinued also if

$$|A_n| \ge 2^{12} \vee |B_n| \ge 2^{12} \vee n > 20,$$

in which case the output is formed in floating-point notation (note that $2^{12} - 1$ is the largest integer in the formula-manipulation system).

<u>Remark</u>: if *rel acc* = $_{10}{-10}$, then 3.14159265359 is output in floating-point notation.

A real number "r" may occur in several different places in a formula "f"; this has consequences on the form of the output.

<u>Examples</u>:

Let	r = $_{10}10$; the output is:	r = $-_{10}10$; the output is:	r = 2/5; the output is:	r = - 2/5; the output is:
f = r	$_{10}10$	$-_{10}10$	2/5	- 2/5
f = a + r	$+_{10}10$	$-_{10}10$	+ 2/5	- 2/5
f = r × a	$_{10}10$	$-_{10}10$	2/5	- 2/5
f = a × r × b	$_{10}10$	$(-_{10}10)$	2/5	(- 2/5)
f = a ↑ r	$_{10}10$	$(-_{10}10)$	(2/5)	(- 2/5)

The need for an extra "+" symbol, the need for brackets around negative numbers and the need for brackets around rationals, is determined by the actual values of the Boolean parameters: *with sign*, *with br1* and *with br2*, respectively, of the procedure *PR real num*.

The output of a complex number is produced by *PR comp num*, which also has three Boolean parameters: *with sign*, *with br1* and *with br2*.
The following table gives, for some examples, the form of the output of a complex number "cn".
It is assumed that cn = R + iI, that

$$f_1 = cn,\ f_2 = cn \times a,\ f_3 = a + cn \times b,\ f_4 = a + cn,$$

and that "a" and "b" are algebraic variables.

R, I	cn in f_1	cn in f_2	cn in f_3	cn in f_4
0 +1	i	i	+ i	+ i
0 -1	- i	- i	- i	- i
0 +5	i × 5	i × 5	+ i × 5	+ i × 5
0 -5	- i × 5	- i × 5	- i × 5	- i × 5
+1 +1	1 + i	(1 + i)	+ (1 + i)	+ 1 + i
-1 +1	- 1 + i	(i - 1)	+ (i - 1)	- 1 + i
+1 -1	1 - i	(1 - i)	+ (1 - i)	+ 1 - i
+1 +5	1 + i × 5	(1 + i × 5)	+ (1 + i × 5)	+ 1 + i × 5
+1 -5	1 - i × 5	(1 - i × 5)	+ (1 - i × 5)	+ 1 - i × 5
-1 -5	- 1 - i × 5	- (1 + i × 5)	- (1 + i × 5)	- 1 - i × 5
0 +.1	i/10	i/10	+ i/10	+ i/10
0 +.4	i × 2/5	i × 2/5	+ i × 2/5	+ i × 2/5
+.1 +.1	1/10 + i/10	(1/10 + i/10)	+ (1/10 + i/10)	+ 1/10 + i/10
+.1 +.4	1/10 + i × 2/5	(1/10 + i × 2/5)	+(1/10 + i × 2/5)	+1/10 + i × 2/5
with sign	*false*	*false*	*true*	*true*
with br1	*false*	*true*	*true*	*false*

table 3.2 The form of the output of a complex number

The actual value of the two Boolean parameters *with sign* and *with br1*, which determine the desired form of the output, are given in the last two rows of table 3.2.
Another possibility is that "cn" has the form of a product or quotient as e.g. "i × 5" or "i × 2/5" and "cn" is the rhs or lhs of a power, as e.g. in: "cn ↑ a" or"a ↑ cn".
In these cases brackets have to be output also; therefore, the third Boolean parameter, *with br2*, is introduced.

The output of an arbitrary number is produced by the procedure *OUTPUT N*.

3.3.2.2.5. The output procedures

We now give a reproduction of the output procedures.

```
procedure PR real num(r,with sign,with br1,with br2);
   value r,with sign,with br1,with br2; real r;
   Boolean with sign,with br1,with br2;
begin integer i,n,m; real A0,A1,A2,B0,B1,B2,r1,R; Boolean neg sign,aux;
   neg sign:= r < 0; if neg sign then r:= - r;
   A0:= B1:= 1; B0:= 0; R:= r; n:= 0;
   if R ≥ 2↑12 then goto FLOAT; A1:= entier(R+.5);
   R:= R - A1; if abs(R) ≤ null then
   begin A2:= A1; B2:= 1; goto RATIONAL end;
AGAIN: R:= 1/R; r1:= entier(R+.5); R:= R - r1;
   A2:= r1 × A1 + A0; B2:= r1 × B1 + B0;
   if abs(A2) ≥ 2↑12 ∨ abs(B2) ≥ 2↑12 then goto FLOAT;
   if ¬ (abs(A2/B2 - r) ≤ rel acc × r) then
   begin n:= n + 1; A0:= A1; A1:= A2; B0:= B1; B1:= B2;
      goto if n > 20 then FLOAT else AGAIN
   end;
RATIONAL: aux:= with br2 ∨ (¬ with sign ∧ with br1 ∧ neg sign);
   if aux then PR string(‹(›) else if with sign ∧ ¬ neg sign
   then PR string(‹+›); if neg sign then PR string(‹-›);
   PR int num(abs(A2)); if B2 ≠ 1 then
   begin PR string(‹/›); PR int num(abs(B2)) end;
   if aux then PR string(‹)›); goto OUT;
FLOAT: aux:= ¬ with sign ∧ with br1 ∧ neg sign;
   if aux then PR string(‹(›) else if with sign ∧ ¬ neg sign then
   PR string(‹+›); if neg sign then PR string(‹-›); PR string(‹.›);
   n:= entier(ln(r)×.43429 4482 + 1); r:= r/10↑(n-1); r:= r × (1+⏨-12);
   if r < 1 then begin n:= n-1; r:= r × 10 end;
   if r ≥ 10 then begin n:= n+1; r:= r/10 end; i:= 0;
```

```
A: i:= i+1; m:= entier(r); r:= (r - m) × 10; A1:= if i = 1 then m × ⏨-10
   else A1 × 10; PR sym(m); if ¬(i = 12 ∨ abs(r) ≤ A1) then goto A;
   PR string(‘⏨’); PR int num(n); if aux then PR string(‘)’);
OUT: end PR real num;

procedure PR comp num(R,I,with sign,with br1,with br2);
   value R,I,with sign,with br1,with br2; real R,I;
   Boolean with sign,with br1,with br2;
begin integer n,m; Boolean aux,neg sign;
   if abs(R) ≤ null then
   begin neg sign:= I < 0; if neg sign then I:= - I;
      aux:= with br2 ∨ (¬ with sign ∧ with br1 ∧ neg sign);
      if aux then PR string(‘(’) else if with sign ∧ ¬ neg sign
      then PR string(‘+’); if neg sign then PR string(‘-’);
      PR string(‘i’); if abs(I - 1) > rel acc then
      begin FIX; TYPE(RN(1/I),n,m); ERASE; if n = integer then
         begin PR string(‘/’); PR int num(1/I) end else
         begin PR string(‘×’); PR real num(I,false,false,false) end
      end; if aux then PR string(‘)’)
   end else
   begin neg sign:= R < 0 ∧ I < 0; aux:= with br1 ∨ with br2;
      if with sign then
      begin if neg sign ∧ aux then
         begin R:= -R; I:= -I; PR string(‘-’) end
         else if ¬ neg sign then PR string(‘+’)
      end; if aux then PR string(‘(’);
      if R < 0 ∧ I > 0 then
      begin PR comp num(0,I,false,false,false);
         PR real num(R,true,false,false)
      end else
      begin PR real num(R,false,false,false);
         PR comp num(0,I,true,false,false)
      end; if aux then PR string(‘)’)
end end PR comp num;
```

```
procedure OUTPUT N(f,with sign,with br1,with br2);
  value f,with sign,with br1,with br2; integer f;
  Boolean with sign,with br1,with br2;
begin integer t,a,b; real R,I; t:= TYPE(f,a,b);
  if a = integer then
  begin VAL OF INT NUM(f,b); if with br1 ∧ b < 0 then
    PR string(⊰(⊱); if with sign ∧ b ≥ 0 then PR string(⊰+⊱);
    PR int num(b); if with br1 ∧ b < 0 then PR string(⊰)⊱)
  end else
  if a = real then
  begin VAL OF REAL NUM(f,R); PR real num(R,with sign,with br1,
    with br2)
  end else
  if a = complex then
  begin VAL OF COMP NUM(f,R,I);
    PR comp num(R,I,with sign,with br1,with br2)
  end
end OUTPUT N;

procedure OUTPUT RC(f,case,type); value f,case,type;
  integer f,case,type;
begin integer t,a,b;
  procedure pr(r,c); string r,c; if case ≠ 3 then PR string(r)
    else PR string(c);
  procedure pr1(c); string c; pr(⊰⊱,c);
  procedure OP(f,t1); value f,t1; integer f,t1;
  if case = 3 then OUTPUT C(f) else OUTPUT RC(f,case,t1);
  procedure LBR; if case = 3 ∨ t < type then PR string(⊰(⊱);
  procedure RBR; if case = 3 ∨ t < type then PR string(⊰)⊱);

  t:= TYPE(f,a,b);
  if t = sum then
  begin LBR; if case = 2 then OUTPUT R(f) else
    begin OP(a,sum); PR string(⊰+⊱); OP(b,sum) end; RBR
  end else
```

```
if t = difference then
begin pr1(⊀D⊁); LBR; OP(a,sum); pr(⊀-⊁,⊀,⊁); OP(b,product);
  RBR
end else

if t = product then
begin LBR; if case = 2 then OUTPUT R(f) else
  begin OP(a,product); PR string(⊀×⊁); OP(b,product) end; RBR
end else

if t = quotient then
begin pr1(⊀Q⊁); LBR; OP(a,product); pr(⊀/⊁,⊀,⊁); OP(b,power);
  RBR
end else

if t = integral power then
begin pr1(⊀INT POW⊁); t:= power; LBR; OP(b,power); pr(⊀↑⊁,⊀,⊁);
  PR int num(a); RBR
end else

if t = power then
begin pr1(⊀POWER⊁); LBR; OP(a,power); pr(⊀↑⊁,⊀,⊁);
  OP(b,power + 1); RBR
end else

if t = function then
begin if a = expf then pr(⊀exp⊁,⊀EXP⊁) else if a = lnf then
  pr(⊀ln⊁,⊀LN⊁) else if a = sinf then pr(⊀sin⊁,⊀SIN⊁) else
  if a = cosf then pr(⊀cos⊁,⊀COS⊁) else if a = arctanf then
  pr(⊀arctan⊁,⊀ARCTAN⊁) else pr(⊀sqrt⊁,⊀SQRT⊁);
  PR string(⊀(⊁); OP(b,0); PR string(⊀)⊁)
end else
```

```
if t = number then
begin if case = 3 then
  begin real R,I; if f = im unit then PR string(⟨iu⟩) else
    begin VAL OF NUM(f,R,I); PR string(⟨CN(⟩);
      PR real num(R,false,false,false); PR string(⟨,⟩);
      PR real num(I,false,false,false); PR string(⟨)⟩)
  end end else OUTPUT N(f,false,type > 0,type > quotient)
end else

if t = tr pow series then
begin integer i,x; integer array coeff[0:a]; COEFFICIENT(f,a,x,
  coeff); if case = 3 then
  begin Boolean exp; exp:= expand; expand:= false; FIX;
    f:= Sum(i,0,a,P(coeff[i],INT POW(x,i))); OUTPUT C(f); ERASE;
    expand:= exp
  end else
  begin LBR; OP(coeff[0],0); for i:= 1 step 1 until a do
    begin PR string(⟨+⟩); OP(coeff[i],if case = 1 then product
      else quotient); PR string(⟨×⟩); OP(x,0); if i > 1 then
      begin PR string(⟨↑⟩); PR int num(i) end
    end; PR string(⟨+O(⟩); OP(x,0); PR string(⟨↑⟩);
    PR int num(a+1); PR string(⟨)⟩); RBR
end end else OUTPUT VARIABLE(f)
end OUTPUT RC;

procedure OUTPUT R(f); value f; integer f;
if ¬ expand then OUTPUT RC(f,1,0) else
begin integer t,a,b; t:= TYPE(f,a,b);
  if ¬ (t = sum ∨ t = product) then OUTPUT RC(f,2,0) else
  begin integer i,j; integer array A[0:1],B[1:1,1:1];
    CONVERT(f,B,A,true);
    begin integer array F[1:A[0],0:A[1]],LF[0:A[0]];
      CONVERT(f,F,LF,false); SIMPL2REPR(F,LF);
      if LF[0] = 0 then PR string(⟨0⟩) else
      for i:= 1 step 1 until LF[0] do
```

```
        begin if ¬(F[i,0] = one ∨ F[i,0] = minone) then
          begin real R,I; VAL OF NUM(F[i,0],R,I);
            OUTPUT N(F[i,0],i > 1,LF[i] > 0 ∧ abs(R) > null ∧ abs(I) >
              null,false);
          end; if LF[i] = 0 then
          begin if F[i,0] = one then
            begin if LF[0] = 1 then PR string(⟨1⟩) else PR string(⟨+1⟩)
            end else
            if F[i,0] = minone then PR string(⟨-1⟩)
          end else
          begin if F[i,0] = one then
            begin if i ≠ 1 then PR string(⟨+⟩) end else
            if F[i,0] = minone then PR string(⟨-⟩) else
            PR string(⟨×⟩)
          end;
          for j:= 1 step 1 until LF[i] do
          begin FIX; OUTPUT RC(INT POW(T(F,i,j,1),
            T(F,i,j,2)),2,product); ERASE;
            if j < LF[i] then PR string(⟨×⟩)
end end end end end OUTPUT R;

procedure OUTPUT VARIABLE(f); value f; integer f;
begin integer p,a,sym,i,j; TYPE(f,j,i); a:= 0;
  for j:= 0 step 1 until 7 do
  begin p:= 64↑(3-j+j:4×4); sym:= (identifier list[i,j:4] - a): p;
    a:= if j = 3 then 0 else a + sym × p;
    if sym = 0 then goto OUT else PR sym(sym - 1)
  end;
OUT: end OUTPUT VARIABLE;

procedure COUNT(f,type,array,bounds only);
value f,type; integer f,type; Boolean bounds only;
integer array array;
begin integer t,a,b; t:= TYPE(f,a,b);
```

```
  if t = type then
  begin COUNT(a,type,array,bounds only);
    COUNT(b,type,array,bounds only)
  end else
  begin array[0]:= array[0] + 1;
    if ¬ bounds only then array[array[0]]:= f
  end
end COUNT;

Boolean procedure REAL(f); value f; integer f;
begin integer t,a,b; t:= TYPE(f,a,b);
  if t = sum ∨ t = difference ∨ t = product ∨
    t = quotient then
  begin if REAL(a) then REAL:= REAL(b) else REAL:= false
  end else
  if t = integral power ∨ (t = function ∧
    (a = expf ∨ a = sinf ∨ a = cosf))
  then REAL:= REAL(b) else
  if t = number ∨ t = algebraic variable then
    REAL:= a ≠ complex else
  if t = tr pow series then
  begin integer i,x; Boolean B; integer array c[0:a];
    COEFFICIENT(f,a,x,c); B:= REAL(x); i:= - 1;
    for i:= i + 1 while B ∧ i ≤ a do B:= B ∧ REAL(c[i]);
    REAL:= B
  end else REAL:= false
end REAL;

procedure OUTPUT C(f); value f; integer f;
begin integer t,a,b; t:= TYPE(f,a,b);
  if t = sum ∨ t = product then
  begin integer i,j,k; integer array A[0:0];
    A[0]:= 0; COUNT(f,t,A,true);
```

```
    begin integer array B[0:A[0]]; Boolean array BOOL[1:A[0]];
      B[0]:= 0; COUNT(f,t,B,false); j:= k:= 0;
      for i:= 1 step 1 until B[0] do
      begin BOOL[i]:= REAL(B[i]); if BOOL[i] then j:= j + 1 end;
      if j < B[0] ∧ j > 0 then
      begin if t = sum then PR string(⊰SR(⊱) else PR string(⊰PR(⊱) end;
      if j > 0 then
      begin if j = B[0] then PR string(⊰RN(⊱); for i:= 1 step 1 until B[0] do
        begin if BOOL[i] then
          begin OUTPUT RC(B[i],1,t); k:= k + 1; if k < j then
            begin if t = sum then PR string(⊰+⊱) else PR string(⊰×⊱) end
          end
        end; if j = B[0] then PR string(⊰)⊱)
      end; if j < B[0] ∧ j > 0 then PR string(⊰,⊱);
      if j < B[0] then
      begin k:= 0; for i:= 1 step 1 until B[0] do
        begin if ¬ BOOL[i] then
          begin k:= k + 1; if B[0] - j - k ≥ 1 then
            begin if t = sum then PR string(⊰S(⊱) else PR string(⊰P(⊱) end;
            OUTPUT C(B[i]); if B[0] - j - k ≥ 1 then PR string(⊰,⊱)
          end
        end; for i:= 2 step 1 until k do PR string(⊰)⊱)
      end; if j < B[0] ∧ j > 0 then PR string(⊰)⊱)
    end
  end else
  if REAL(f) then
  begin PR string(⊰RN(⊱); OUTPUT RC(f,1,0); PR string(⊰)⊱) end
  else OUTPUT RC(f,3,0)
end OUTPUT C;
```

3.3.2.3. The derivative procedure

In this section we reproduce the procedure *DER*, which calculates the derivative of a formula.
Apart from the statements concerning the special derivatives, this procedure is identical to the procedure given in 2.12.

```
integer procedure DER(f,x); value f,x; integer f,x;
begin integer t,a,b; t:= TYPE(f,a,b);
  if x = spec der[0,1] then
  begin integer i; for i:= 1 step 1 until numb of spec der do
    begin if f = spec der[i,1] then
      begin DER:= spec der[i,2]; goto END end
  end end;
  if f = x then DER:= one else
  if t = sum then DER:= S(DER(a,x),DER(b,x)) else
  if t = difference then DER:= D(DER(a,x),DER(b,x)) else
  if t = product then DER:= S(P(DER(a,x),b),P(a,DER(b,x))) else
  if t = quotient then
  begin integer da,db; da:= DER(a,x); db:= DER(b,x);
    DER:= if db = zero then Q(da,b) else Q(D(P(da,b),P(a,db)),P(b,b))
  end else
  if t = power then DER:= P(f,DER(P(b,LN(a)),x)) else
  if t = integral power then
  begin integer d; d:= DER(b,x); DER:= if d = zero then zero else
    P(IN(a),P(INT POW(b,a-1),d))
  end else
  if t = tr pow series then
  begin integer i,y; integer array coeff[0:a]; COEFFICIENT(f,a,y,coeff);
    DER:= S(TPS(i,a,y,DER(coeff[i],x)), if x ≠ y ∨ a = 0 then zero else
      TPS(i,a-1,y,P(IN(i+1),coeff[i+1])))
  end else
```

```
if t = function then
begin integer d; d:= DER(b,x);
  DER:= if d = zero then d else
    if a = expf then P(f,d) else
    if a = lnf then P(INT POW(b,-1),d) else
    if a = sinf then P(COS(b),d) else
    if a = cosf then P(min one,P(SIN(b),d)) else
    if a = arctanf then P(Q(one,S(one,P(b,b))),d) else
    P(Q(RN(.5),f),d)
end else DER:= zero;
END: end DER;
```

3.3.3. The third part of the processor

The third and final part of the processor is dedicated to the reading and execution of the formula program. It starts with a search for the closing bracket ")" of the heading of the formula program to be executed. It then consists of labelled series of statements. Except for *NEXT* and *Assign*, all these labels occur in the switch list of the switch *CASE* declared in section 3.3.2.
These switch-list elements correspond to the formula-statement identifiers of the preparatory information of the input tape (see section 3.3.1).

The statements after *NEXT* form the central part; they read an identifier from input tape. If this identifier is a formula-statement identifier, then the processor jumps to the corresponding label of the switch list of *CASE*; if this identifier is not a formula-statement identifier, then it is necessarily a formula identifier, to which a formula will be assigned through an assignment statement, and the processor jumps to the label *Assign*.

We now give a description of the statements after the labels *Real* up to *Solve linear equations*.

Real: identifiers are read from input tape; they are associated with algebraic variables of real type.

Spec der: the special derivatives are read from input tape. Due to the fact that *numb of spec der* is augmented afterwards, it is possible to have e.g. the special-derivative statement:

"SPEC DER(x, y, DER(sin(x)))"

which would not be possible if *numb of spec der* were augmented before the reading of "y" and "DER(sin(x))".

Output c: An output statement for the complex form of the output is executed.

Output r: An output statement for the real form of the output is executed; depending on the values of *expand*, the optimal form or the ordinary form for the output is produced.

FIX: A call of *FIX* is invoked.

Erase: Formula identifiers, corresponding to formulae which are erased by a call of *ERASE* lose their meaning; if the procedure *primary* comes across such an identifier, it will treat it as an algebraic variable.

Erase but retain: The identifiers of formulae to be retained are read and stored into the array *g*. The "erase but retain process" of section 2.7 is used by means of a call of *STORE TO DRUM*, in order to erase some formulae and retain others.
As actual parameter of *STORE TO DRUM*, the procedure *RET VAR* is substituted. The tasks of *RET VAR* are briefly described:
Let "a" be an algebraic variable which occurs in a formula to be retained. Let "a^*" be the index of the location at which "a" is stored in the general formula-manipulation-system. Let "a^+" be the index in *identifier list* at which the identifier "a" is stored. Then *identifier list*[a^+,-1] = a^* and the rhs quantity of the formula "a" is equal to "a^+". After the "erase but retain process", the locations given by "a^*", where "a" is stored in the general formula-manipulation-system, have, in general, been changed; hence, *identifier list*[a,-1] has to be changed too. This is one task of *RET VAR*.
It is possible that "a" occurs at several places in the formulae to be retained. However, only the first time that the "erase but retain process" encounters "a", does "a^*" have to be changed; the second, third, etc., occurrences of "a" should not lead to repeated storage of the algebraic variable "a".
Therefore, *RET VAR* maintains a list of algebraic variables, by means of the array *id*, with pointer *n*. After the execution of the "erase but retain process", this list contains all the different algebraic variables "a" whose locations "a^*", in the general-formula-manipulation system, have been changed.
By means of this list, it is possible to decide which identifiers should lose their meaning.

Nlcr: outputs an nlcr symbol.

Pr string: a string of symbols is output. The occurrence of two successive symbols ")" has as effect the output of one symbol ")". The last symbol ")" of an uneven number of these symbols indicates the end of the string.

Expand and *Not expand*: The processor is put into the expanding and the not expanding state,respectively.

Tps coeff: Identifiers, read from input tape, are associated with the coefficients of a truncated power-series.

Solve linear equations: By means of an elimination process n (given by the absolute value of the first-read integer s) equations (their left-hand sides given by $f[i]$, $i = 1, \ldots, n$) are solved for n unknowns (given by $xt[i]$, $i = 1, \ldots, n$).

End: The execution of the formula program is discontinued by a call of *ERROR(true, ⊰ready⊱)*.

```
   comment begin of the body of the formula program processor;
   numb of spec der:= 0; spec der[0,1]:= -1000;
O: s:= RE sym; if s = underlining then s:= RE sym else
   if s ≠ right br then goto O;
   ERROR(s ≠ right br,⊰heading not closed with )⊱); next symbol:= semicolon;
   line counter:= 0;
NEXT: ERROR(next symbol ≠ semicolon,⊰statement not closed with;⊱);
   NS; s:= IDENTIFIER; if s > numb of spec id then goto Assign
   else goto CASE[s];

Assign: ERROR(next symbol ≠ colon,⊰wrong assignment st⊱); NS;
   ERROR(next symbol ≠ equal,⊰wrong assignment st⊱); NS;
   identifier list[s,-1]:= formula; goto NEXT;

Real: NS; s:= IDENTIFIER; identifier list[s,-1]:= STORE(real,algebraic
   variable,s); if next symbol ≠ right br then goto Real;
   NS; goto NEXT;
```

```
Spec der: NS; s:= primary; spec der[0,1]:= s; numb of spec der:= 0;
  again: if next symbol = right br then begin NS; goto NEXT end;
    NS; spec der[numb of spec der+1,1]:= primary; NS;
    spec der[numb of spec der+1,2]:= formula;
    numb of spec der:= numb of spec der + 1; goto again;

Output c:
  begin Boolean b; b:= false; s:= RE sym;
    if s = question mark then b:= true else
    begin PR nlcr; PR string(‹ASSIGN(›); PR sym(s) end;
    for s:= RE sym while s ≠ colon do PR sym(s); NS; NS;
    if ¬ b then PR string(‹,›); OUTPUT C(formula);
    if ¬ b then PR string(‹);›); NS; goto NEXT
  end;

Output r:
  begin Boolean b; b:= false; s:= RE sym;
    if s = question mark then b:= true else
    begin PR nlcr; PR sym(s) end;
    for s:= RE sym while s ≠ colon do PR sym(s); NS; NS;
    if ¬ b then PR string(‹:= ›); OUTPUT R(formula);
    if ¬ b then PR string(‹;›); NS; goto NEXT
  end;

Fix: FIX; goto NEXT;

Erase: for s:= numb of spec id + 1 step 1 until idp do
  begin if ¬ FIXED(identifier list[s,-1]) then identifier list[s,-1]:= -1
  end; ERASE; goto NEXT;
```

```
Erase but retain:
  begin integer k,n,i,p; integer array g,id[1:idp - numb of spec id];
    Boolean procedure RET VAR(f); integer f;
    begin k:= di; if FIXED(f) then RET VAR:= true else
      begin integer i,a,b; RET VAR:= false;
        if TYPE(f,a,b) = algebraic variable then
        begin n:=n + 1; id[n]:=b; identifier list[b,-1]:=k + 1 end
    end end RET VAR;
    n:= i:= 0;
  A: NS; i:= i + 1; g[i]:= IDENTIFIER; if next symbol ≠ right br then
    goto A; NS; s:= i;
    STORE TO DRUM(i,1,s,identifier list[g[i],-1],RET VAR);
    for i:= 1 step 1 until s do identifier list[g[i],-1]:=
      -identifier list[g[i],-1] - 2;
    for i:= 1 step 1 until n do
    begin if identifier list[id[i],-1] ≥ 0 then
      identifier list[id[i],-1]:= - identifier list[id[i],-1] - 2
    end; for i:= numb of spec id + 1 step 1 until idp do
    begin p:= identifier list[i, -1]; s:= if p < -1 then -p - 2 else p;
      identifier list[i, -1]:= if FIXED(s) ∨ p < -1 then s else -1
end end; LOWER INDEX; goto NEXT;

Nlcr: PR nlcr; goto NEXT;

Pr string: s:= RE sym; if s = right br then
begin integer s1; s1:= RE sym; if s1 = right br then
  begin PR sym(s); goto Pr string end else
  begin next symbol:= semicolon; goto NEXT
end end else begin PR sym(s); goto Pr string end;

Expand: expand:= true; goto NEXT;
```

```
Not expand: expand:= false; goto NEXT;

Tps coeff: NS; s:= formula;
  begin integer t,i,x,n; t:= TYPE(s,n,i);
    if t ≠ tr pow series then
    begin NS; n:= IDENTIFIER; identifier list[n,-1]:= s;
    B: if next symbol = right br then begin NS; goto NEXT end;
      NS; n:= IDENTIFIER; identifier list[n,-1]:= zero; goto B
    end else
    begin integer array co,coeff[0:n]; i:= -1;
    A: NS; i:= i + 1; ERROR(i > n,⊰degree of tr pow series too small⊱);
      co[i]:= IDENTIFIER; if next symbol ≠ right br then goto A;
      COEFFICIENT(s,i,x,coeff);
      for n:= 0 step 1 until i do identifier list[co[n],-1]:= coeff[n];
      NS; goto NEXT
  end end;

Solve linear equations: s:= read;
  begin integer i,j,d,n,k; integer array xt,x,y,p,f[1:abs(s)];
    n:= abs(s); for i:= 1 step 1 until n do
    begin p[i]:= i; NS; xt[i]:= y[i]:= formula; TYPE(xt[i],d,x[i]) end;
    for i:= 1 step 1 until n do begin NS; f[i]:= formula end;
    for i:= 1 step 1 until n do FIX; i:= 0; if n = 0 then goto out;
  A: i:= i + 1; k:= 0;
  B: FIX; d:= DER(f[i],xt[p[i]]); if d = zero then
    begin ERASE; k:= k + 1; ERROR(i+k > n,⊰singular system⊱);
      d:= p[i]; p[i]:= p[i + k]; p[i + k]:= d; goto B
    end;
    LOWER INDEX;
```

```
    xt[p[i]]:= SIMPLIFY(P(Q(minone,d),SUBSTITUTE(f[i],j,1,1,xt[p[i]],zero)));
    if n > 1 ∧ i = n ∧ s < 0 then ERASE BUT RETAIN(k,1,1,xt[p[i]]);
    for k:= 1 step 1 until i do
    begin if i = n then FIX; if k < i ∧ ¬(i = n ∧ s < 0) then
      xt[p[k]]:= SIMPLIFY(SUBSTITUTE(xt[p[k]],j,1,1,y[p[i]],xt[p[i]]));
      if i = n then
      begin PR nlcr; FIX; OUTPUT R(STORE(real,algebraic variable,
        x[p[k]])); ERASE; PR string(⟨:= ⟩);
        OUTPUT R(if s > 0 ∨ k = n then xt[p[k]] else
          SUBSTITUTE(xt[p[k]],j,1,1,y[p[i]],xt[p[i]])); PR string(⟨;⟩);
        if s < 0 then begin ERASE; identifier list[x[p[k]],-1]:= -1 end
        else LOWER INDEX
    end end; if i ≠ n ∨ s > 0 then
    begin ERASE BUT RETAIN(j,1,i,xt[p[j]]); for k:= 1 step 1 until i do
      identifier list[x[p[k]],-1]:= xt[p[k]]
    end else LOWER INDEX; if i < n then
    begin if i+1 = n ∧ s < 0 then FIX; f[i+1]:= SUBSTITUTE(f[i+1],
      j,1,i,y[p[j]],xt[p[j]]); goto A
    end;
  out: end; NS; goto NEXT;
End: ERROR(true,⟨ready⟩)
end; comment the next two ends correspond to the two begins of the
general system;
end end
end end          +-×/=∧,:; ()      i. ₁₀?
REAL,SPEC DER,OUTPUT C,OUTPUT R,FIX,ERASE,ER B RET,NLCR,
PR STRING,EXPAND,NOT EXP,COEFF,SOL LIN EQ,END+
TPS,exp,ln,sin,cos,arctan,sqrt,DER,SIMPL,CC,SUBST,QUOT,COMM DIV;
```

Appendix

The headings of the Mathematical Centre standard-procedures as used in the ALGOL 60 programs are given below along with a short description. More information is given in [8].

Input procedures:

real procedure read; reads a number from input tape and becomes equal to this number.
Text between two apostrophes is skipped.
integer procedure RESYM; reads a symbol from input tape and becomes equal to the internal representation of this symbol;

Output procedures for printing:

procedure PRSYM(n); value n; integer n;
If the actual value of n corresponds to the internal representation of a symbol which may occur on input tape, then this symbol is printed after a call of this procedure.

procedure NLCR; a call of this procedure has the effect that a new-line-carriage-return command is given to the printer.

procedure ABSFIXT(n,m,x); value n, m, x; integer n, m; real x;
a call of this procedure has as effect the printing of the absolute value of the actual parameter x with n decimals before and m decimals after the decimal point.

procedure FIXT(n,m,x); value n, m, x; integer n, m; real x;
a call of this procedure has as effect the printing of the value of the actual parameter x (including the sign symbol) with n decimals before and m decimals after the decimal point.

procedure FLOT(n,m,x); value n, m, x; integer n, m; real x;
a call of this procedure has as effect the printing of the value of the actual parameter x in floating-point notation (including the sign symbol) with n decimals for the mantissa and m decimals for the exponent of 10.

procedure *PRINTTEXT(s); string s;*
a call of this procedure has as effect the printing of the string given by the actual parameter *s*, without the string quotes ⊀ and ⊁.

procedure *SPACE(n); value n; integer n;*
a call of this procedure has as effect the printing of *n* space symbols.

Output procedures for punching:

The headings of these procedures are almost the same as the headings of the procedures for printing; the procedure identifiers are changed into *PUSYM*, *PUNLCR*, *ABSFIXP*, *FIXP*, *FLOP*, *PUTEXT* and *PUSPACE*.

procedure *RUNOUT;* a call of this procedure has as effect the punching of a piece of blank tape.

procedure *STOPCODE;* a call of this procedure has as effect the punching of a stopcode symbol and a call of *RUNOUT*.

Other procedures

procedure *EXIT;* a call of this procedure has as effect that the execution of the program is discontinued.

integer procedure *EVEN(n); value n; integer n;*
may be described as: $EVEN := (-1) \uparrow n$.

real procedure *SUM(i,a,b,xi); value b; integer i, a, b; real xi;*
may be described as:

$$SUM := \sum_{i=a}^{b} xi$$

The actual value of *xi* will in general depend on the current value of *i*.

procedure *TO DRUM(A,p); value p; array A; integer p;*
A call of this procedure has as effect that the array elements of *A* are stored on the drum; the first array element is stored in a location defined by the actual value of *p*.

<u>procedure</u> FROM DRUM(A,p); <u>value</u> p; <u>array</u> A; <u>integer</u> p;
A call of this procedure has as effect that a set of data occurring on the drum and with begin address defined by *p* are stored into the array elements of *A*.

<u>real</u> <u>procedure</u> time;
This procedure delivers the time, measured in seconds with an accuracy of .01 sec, during which an ALGOL 60 program was under execution.

References

[1] M. Abramowitz, I.A. Stegun (editors), Handbook of Mathematical Functions, U.S. Department of Commerce, National Bureau of Standards, Applied Mathematics Series 55, Third printing, March 1965.

[2] S.J. Bijlsma, Algebraic Operations in ALGOL 60. The Saddlepoint method, Report TN 45, Mathematical Centre, Amsterdam, May 1966.

[3] E. Bond, M. Auslander, S. Grisoff, R. Kenney, M. Myszewski, J.E. Sammet, R.G. Tobey, S. Zilles, FORMAC - An experimental formula manipulation compiler, Proceedings of the ACM National Conference, August 1964.

[4] W.S. Brown, The ALTRAN language and the ALPAK system for symbolic algebra on a digital computer, Presented at the Princeton University Conference in Computer Sciences, August 8 - September 2, 1966.

[5] R. Courant, D. Hilbert, Methods of Mathematical Physics, volume II, Interscience 1962.

[6] A. Gibbons, A program for the automatic integration of differential equations, using the method of Taylor series, The Computer Journal, 3 (1960), pp. 108-111.

[7] F.E.J. Kruseman Aretz, ALGOL 60 translation for everybody, Electronische Datenverarbeitung Heft 6/1964, pp. 233-244.

[8] F.E.J. Kruseman Aretz, Het MC-ALGOL 60-systeem voor de X8. Voorlopige programmeursopleiding. Report MR 81, Mathematical Centre, Amsterdam, June 1966.

[9] J. McCarthy, P. Abrahams, D. Edwards, T. Hart, M. Levin, LISP 1.5 Programmer's Manual,
Computation Centre and Research Laboratory of Electronics, MIT, Cambridge Mass.

[10] R.E. Moore, The automatic analysis and control of error in digital computation based on the use of interval numbers,
published in Error in Digital Computation, vol. 1, edited by Louis B. Rall,
John Wiley & Sons, Inc., New York 1964.

[11] P. Naur (editor), Revised report on the algorithmic language ALGOL 60,
Regnecentralen, Copenhagen, 1962.

[12] A.J. Perlis, R. Itturiaga, T.A. Standish, A Definition of Formula ALGOL,
Carnegie Institute of Technology, Pittsburgh, P.A., March 1966.

[13] R.D. Richtmyer, Power series solution, by machine, of a nonlinear problem in two-dimensional fluid flow,
Annals of the New York Academy of Sciences, 86 (1960), pp. 828-843.

[14] R.P. van de Riet, Algebraic Operations in ALGOL 60. A second order problem.
Report TW 96, Mathematical Centre, Amsterdam, March 1965.

[15] R.P. van de Riet, Algebraic Operations in ALGOL 60. The Cauchy Problem I,
Report TW 97, Mathematical Centre, Amsterdam, December 1965.

[16] R.P. van de Riet, An application of a method for algebraic manipulation in ALGOL 60,
Report TW 99, Mathematical Centre, Amsterdam, January 1966.

[17] R.P. van de Riet, Formula manipulation in ALGOL 60 (preliminary report),
Report TW 101, Mathematical Centre, Amsterdam, August 1966.

[18] R.P. van de Riet, Complex Arithmetic in ALGOL 60,
Report TW 105, Mathematical Centre, Amsterdam, March 1967.

[19] R.P. van de Riet, Algorithm 186 Complex Arithmetic,
Communications of the ACM, 6, 7, July 1963.

[20] J.E. Sammet, Formula Manipulation by Computer,
Advances in Computers, Volume 8, McGraw Hill,
Preliminary version as Technical Report TROO.1363, IBM
Systems Development Division,
Poughkeepsie, N.Y., November 1965.

[21] J.E. Sammet, Survey of the Use of Computers for Doing Non-Numerical Mathematics,
Technical Report TROO.1428,
IBM Systems Development Division,
Poughkeepsie, N.Y., March 1966.

[22] J.E. Sammet, An Annotated Descriptor Based Bibliography on the Use of Computers for Non-Numerical Mathematics,
Computing Reviews 7, 4, July-August 1966.

[23] J.C. Schoenfeld, Complex Arithmetics in ALGOL,
RIJSKWATERSTAAT, Directie Waterhuishouding en Waterbeweging,
Mathematisch-Fysische Afdeling,
Boorlaan 14, Den Haag, Netherlands, 1965.

[24] R.L. Smith, Algorithm 116 Complex Division,
Communications of the ACM, 5 (1962).

[25] C.R. Traas, Blunt body supersonic flow. Investigations about an inverse and a direct method,
report G 63, National Aerospace Laboratory, Amsterdam (1967).

[26] B.L. van der Waerden, Algebra, Vierte Auflage,
Springer-Verlag, Berlin-Göttingen-Heidelberg (1955).

[27] P. Wynn, An Arsenal of ALGOL procedures for Complex Arithmetic,
BIT, 2(1962), pp. 232-255.

[28] R.P. van de Riet, Garbage-collection methods for ABC in ALGOL 60,
TW report 110, Mathematical Centre, Amsterdam 1969.

[29] W.P. de Roever, Infinite-precision rational-function procedures
for ABC ALGOL, to be published.

[30] G. ten Velden, Simplification-procedures for ABC ALGOL, Report
MR 112, Mathematical Centre, Amsterdam 1970.