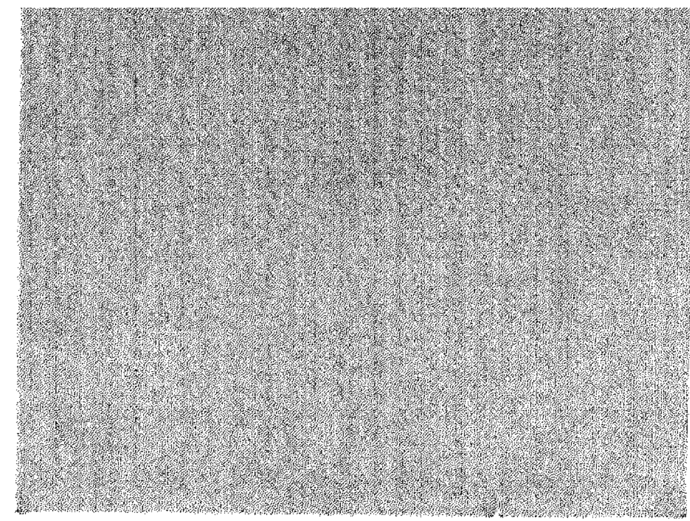
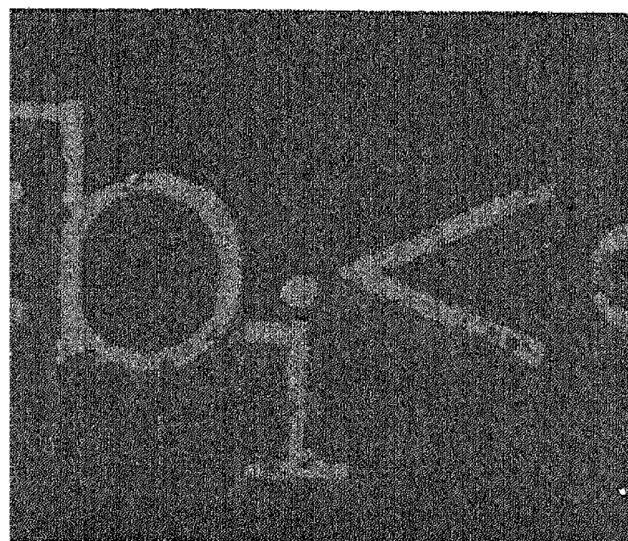
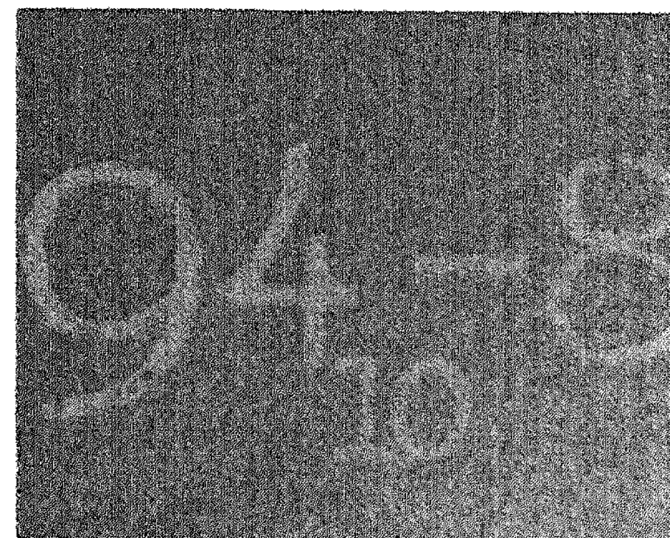
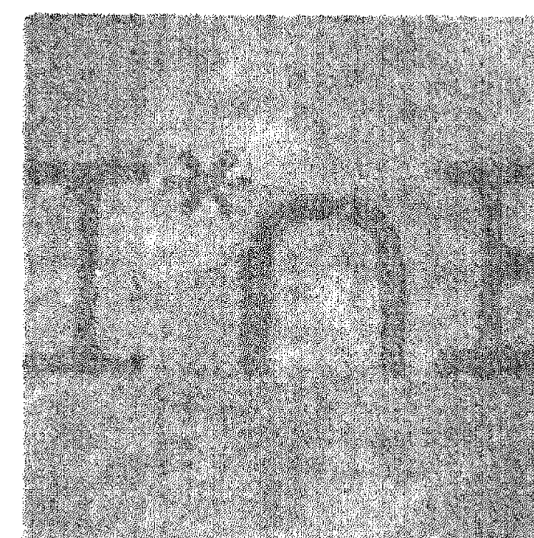
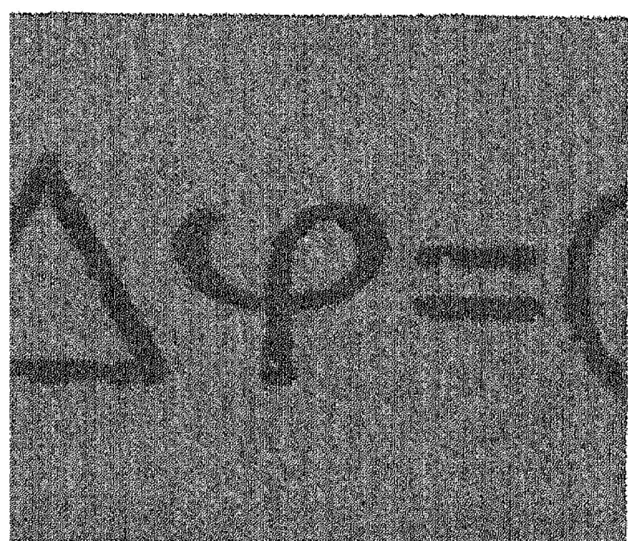
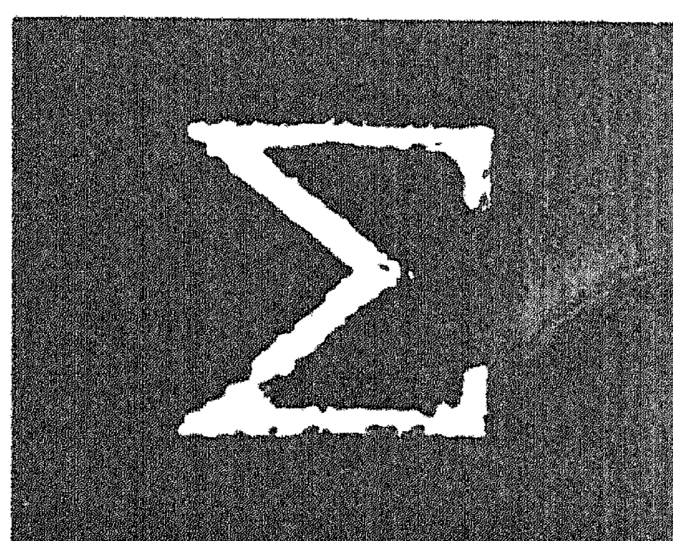
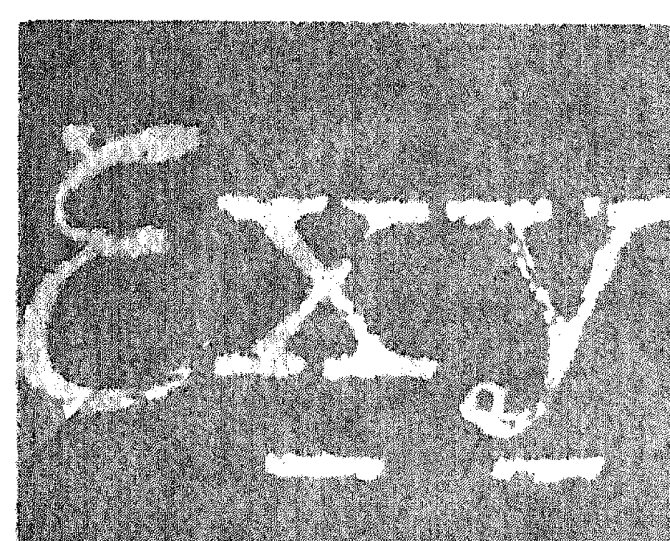
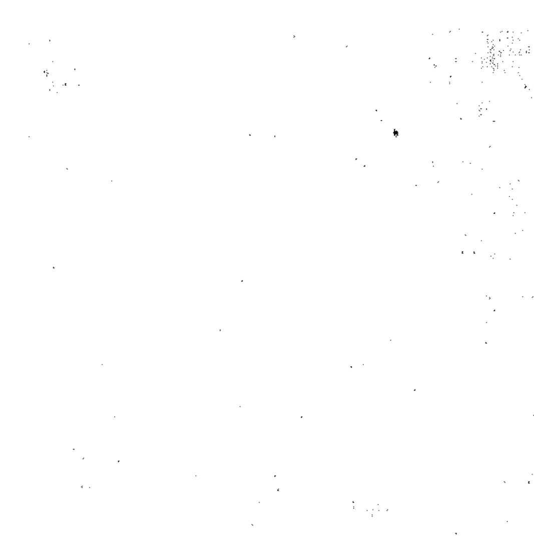
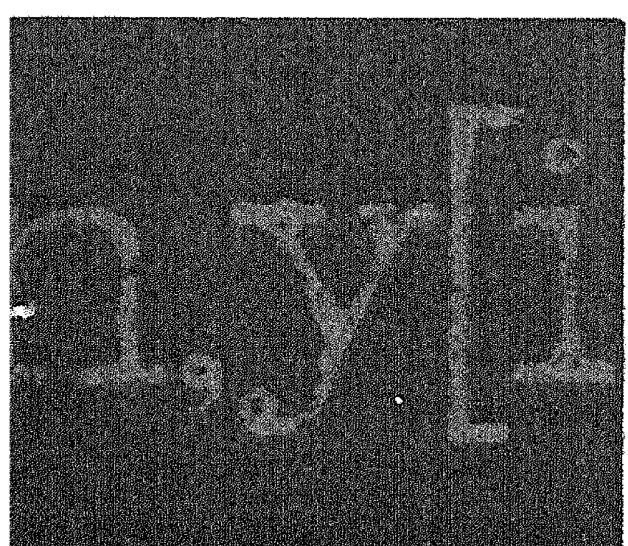


ABC ALGOL

A PORTABLE LANGUAGE FOR FORMULA MANIPULATION SYSTEMS

PART 1 : THE LANGUAGE

R.P. VAN DE RIET

MATHEMATICAL CENTRE TRACTS

46

MATHEMATICAL CENTRE TRACTS 46

R.P. VAN DE RIET

ABC ALGOL

A PORTABLE LANGUAGE FOR FORMULA MANIPULATION SYSTEMS

PART 1 : THE LANGUAGE

MATHEMATISCH CENTRUM AMSTERDAM 1973

ACM - Computing Reviews - category: 4.22

Table of contents of Part 1

0.	Introduction	1
0.1.	Summary	1
0.2.	Acknowledgements	3
1.	First acquaintance with the language	5
1.1.	An overview of the language	5
1.2.	Solution of differential equations in ABC ALGOL	10
1.3.	A first complete ABC ALGOL program	14
2.	Formal definition of ABC ALGOL	27
2.1.	Program	27
2.2.	Block	27
2.3.	Declaration	28
2.3.1.	Formula declaration	28
2.3.2.	Formula array declaration	29
2.3.3.	Procedure declaration	29
2.3.3.1.	The specification part	30
2.3.3.2.	Operator identifiers	31
2.3.3.3.	Assignment to formula procedure identifier	34
2.4.	Statement	34
2.5.	Expression	35
2.6.	The procedure replace	40
2.7.	Error messages and end of compilation	42
2.8.	ABC ALGOL compared with other languages	43
3.	Simplification and rational numbers	46
3.1.	The representation of the formulae	46
3.2.	How to use the system	49
3.3.	The procedures of the system	51
3.4.	The rational-number system	63
3.5.	Some examples	77
3.5.1.	Test of the rational-number system	78
3.5.2.	Test of the formula system	79
3.5.3.	A more sophisticated example	80
3.6.	An alternative simplification algorithm	90
4.	The translated program	98
4.1.	The simplest but unrealistic translation	100
4.2.	The dynamic-storage allocation system	101

4.3.	The external non-relocation method	104
4.3.1.	Translating an expression	106
4.3.1.1.	The reference-to-a-name approach	106
4.3.1.2.	The reference-to-a-value approach	108
4.3.2.	The translation of parameters of type formula	112
4.3.3.	The translation of a formula procedure	113
4.3.4.	An example	114
4.4.	A more clever and more efficient system	115
4.5.	Formula procedures	123
4.6.	Jumps leading out of a block	127
4.7.	The basic structures	131
4.8.	The run-time system	136
5.	The compiling process	144
5.1.	The overall strategy	144
5.1.1.	The lexical scan	145
5.1.2.	The syntactical scan	146
5.1.3.	The information list	147
5.1.4.	The reading mechanism	151
5.1.5.	Error detection and error recovery	152
5.1.6.	Identifiers	155
5.2.	The compiler from begin to end	157
5.2.1.	Prerequisites for translation process	158
5.2.2.	Translation of a block	159
5.2.3.	Translation of a declaration	159
5.2.3.1.	Translation of a formula declaration	160
5.2.3.2.	Translation of a formula-array declaration	160
5.2.3.3.	Opening and closing a block	162
5.2.3.5.	Translation of a procedure declaration	162
5.2.4.	Translation of a statement	163
5.2.4.1.	Translation of an assignment statement	163
5.2.4.2.	Translation of a procedure statement	165
5.2.5.	Translation of an expression	165
5.2.6.	The auxiliary equipment	166
5.2.7.	Initialization	166
5.2.8.	A procedure library	167
5.3.	Alphabetic listing of important identifiers	169

Table of contents of Part 2

6.	The ABC ALGOL compiler	1
6.0.	Preliminaries of translation process	1
6.1.	Prerequisites for translation process	1
6.2.	Translation of a block	2
6.3.	Translation of a declaration	4
6.3.1.	Translation of a formula declaration	6
6.3.2.	Translation of a formula-array declaration	9
6.3.3.	Opening and closing a block	14
6.3.4.	Translation of a (formula or type) procedure declaration	21
6.4.	Translation of a statement	29
6.4.1.	Translation of a conditional statement	30
6.4.2.	Translation of a for statement	31
6.4.3.	Translation of an unconditional statement	31
6.4.4.	Translation of an assignment statement	32
6.4.5.	Translation of a procedure statement	37
6.5.	Translation of an expression	40
6.5.1.	Translation of a simple other expression	42
6.5.2.	Translation of a simple formula expression	45
6.6.	Auxiliary equipment	51
6.6.1.	Declaration and initialization of symbols	51
6.6.2.	The text-reading equipment	55
6.6.3.	The printing equipment	70
6.6.4.	The information-cells equipment	74
6.6.5.	The main program	78
7.	Examples of compiled programs	81
7.1.	General structure and information list	81
7.2.	Block-entry, block-exit	87
7.3.	Formula array in a procedure body	91
7.4.	Translation of procedure parameters	93
7.5.	Protection mechanism for formula procedures	94
7.6.	Assignment to formula-procedure identifier	99
7.7.	Error detection and error recovery	101
8.	Machine dependencies	109
8.1.	Size of tables	109

8.2.	The MC-ALGOL 60 standard procedures	110
8.3.	Other machine-dependent features	111
9.	References	115

0. Introduction

The language ABC ALGOL may be described as follows:

ABC ALGOL = ALGOL 60 + arbitrary data structures
+ operator definitions.

It was originally motivated by a desire to write formula-manipulation systems in ALGOL 60, but it is also useful in other areas, where the objects are dynamically changing data structures, as, for example, in artificial intelligence.
The ABC ALGOL compiler is written in ALGOL 60; ABC ALGOL programs are compiled to ALGOL 60 and the run-time system for running the compiled program is written in ALGOL 60, so that the only necessary prerequisite for using ABC ALGOL is a good ALGOL 60 system. All the ALGOL 60 programs which one needs are fully reproduced in the following chapters.
The ABC ALGOL system was designed with emphasis on clarity of design (for educational and "ethical" purposes) and on portability (an important issue in the software-engineering field) rather than on efficiency with respect to execution speed. However, a considerable gain in efficiency can be obtained by rewriting the few procedures of the run-time system in machine-code.

0.1. Summary

The first MC TRACT on formula manipulation in ALGOL 60 [11] described a system of ALGOL 60 procedures which can be used in an ALGOL 60 program to manipulate formulae. Moreover, it described an interpreter which reads and executes so-called formula programs, i.e. sequences of statements, each one performing a certain formula-manipulation function. The formula-manipulation used in this early system was not provided with dynamic storage allocation techniques.
In the second MC TRACT [12] on this subject, the formula- manipulation apparatus is applied to the computation of functions defined by a Cauchy-problem, i.e. a set of partial differential equations together with initial conditions. In addition, a small chapter is devoted to complex arithmetic in ALGOL 60.

It is the purpose of this MC TRACT and of the following MC TRACT, in which the second part is published, to derive techniques for manipulating formulae using dynamic storage allocation, together with an automatic garbage collector.
Like its predecessor, ABC ALGOL is an ALGOL 60 system consisting of ALGOL 60 procedures with which a user can describe his formula- manipulation programs. Due to the automatic storage allocation, this system of procedures is rather complicated and not easy to use. It therefore became necessary to design a language, in which a user can program conveniently, and to build a compiler for taking care of all the complexities. The language, called ABC ALGOL (ABC is an acronym for the Dutch: "Algebraische Bewerkingen met behulp van de Computer"), is an extension of ALGOL 60, the main addition being the introduction of a new type: formula. Data structures of this type are binary trees. It is, moreover, possible to define the ordinary operators (+, -, ×, etc.) for operands of type formula.

In chapter 1 the language is introduced by means of detailed examples.
In chapter 2 the ABC ALGOL language is described more precisely in the form of BNF rules, semantic rules and examples.
Chapter 3 contains a system of ABC ALGOL procedures as an aid to the user with as typical problem: the simplification of large expressions involving the +, -, × and / operators, together with rational numbers of arbitrarily long magnitude.
Chapter 4 describes several strategies for the run-time system, gradually increasing in complexity and efficiency. It also contains the ALGOL 60 procedures comprising the run-time system. It is noteworthy that this system, being the kernel of the ABC ALGOL system, needs only six pages.
Commentary on the overall structure of the compiler is given in chapter 5, while the rather voluminous ALGOL 60 text of the compiler is given in chapter 6. Not only are the syntax-reading procedures interesting, the way semantic information is extracted from the to-be-compiled program and manipulated with in an information list is also noteworthy.
The 7-th chapter contains worked-out examples demonstrating the subtleties of the compiling process.
The standard MC-ALGOL procedures for input and output used at the Mathematical Centre, together with restrictions and peculiarities of the EL-X8 computer, are described in chapter 8.

This tract is organized with several groups of readers in mind. First, the reader who wants to use an existing formula manipulation system only. He is advised to read the first half of chapter 1, chapter 2 and the first half of chapter 3. Second, the reader who wishes to program in ABC ALGOL and to make full use of it. For him chapters 1, 2 and 3 suffice. Third, the reader who wants to become acquainted with ABC ALGOL and its implementation. For him the whole tract has been written.

Relating this work to existing systems, we make the following remarks. Due to lack of manpower, the size of the project necessarily had to be small. In addition, the principles of the implementation were of more concern than the implementation itself. Finally, to make the system more useful, a large amount of standard (library) formula-manipulation procedures will have to be written.

Other systems are normally rather large (FORMAC, ALPAK, SAC, MATHLAB, to mention a few) while the host language is not ALGOL 60 but FORTRAN, PL/1, LISP or machine-code. This tract shows that ALGOL 60 is suited for manipulating data structures as complex as binary trees as well as for compiler-writing techniques.

A comparison of the language ABC ALGOL with some other languages derived from ALGOL 60 (SIMULA, Formula ALGOL, ALGOL W and ALGOL 68) is given in section 2.8.

For literature we refer to the proceedings of the second SIGSAM symposium [1] and the annotated bibliography prepared by J. Sammet [15a].

Chapters 0-5 are given in the first part of the tract; chapters 6-8 and the list of references are given in the second part.

0.2. Acknowledgements

The work, described in this tract, had to be carried out in spare moments and would definitely not have been finished without the enthusiastic and most inspiring contributions of the following people:

W.P. de Roever and G. ten Velden aided in the early stages of the design of the system and the construction of the compiler.

A. de Bruin contributed considerably by designing the simplification system in chapter 3 and by rewriting de Roever's rational number system into ABC ALGOL.

R. Wiggers added complicated portions to the compiler and wrote the alternative simplification procedures of section 3.6.
The author is, furthermore, most grateful to:
John W. Carr III, for painfully scrutinizing the text of this tract,
Mrs. M. Homburg-Knieper, for typing the text of this tract,
D. Grune, for the use of his "tekstschaaf" [4] for justifying the text of this tract,
Mr. D. Zwarst, Mr. J. Suiker and Mr. J. Schippers, for reproducing this tract
and to the operators of the EL - X8 computer for running the programs.

The investigations were carried out mainly while the author was employed by the Mathematical Centre. The author is grateful to the board of the MC for providing this opportunity. The tract was finished while the author was at the "Vrije Universiteit" in Amsterdam, where he is presently working. His address is:

Vrije Universiteit
afd. Informatica
de Boelelaan 1105
Amsterdam, The Netherlands

1. First acquaintance with the language

In this chapter, we give a short overview with some introductory ABC ALGOL programs, a rather sophisticated ABC ALGOL program for a non-trivial mathematical problem, and a complete ABC ALGOL system comprising operator definitions and representations of data structures.

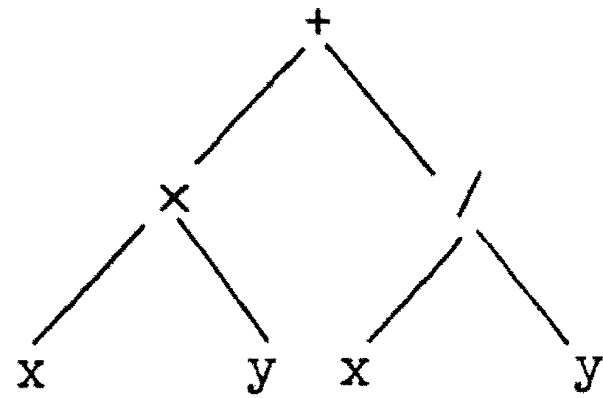
1.1. An overview of the language

In ABC ALGOL one can write programs for manipulating with formulae; the formulae being mathematical expressions involving operators, variables, numbers, functions, etc. Henceforth we shall use the term <u>formula</u> in this sense. Consider, as an example, the following piece of an ABC ALGOL program on the left-hand side of the page, together with the results shown on the right-hand side of the page. (The procedure "nlcr" prints a new line.)

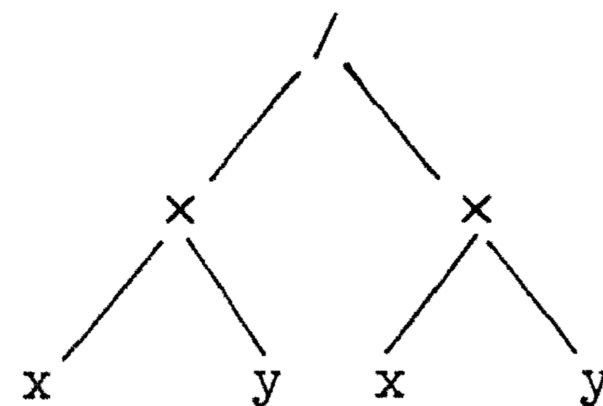
```
x:= av(⊀x⊁); output(x); nlcr;                      x
y:= av(⊀y⊁); output(y); nlcr;                      y
z:= av(⊀z star⊁); output(z); nlcr;                 z star
f:= x+y; output(f); nlcr;                          x+y
        output(lhs of f); nlcr;                    x
        output(rhs of f); nlcr;                    y
f:= x×y + x/y; output(f); nlcr;                    x×y+x/y
        output(lhs of f); nlcr;                    x×y
        output(rhs of f); nlcr;                    x/y
        output(lhs of rhs of f); nlcr;             x
        print(type of f); nlcr;                    +4
        print(type of lhs of f); nlcr;             +5
        print(type of rhs of f); nlcr;             +6
        print(type of lhs of rhs of f); nlcr;      +3
f:= 0; output(f); nlcr;                            0
f:= 1; output(f); nlcr;                            1
f:= 2; output(f); nlcr;                            2
f:= 1+2; output(f); nlcr;                          3
f:= 1+x; output(f); nlcr;                          1+x
f:= 0+x; output(f); nlcr;                          x
f:= 1×y + 0/x; output(f); nlcr;                    y
```

```
f:= (1+x) + (y+2); output(f); nlcr;              3+x+y
f:= (2×x) × (4×y)/(x×2 × y×2 × 2); nlcr;
                  output(f); nlcr;               x×y/(x×y)
```

The above piece of program demonstrates quite clearly the basic operations and the data structures in ABC ALGOL. For example, the formula x×y + x/y is represented as a binary tree with operators in its nodes:



while the last example is, after some transformations, represented as:



The above piece of program is embedded in a complete ABC ALGOL program containing also operator definitions. These operator definitions have to be given by the ABC ALGOL programmer, as they do not form a part of the ABC ALGOL system. The particular operator definitions, the representations for algebraic variables (the procedure av) and numbers and a declaration of the output procedure are given, also by way of example, in section 1.3. Quite a different set is defined in chapter 3 and it is likely that most ABC ALGOL programmers will not be satisfied with existing sets, so that other sets will be constructed. This shows that the ABC ALGOL system does not provide operator definitions and representations but gives the means to make operator definitions and to construct data structures tailored to one's specific problem in a convenient way only.

To show that ABC ALGOL is an extension of ALGOL 60, we now give an example looking very similar to an ALGOL 60 program. The example is a procedure for computing the determinant of a matrix with formulae as elements, such as:

```
| x     y     z   |
|                 |
| x×y   x×z   y×z |
|                 |
| z     y     x   |
```

The ABC ALGOL procedure is as follows:

```
formula procedure determinant(n,matrix); value n; integer n;
       formula array matrix;
begin  integer array perm,perm1[1:n]; integer i; formula det;
       procedure add next term(m,p,perm sign); value m,perm sign;
       integer m,perm sign; integer array p;
       if m = 1 then
       begin   integer i; formula factor:= 1; perm[n]:= p[1];
               for i:= 1 step 1 until n do
               factor:= factor × matrix[i,perm[i]];
               det:= det + perm sign × factor
       end else
       begin   integer array q[1:m-1]; integer i,j;
               for i:= 1 step 1 until m do
               begin   perm[n - m+1]:= p[i];
                       for j:= 1 step 1 until i-1 do q[j]:= p[j];
                       for j:= i+1 step 1 until m do q[j-1]:= p[j];
                       add next term(m-1,q,perm sign × even(i-1));
                       comment The standard procedure even(n)
                       becomes +1 if the integer n is even,
                       otherwise -1;
       end end add next term;
       for i:= 1 step 1 until n do perm1[i]:= i;
       det:= 0; add next term(n,perm1,1); determinant:= det
end
```

The determinant of the example above will be calculated by the following program:

```
begin   formula array a[1:3,1:3]; formula x = av(≮x≯),
                                  y = av(≮y≯),z = av(≮z≯);
        a[1,1]:= x;   a[1,2]:= y;   a[1,3]:= z;
        a[2,1]:= x×y; a[2,2]:= x×z; a[2,3]:= y×z;
        a[3,1]:= z;   a[3,2]:= y;   a[3,3]:= x;
        output(determinant(3,a))
 end
 with the results:
 x×x×z×x-x×y×z×y-y×x×y×x+y×y×z×z+z×x×y×y-z×x×x×z×z
```

The above example could have been written in ALGOL 60, except that the array would have to have elements of type real instead of formula and that the declaration "formula factor:= 1" would have to have been written as "real factor; factor:= 1".

We now give an example for which no simple ALGOL 60 counterpart exists; it is the computation of a derivative, typical in the formula-manipulation field. We assume that the formulae to be differentiated have as operators the +, × and / operators only and as operands algebraic variables and numbers only. (As a-b can be written as a + (-1) × b we can easily circumvent the - operator).

```
formula procedure der(f,x); value f,x; formula f,x;
begin  integer t; t:= type of f;
       der:= if t = sum then der(lhs of f,x) + der(rhs of f,x)
       else if t = product then der(lhs of f,x) × rhs of f +
                                lhs of f × der(rhs of f,x) else
       if t = quotient then (der(lhs of f,x) × rhs of f -
              lhs of f × der(rhs of f,x))/(rhs of f × rhs of f)
       else if t = alg var then (if f = x then 1 else 0)
       else 0
end
```

The integers alg var, sum, product and quotient have, in the system of procedures of section 1.3, the values 3, 4, 5 and 6, respectively (see also the output of the program in the beginning of this section).
Another typical formula-manipulation procedure is a substitution procedure which substitutes the formula y for the algebraic variable x occurring in a formula f.

```
formula procedure subst(f,x,y); value f,x,y; formula f,x,y;
begin  integer t; t:= type of f;
       subst:= if t = sum   then subst(lhs of f,x,y) +
                                  subst(rhs of f,x,y)
       else if t = product  then subst(lhs of f,x,y) ×
                                  subst(rhs of f,x,y)
       else if t = quotient then subst(lhs of f,x,y)/
                                  subst(rhs of f,x,y)
       else if f = x then y else f
end
```

The above procedures are now tested with the following program. The results are shown again on the right-hand side of the page.

```
output(der(x+y,x)); nlcr;                      1
output(der(x×y + x×x,x)); nlcr;                y+x+x
output(der(x×x × x+x × y,x)); nlcr;            (x+x)×x+x×x+y
output(der((x×x×x + x×y)/(x + x×x),x));        (((x+x)×x+x×x+y)×(x+x×x)-
       nlcr; f:= x+y;                            (x×x×x+x×y)×(1+x+x))/
                                                 ((x+x×x)×(x+x×x))
output(subst(f,x,y)); nlcr;                    y+y
output(subst(subst(f,x,y),y,1));               2
       nlcr; f:= x×x + x×y + y×y; nlcr;
output(subst(f,x,x+y)); nlcr;                  (x+y)×(x+y)+(x+y)×y+y×y
```

Extensive use of the procedures der and subst is made in the next section.

We again emphasize that the pieces of ABC ALGOL in this section have to be supplied with operator definitions, representations of operands and output specifications in order to form complete ABC ALGOL programs. The test runs have been made using the apparatus of section 1.3. One is of course free to use the programs of section 1.3, but one may also define one's own apparatus, or use the procedures of chapter 3 for automatic simplification and rational number arithmetic.

At the end of this short overview we remark that the ABC ALGOL compiler translates an ABC ALGOL program into ALGOL 60 text, which, combined with a preceding standard set of ALGOL 60 procedures and enclosed between begin and end, forms a complete ALGOL 60 program. Execution of the ABC ALGOL program is accomplished by execution of this derived ALGOL 60 program.

1.2. Solution of differential equations in ABC ALGOL

A simple example, but one which clearly demonstrates the need for automatic formula-manipulation techniques, is the following:
Consider the differential equation:

(1.1) $$f(x,y,y') = 0, \qquad y(0) = a,$$

for the unknown function $y(x)$, where f is assumed to be analytic in its three variables; i.e. differentiable a sufficient number of times. Let b be a solution of $f(0,a,b) = 0$ and be given beforehand, we may then formally calculate the first n coefficients of the Taylor series expansion of the function $y(x)$ as follows:

$$\text{Let } y(x) = \sum_{i=0}^{n} c_i x^i + O(x^{n+1}), \text{ then } c_i = \frac{1}{i!} \frac{d^i y}{dx^i}\bigg|_{x=0} .$$

From the initial conditions we know: $c_0 = a, c_1 = b$.
Differentiating (1.1) with respect to x gives:

$$\frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{dy}{dx} + \frac{\partial f}{\partial y'}\frac{d^2y}{dx^2} = 0 .$$

Substituting $x = 0$, $y = a$ and $dy/dx = b$, we find:

$$c_2 = -\tfrac{1}{2}(f_x(0,a,b) + f_y(0,a,b)b)/f_{y'}(0,a,b),$$

where $f_x(0,a,b)$ means $\partial f/\partial x$ after substituting $x = 0$, $y = a$ and $y' = b$.

The general algorithm is now easy to understand: Introduce n functions $f^{(i)}(x,y,y',\ldots,y^{(i+1)})$, depending on i+3 independent variables $x,y,y',\ldots,y^{(i+1)}, i = 0,\ldots,n-1$, as follows:

$$f^{(0)}(x,y,y') = f(x,y,y'),$$

(1.2)

$$f^{(i)}(x,y,y',\ldots,y^{(i+1)}) = \frac{Df^{(i-1)}}{Dx} \underset{=}{\mathrm{def}} \frac{\partial f^{(i-1)}}{\partial x} + \sum_{j=0}^{i} \frac{\partial f^{(i-1)}}{\partial y^{(j)}} y^{(j+1)},$$

where $\partial/\partial t$ means partial differentation with respect to t.

By substituting $x=0$, $y=0$, $y=c_0$, $y'=c_1,\ldots,y^{(i+1)} = (i+1)!\ c_{i+1}$ into $f^{(i)}$ and equating $f^{(i)}$ to zero, we obtain the following equation:

$$(1.3) \quad f^{(i)}(0,c_0,c_1,\ldots,(i+1)!\ c_{i+1}) = 0.$$

This equation is linear in c_{i+1}; hence c_{i+1} can be determined by calculating the constant term by substituting $c_{i+1} = 0$ and by calculating the linear term by differentiating with respect to c_{i+1}.

We shall now give the description of the above algorithm in ABC ALGOL, in the form of a procedure "Compute Taylor coefficients". Auxiliary equipment in the form of procedures for creating algebraic variables, in the form of operator definitions and in the form of declarations and initialization of variables, will be given in section 1.3.

```
procedure Compute Taylor coefficients(f,x,y,ypr,a,b,c,n);
value f,x,y,ypr,a,b,n; formula f,x,y,ypr,a,b; integer n;
formula array c;
begin  integer i,j; formula fi,fim1,fs,z = av(⊀z⊁);
       comment z is a constant algebraic variable, which
       should be printed as the string "z" on output. z
       is  used to replace the unknown c[i+1];
       formula array yprime[0:n]; array fact[0:n];
       c[0]:= a; c[1]:= b; yprime[0]:= y; yprime[1]:= ypr;
       fim1:= f; fact[0]:= fact[1]:= 1;
       for i:= 1 step 1 until n-1 do
       begin  yprime [i+1]:= ar(⊀y1⊁,i+1);
              comment  yprime[i+1] is introduced now  as an
              algebraic variable, which, on  output, is printed
              as "y1["followed by the number  representation of
              i+1, closed by"]";
              fi:= der(fim1,x);
              comment der computes the partial derivative of fim1
              with respect to x;
              for j:= 0 step 1 until i do
              fi:= fi + der(fim1,yprime[j]) × yprime[j+1];
              comment this is the direct translation of (1.2);
              fact[i+1]:= (i+1) × fact[i];
              fs:= subst(fi,x,0); c[i+1]:= z;
              comment subst makes the substitution x=0 in fi;
              for j:= 0 step 1 until i+1 do
              fs:= subst(fs,yprime[j],fact[j] × c[j]);
              comment in accordance with (1.3);
              c[i+1]:= -subst(fs,z,0)/der(fs,z);
              comment let fs = p+qz, then fs=0 means
              z = -p/q;
              fim1:= fi
       end
end Compute Taylor coefficients;
```

The direct translation of the mathematical description to a program shows the relatively simple way ABC ALGOL can be used for formula-manipulation processes.
The procedure "Compute Taylor coefficients" is completely general as it does only assume that the differential equation is analytic. No assumptions have been made otherwise on the form of the differential equation. In practice, this procedure must be embedded in an environment of ABC ALGOL procedures for operator definitions, such as those given in section 1.3. For particular definitions of derivative (der) and substitution (subst) see section 1.1. A testprogram for this procedure is the following:

```
begin   formula x = av(‹x›),y = av(‹y›),yp = av(‹yp›),
                f = y×y + yp×yp - 1;
        formula array c[0:10]; integer i;
        Compute Taylor coefficients(f,x,y,yp,0,1,c,10);
        for i:= 0 step 1 until 10 do begin nlcr; print(i);
                                           output(c[i])
                                     end
end
```

with the following results (the sin(x) expansion):

```
+0      0
+1      1
+2      0
+3      -1/6
+4      0
+5      1/120
+6      0
+7      -1/5040
+8      0
+9      1/362880
+10     0
```

1.3. A first complete ABC ALGOL program

In this section we implement a particular, very simple, system of ABC ALGOL procedures for formulae with only the operators +, -, × and /, with algebraic variables and with numbers, integers as well as reals. In this system some simple simplifications are automatically performed concerning numbers and combinations of products and quotients. For a system in which the full power of simplification is built in together with rational numbers of unrestricted length, we refer to chapter 3.
A reader not interested in writing his own ABC ALGOL system but only interested in the system of chapter 3. may skip this section.

The remainder of this section shall be given in the form of an ABC ALGOL program amply supplied with comment. As we need sometimes semicolons in comment we have adopted the convention that in ABC ALGOL the symbol ? replaces the semicolon in comment. Here follows the program:

<u>begin</u> <u>comment</u>

The basic structures from which the data structures of type <u>formula</u> can be constructed are:
1. The <u>constant structure</u> consisting of three parts: a (small) positive integer: type and two integers: lhs and rhs. The size of lhs is the size of an ordinary X8 integer, i.e. abs(lhs) < 2 ↑ 26. The size of rhs, however, is more restricted, we require: abs(rhs) < 2 ↑ 17 -1. The structure is created by means of the expression: "<u>constant</u>(type,lhs,rhs)".
2. The <u>monadic structure</u> consisting of three parts: a (small) positive integer: type, an integer: lhs and a reference to another data structure of type formula: rhs. The structure is created by means of the expression: "<u>monadic</u>(type,lhs,rhs)". The term "monadic" is chosen, as this structure contains one reference to another data structure.
3. The <u>dyadic structure</u> consisting of three parts: a (small) positive integer: type, and two references to other data structures of type formula: lhs and rhs. The structure is created by means of the expression: "<u>dyadic</u>(type,lhs,rhs)". The term "dyadic" is chosen as this structure contains two references to other data structures.

4. The rowadic structure consisting of (n+1) parts: a (small) positive integer: type and n, $n \geq 0$, integers, called the elements of the rowadic structure. The structure is created by means of the expression: "rowadic(type,i,n,el i)", where for given i, $0 \leq i \leq n$, el i is the i-th element. E.g. if type = 10, n = 3 and the elements are 1, 7 and 8. we may create this by:

rowadic(10,i,3,if i = 1 then 1 else if i = 2 then 7 else 8)

One sees that the variable i plays the role of a Jensen variable.

5. The polyadic structure consisting of (n+1) parts: a (small) positive integer: type and n, $n \geq 0$, references to other data structures of type formula, called the elements of polyadic structure. The structure is created by means of the expression: "polyadic(type,i,n,el i)",where for given i, $0 \leq i \leq n$, el i is the i-th element. E.g. if type = 11, n = 4 and the elements u, x, y, z, we may create the structure by means of:

polyadic(11,j,4,if j = 1 then u else if j = 2 then x
else if j = 3 then y else z)

where we have now used another variable j as the Jensen variable.

The term "polyadic" is chosen as this structure contains many references to other data structures.

The small integer: type should be smaller than 31.
Besides the above mentioned constructive equipment, we have descriptive equipment for asking how a data structure has been constructed. For this purpose we may use the expressions:
"type of f" delivering the type of the data structure f,
"lhs of f" delivering the lhs of the data structure f,
"rhs of f" delivering the rhs of the data structure f,
"length of f" delivering the number of elements of the data structure f,
"el i of f" delivering the i-th element of the data structure f.

The constant structures can be used to store integers. For example, an integral number n can be represented by a constant structure as a result of executing:

"constant(integral number,n,0)",

where the rhs field is filled with the number 0.

A real number r is represented in the memory of the EL X8 by two consecutive words, called head and tail. The standard procedures "head of" and "tail of" decompose this real number into these words and deliver these words as two integers "head" and "tail". Representing these two integers in a constant structure by means of:

"constant(real number,head,tail)",

is not possible, although this seems to be very attractive, due to the restriction on the rhs field of a constant structure which may not exceed in absolute value 2 ↑ 17 - 1. Therefore, a more complex structure is used for a real number r:

"monadic(real number,head of (r),
constant(0,tail of(r),0))".

The two zero's meaning that the respective parts do not contain information. It must be explicitly stated, however, that the ABC ALGOL programmer is free in his choice for representing real numbers in the same way as he is free to choose a representation for algebraic variables or for expressions like a+b×c. The data structures suggest that a+b×c may be represented as a binary tree created through:

"dyadic(sum,a,dyadic(product,b,c))",

but it is not necessary. Only if a data structure has the form of a string, is the representation very specific: the consecutive characters of the string are packed, three in a word (with shift factor 2 ↑ 8 = 256) and the resulting integers are grouped together to form a rowadic structure with type = 31. Note that this type value is excluded for the programmer. Hence, the following piece of program:

f:= ⊀12345678910 a b c d e f⊁

will result, also counting the spaces, in a rowadic structure of 8 integer elements, the first one being the internal representation of the characters "123", the last one being the internal representation of the two characters " f". And (el 5 of f) ÷ 2 ↑ 16-1 will give the internal representation of the character "a".

The internal representation is the MC-resym value increased with one (see [5]).

We begin with two procedures for introducing algebraic variables. The first one is for a simple algebraic variable, the second one for a "subscripted" algebraic variable y[i]. If output is needed for y[10], say, then this variable is output as "y[10]".;

```
formula procedure av(s); value s; formula s;
av:= monadic(alg var,0,s);
formula procedure ar(s,i); value s,i; formula s; integer i;
ar:= monadic(alg var,i,s);
```

comment From the above choice we see that we have to require that ar should be called only with $i \neq 0$, and if it is called with i = 0 then it is no longer possible to discriminate the two cases. The actual parameter for s is assumed to be a string, so that "z:= av(‹z›)" and yprime[i]:= ar(‹ypr›,i)" indeed lead to introduction of data structures of type alg var, alg var being a small integer. Being familiar with type procedures in ALGOL 60, we have no problem with the fact that av and ar are declared to be formula procedures. The result of executing the procedure body should be a data structure of type formula. Also the occurrence of s in the value list does not confuse us as we are interested in a value, in the form of a rowadic data structure being a string, and not in a name to which we want to assign a data structure as value (as the parameter c, being the name of a formula array, of Compute Taylor coefficients).

The next step is to introduce procedures for introducing integral and real numbers. Here, however, we arrive at a difficulty. It would be very easy to declare the following procedures:

(Recall that we use the symbol ? for a semicolon occurring in comment)

```
formula procedure in(i)? value i? integer i?
   in:= if i = 0 then zero else if i = 1 then one else
        if i = -1 then minone else
        constant(integral number,i,0)?
formula procedure rn(r)? value r? real r?
begin   integer h? h:= head of(r)?
        rn:= if h = 0 then in(tail of(r)) else
        monadic(real number,h,constant(0,tail of(r)),0))
end
```

With these procedures, however, we have to enclose each integer and each real number, occurring in a formula expression, in "in( )" and "rn( )", respectively. Thus we have to write: "f:= in(1) + rn(3.14) × x" instead of the much more simple expression: "f:= 1 + 3.14 × x". This, of course, is ridiculous as we have also a compiler which can very well see that 1 and 3.14 are an integer and a real number occurring in a formula expression. The problem is, however, that the compiler should know that 1 and 3.14 have to be treated as if there stood in(1) and rn(3.14), so that there should be a means for letting the compiler know that this is the case. This means is established by using as procedure identifier a very special one, so special that it can not be used for other purposes. In fact, the identifier "constant +" is used as procedure identifier instead of "in" and the identifier "constant ×" is used as procedure identifier instead of "rn". Extraordinary as it may seem, we have the following correct ABC ALGOL procedure declarations.;

```
formula procedure constant +(i); value i; integer i;
   constant +:= if i = 0 then zero else if i = 1 then one
                else if i = -1 then minone
                else constant(integral number,i,0);
formula procedure constant ×(r); value r; real r;
begin   integer h; h:= head of(r);
        constant ×:= if h = 0 then tail of(r) else
        monadic(real number,h,constant(0,tail of(r),0))
end;
```

comment Observe that, if head of(r) = 0, the result is a data structure of type integral number as the compiler automatically recognizes "tail of(r)" to be an integral expression at a place where a formula expression is required.

Observe also that with the above procedures only one data structure can be present representing 0 and only one data structure for representing 1 and only one for -1, the formula variables referring to these data structures being zero, one and minone. Initializing values to zero, one and minone should be done very carefully as e.g. "zero:= 0" leads to a circular definition (something of the style zero:= zero).

Similar to procedures for treating numbers, we have to make procedures for treating a sum, a difference, a product and a quotient of two operands.
Again, procedures of the following form:

```
formula procedure sm(a,b)? value a,b? formula a,b?
sm:= dyadic(sum,a,b)?
formula procedure pt(a,b)? value a,b? formula a,b?
pt:= dyadic(product,a,b)?
```

would be useless, as a formula expression like "x + y × z" should then be written as "sm(x,pt(y,z))". Therefore, procedures with very special identifiers have to be declared:

- dyadic + for the + operator,
- dyadic - for the - operator,
- dyadic × for the × operator,
- dyadic / for the / operator,
- monadic + for the monadic + operator (as in "+x-y"),
- monadic - for the monadic - operator (as in "-x+y").

In fact, these procedures are operator definitions. It would have been possible also to introduce a special notation for them, say, "operator +(a,b)". The method of using the already available apparatus for formula procedures together with very special identifiers turned out to be amazingly efficient and simple, for the compiler writing as well as for learning the language. Peculiar possible constructions are the result, such as: "dyadic +:= dyadic(sum,a,b)", which yields a dyadic data structure representing a+b.

In the following procedures, we implement some simple simplifications as: 0+a = a, 1×a = a, 2+3 = 5.
The representation is such that no superfluous numbers occur. Therefore, the numbers occurring in (2+3), or in (2+x) + (3+y), or in 2×3, or in (2×x) × (3×y), or in (2/x) × (3/x) are combined. The method is roughly as follows. If e.g., the sum of two data structures, representing the formulae a and b, has to be formed, it is first checked whether a or b is zero, this not being the case it is checked whether a or b are numbers, or whether they are of the form "number + formula" in which case numbers are combined, this not being the case a dyadic structure is created to represent a + b.
For the × operator and the / operator the simplifications with zero's and one's are performed also together with the following transformations:

```
(x/y) × b --> (x×b)/y
a × (u/v) --> (a×u)/v
(x/y)/b   --> x/(y×b)
a/(u/v)   --> (a×v)/u
```

After these transformations have been applied, numbers are combined, so that (2×x) × (y×3) is stored as 6 × (x×y) and (4/x)/(2/y) as 2 × (y/x).
With respect to the - operator, we can be very short: a-b is represented as a + (-1) × b.
A definition of the + operator might have been the following:

```
formula procedure dyadic +(a,b)? value a,b? formula a,b?
dyadic +:= if a = zero then b else if b = zero then a else
           if "numbers can be  combined" then
           oper on num(sum,a,b) else dyadic(sum,a,b)?
```

The process of combining numbers is for all three operators almost the same. In fact the two operands can be of three forms: a number n, the sum (or product) of a number n and a non-number x, or a non-number x. There are therefore nine possibilities to be catered for. And these have to be programmed for the three operators, of which the / operator behaves slightly differently. The following procedure does the whole job, for all operators. The idea is to write a and b under all circumstances as n+x and m+y (for a sum) and to deliver the result in the form (n+m) + (x+y), where it may well be that some of the operands have the value zero so that the automatic zero-simplification takes care of transformations necessary to carry (0+b) + (a+0) over into b+a, in case that b is a number and a does not contain a number component. Only when it is evident that the result (n+m) + (x+y) is not meaningful the shorter dyadic(sum,a,b) is delivered. This is the case when b does not contain a number component (nb = false) and a is not of the form n+x (nx = false)(for nb and nx see the procedure below).;

```
formula procedure comb num(oper,a,b); value oper,a,b;
        integer oper; formula a,b;
begin   formula n,m,x,y; Boolean nx,nb; integer ta,tb,oper2;
        real ra,rb;
        ta:= type of a; tb:= type of b; nx:= nb:= false;
        oper2:= if oper = quotient then product else oper;
        n:= m:= if oper = sum then zero else one; x:= a; y:= b;
        if ta = oper2 then
        begin   if num(type of lhs of a) then
                begin   nx:= true; n:= lhs of a;
                        x:= rhs of a
                end
        end else if num(ta) then begin x:= n; n:= a end;
        if tb = oper2 then
        begin   if num(type of lhs of b) then
                begin   nb:= true; m:= lhs of b;
                        y:= rhs of b
                end
        end else if num(tb) then begin nb:= true;
                                       y:= m; m:= b
                                 end;
        if nb then
        begin   ra:= if type of n = integral number then lhs of n
                     else compose(lhs of n, lhs of rhs of n);
                rb:= if type of m = integral number then lhs of m
                     else compose(lhs of m, lhs of rhs of m);
                ra:= if oper = sum then ra + rb else if oper =
                     product then ra × rb else ra/rb; n:= ra
        end;
        comb num:= if ¬ (nx v nb) then
                dyadic(oper,a,b) else
                if oper = sum then n + (x+y) else
                if oper = product then n × (x×y) else n × (x/y)
end;

formula procedure dyadic + (a,b); value a,b; formula a,b;
```

```
dyadic +:= if a = zero then b else if b = zero then a
          else comb num(sum,a,b);

formula procedure dyadic - (a,b); value a,b; formula a,b;
begin integer i; i:= -1; dyadic -:= a+i×b end;

formula procedure dyadic × (a,b); value a,b; formula a,b;
dyadic ×:= if a = zero v b = zero then zero else
          if a = one then b else if b = one then a else
          if type of a = quotient then (lhs of a×b)/rhs of a else
          if type of b = quotient then (a × lhs of b)/rhs of b
          else comb num(product,a,b);

formula procedure dyadic / (a,b); value a,b; formula a,b;
dyadic /:= if a = zero then zero else if b = zero then
          error(‹zero  denominator›) else if b = one then a else
          if type of a = quotient then lhs of a/(rhs of a×b) else
          if type of b = quotient then (a × rhs of b)/ lhs of b
          else comb num(quotient,a,b);
```

comment Besides the dyadic operators, we have the monadic operators and the Boolean procedure num;

```
formula procedure monadic +(a); value a; formula a;
monadic +:= a;
formula procedure monadic -(a); value a; formula a;
monadic -:= 0-a;
Boolean procedure num(t); value t; integer t;
num:= t = integral number v t = real number;
```

comment Next follows the procedure for defining the output of a formula f. The number of brackets appearing in the output is minimal. Moreover, minus operators are produced in constructions like:

```
(-30 + ((((-1 × f) + ((-20 × g) + h)) + (7 × z)) +
        (((-1/k) + (-40/l)) + u)))
```

which is output as:

```
-30-f-20×g+h+7×z-1/k-40/l+u;
```

```
procedure output(g); value g; formula g;
begin procedure op(f,env); value f,env; formula f; integer env;
  begin comment env denotes the operator of the environment,
    i.e. assuming that f is an operand, env is the operator
    of which f is an operand. As the values of sum, product
    and quotient are increasing, it is not difficult to see
    when brackets are necessary: when the type of f is smaller
    than env;
    integer t; t:= type of f;
    if dyadic f then
    begin comment This is a third form in which we encounter
      dyadic. Now it is a monadic operator with f as operand
      delivering the value true if f is a dyadic data structure
      and the value false otherwise;
      if t < env then printtext(‹(›);
      if t = sum then
      begin first term:= true; get terms(f) end
      else if t = product then
      begin op(lhs of f,t); printtext(‹×›);
        op(rhs of f,t)
      end else
      if t = quotient then
      begin op(lhs of f,product); printtext(‹/›);
        op(rhs of f,quotient)
      end; if t < env then printtext(‹)›)
  end else

  if t = integral number then
  begin t:= lhs of f;
    if t < 0 then
    begin if env ≠ 0 then printtext(‹(›); printtext(‹-›) end;
```

```
    pr int num(abs(t));
    if t < 0 ∧ env ≠ 0 then printtext(‹)›)
  end else

  if t = real number then
  begin real r,r0,A0,A1,A2,B0,B1,B2,a; integer s,n;
    r:= compose(lhs of f,lhs of rhs of f);
    if r < 0 then
    begin if env ≠ 0 then printtext(‹(›); printtext(‹-›) end;
    s:= sign(r); r:= r0:= abs(r);
    A0:= B1:= 0; A1:= B0:= 1;
    for n:= 1 step 1 until 20 do
    begin a:= entier(r0 + .5); r0:= r0 - a;
      A2:= a × A1 + A0; B2:= a × B1 + B0;
      if abs(1 - A2/(r × B2)) ≤ ⏨-10 then goto RAT;
      r0:= 1/r0; A0:= A1; A1:= A2; B0:= B1; B1:= B2
    end;
  FLOAT: n:= entier(ln(r) × .4343);
    r:= r × 10 ↑ (11 - n);
    if r < ⏨11 then begin n:= n - 1; r:= r × 10 end;
    printtext(‹.›); pr int num(r); printtext(‹⏨›); pr int num(n + 1);
    goto OUT;
  RAT: pr int num(abs(A2)); B2:= abs(B2);
    if B2 ≠ 1 then begin printtext(‹/›); pr int num(B2) end;
  OUT: if s < 0 ∧ env ≠ 0 then printtext(‹)›)
  end else

  if t = alg var then
  begin integer i,l;
    procedure pr char(w); value w; integer w;
    if w ≠ 0 then
    begin pr char(w ÷ 256); prsym(w-w ÷ 256×256-1) end;
    l:= length of rhs of f;
    for i:= 1 step 1 until l do pr char(el i of rhs of f);
    if lhs of f > 0 then
    begin printtext(‹[›); absfixt(2,0,lhs of f); printtext(‹]›)
  end end end op;
```

```
Boolean first term;
procedure printplus;
begin if ¬ first term then printtext(⩤+⩥);
  first term:= false
end;

procedure get terms(f); value f; formula f;
begin comment This procedures is called with as actual parameter a
  sum. The terms of this sum have to be output preceded by the
  proper + or - operators, the first term, however, should not be
  preceded by a + operator;
  integer s,t; t:= type of f;
  if t = sum then
  begin get terms(lhs of f); get terms(rhs of f) end
  else if t = product ∨ t = quotient then
  begin if num(type of lhs of f) then
    begin s:= sign(lhs of lhs of f);
      comment Note that the head of an X8 real number
      contains the sign as does also the tail;
      if s = -1 then begin printtext(⩤-⩥); first term:= false end
      else print plus;
      if lhs of f = minone ^ t = product then op(rhs of f,sum)
      else begin op(s × lhs of f,t);
             if t = product then printtext(⩤×⩥) else printtext(⩤/⩥);
             op(rhs of f,t)
           end
    end else
    begin print plus; op(f,sum) end
   end else begin print plus; op(f,0) end
 end get terms;
 op(g,0)
end output;

procedure pr int num(n); value n; real n;
if n < 10 then prsym(n) else
begin real m; m:= entier(n/10);
  pr  int num(m); pr sym(n-m×10)
end;
```

```
comment The final procedure to be declared is:;

formula procedure error(string); string string;
begin nlcr; printtext(string); exit; error:= 1 end;

comment All operators being defined and all procedures being declared, it
remains to declare and initialize the constants:;

integer real number,integral number,alg var,sum,product,quotient;
formula one = constant(2,1,0),zero = constant(2,0,0),
        minone = constant(2,-1,0);
real number:= 1; integral number:= 2; alg var:= 3; sum:= 4;
product:= 5; quotient:= 6;

begin   comment The particular programs of sections 1.1 and 1.2
        should be placed here. For example, a calculation of
        some Taylor coefficients, as described in section 1.2,
        may be programmed as follows:
        procedure Compute Taylor coefficients(...)? begin ... end?
        formula procedure der(f,x)? begin see section 1.1 end?
        formula procedure subst(f,x,y)? begin see section 1.1 end?
        begin   formula x = av(⊰x⊱),y = av(⊰y⊱),yp = av(⊰yp⊱)?
                formula array c[0:10]? integer i?
                Compute Taylor coefficients(y×y - yp×yp - 1,x,y,yp,0,1,c,10)?
                for i:= 0 step 1 until 10 do
                begin nlcr? print(i)? output(c[i]) end
        end
end; comment Finally, we come to the last end of the program;
end
```

2. Formal definition of ABC ALGOL

Having introduced the language in the preceding chapter by means of examples, we now want a precise definition in this chapter.
ABC ALGOL is a pure extension of ALGOL 60, without the <u>own</u> concept, so that we may refrain from repeating the ALGOL 60 report. The description given here consists of some BNF rules, some semantic descriptions and a few examples. The subdivision into sections is done in correspondence with the subdivision of chapters 5 and 6, so that it is easy to compare the language description of a certain syntactic variable with that part of the compiler where this syntactic variable is treated. The description is top to bottom.

In order to run the compiled program, one has to add the run-time system (section 4.8) of ALGOL 60 procedures in front and two <u>end</u>'s at the end.
The result is a complete ALGOL 60 program which can be run. The first action of this program is to read an integer, defining the size of the two arrays for storing formulae. Thus, on input tape an integer must appear. For the EL - X8 computer of the MC, this integer may be chosen equal to 15000.

2.1. Program

<program>::= <block><u>;</u>|<compound statement><u>;</u>

From this it follows that ABC ALGOL is not a pure ↑ 2 extension of ALGOL 60 as labels before the first block are not allowed, and a <u>;</u> must close the program.

2.2. Block

<block>::= <block head> <compound tail>
<block head>::= <u>begin</u> <declaration with semicolon>|
<block head> <declaration with semicolon>
<declaration with semicolon>::= <declaration>;
<compound tail>::= <statement> <u>end</u>|<statement>;<compound tail>

This is similar to the ALGOL 60 block.

2.3. Declaration

```
<declaration>::= <formula declaration>|<formula array declaration>|
      <formula procedure declaration>|<procedure declaration>|
      <type declaration>|<array declaration>|<switch declaration>
```

The first three declarations will be considered in this section; for the latter four we refer to the ALGOL 60 report.

2.3.1. Formula declaration

```
<formula declaration>::= formula <formula list>
<formula list>::= <formula definition>|<formula list>,<formula definition>
<formula definition>::= <simple variable> <formula initialization>
<formula initialization>::= <empty>|:= <formula expression>|
                               = <formula expression>
<simple variable>::= <identifier>
```

There are three different types of formula declarations:

1. formula v1; In the block, a value of type formula may be assigned to v1 and, after that, one may ask this value by placing v1 in an expression.
2. formula v2:= form expr; The role of v2 is the same as that of v1, except that, upon declaration v2 gets immediately an (initial) value resulting from execution of the formula expression "form expr".
3. formula v3 = form expr; The role of v3 is the same as that of v2, except that v3 may not be used for assigning a value. It may be used only in an expression where it then denotes the value of the formula expression as calculated once upon declaration. Variables of this type will be called constant variables.

In the following example, the three different declarations are shown functionally for the three variables aux, sum and x:

```
begin   formula x = av(‘x’), sum:= 1,aux; integer i;
        for i:= 1 step 1 until 10 do
        begin aux:= sum × x; sum:= sum + aux × aux end
end
```

Declarations are executed from left to right, so that "formula x = av(⋖x⋗), f:= x+x" is meaningful. The other order: "formula f:= x+x,x = av(⋖x⋗)" leads to an undefined situation and a run-time error will be the result.

2.3.2. Formula array declaration

A formula array declaration is defined as follows:
<formula array declaration>::= formula array <array list>
For <array list>, we refer to the ALGOL 60 report.
Example:
formula array fa1[lb1:ub1],fa2[lb2:ub2,lb3:ub3];
One may assign values to the array elements and the array elements may be used in formula expressions to denote their values.
It is not possible to combine initialization and declaration as was the case for ordinary formula variables.

For subscripted and for ordinary variables of type formula, run-time error messages are given when they are used in expressions without having obtained values beforehand. Run-time error messages are given also when constant variables are used to assign values, either explicitly when they occur in the left-hand side of an assignment statement (in which case a compile-time error message is given) or implicitly when they are transferred as call-by-name parameters in procedure calls.

2.3.3. Procedure declaration

Procedure declarations are almost the same as in ALGOL 60; only in the following aspects they differ:
1. Besides the ordinary specifiers one may use: formula, formula array and formula procedure.
2. Let p be a formal parameter specified as (formula, real, integer or Boolean) procedure, then the types of the possible parameters of p can be specified also.
3. One may use some very special identifiers, operator identifiers, for declaring operators by means of formula procedures.
4. The final action of a body of a formula procedure must be the assignment to the procedure identifier.

2.3.3.1. The specification part

Syntactically, we may define the specification part in the following way:

```
<specification part>::= <empty>|<specification list>;
<specification list>::= <specification>|<specification list>;<specification>
<specification>::= <ordinary specification>|<formal procedure specification>
<ordinary specification>::= <specifier> <identifier list>
<specifier>::= string|<type>|array|<type>array|label|switch|
        procedure|<type> procedure|formula|formula array
<identifier list>::= <identifier>|<identifier list>,<identifier>
<type>::= real|integer|Boolean|boolean
<formal procedure specification>::=
        <formal procedure specifier> <formal procedure segment list>
<formal procedure specifier>::= procedure|<type> procedure|
                                formula procedure
<formal procedure segment list>::= <formal procedure segment>|
        <formal procedure segment list>,<formal procedure segment>
<formal procedure segment>::= <identifier> <specification of parameters>
<specification of parameters>::= (<type of parameters list>)
<type of parameters list>::= <type of parameters>|
        <type of parameters list>,<type of parameters>
<type of parameters>::= <empty>|<specifier>|formula value
```

The effect of specifying a formal parameter of type *formula* or of type *formula array* is that appropriate actions are taken when the parameter occurs in the body. These actions depend on whether the parameter has been placed in the value list or not.

If a *formula* parameter is placed in the value list, the value of the actual parameter is computed once and this value is used when the value of the formal parameter is needed in the procedure body. It is forbidden to assign a value to a *formula* parameter called by value.

If a *formula* parameter is *not* placed in the value list, the value of the actual parameter is computed each time it is needed in the procedure body. For such a parameter, a formula variable can be the actual parameter for which assignment in the procedure body is possible.

With respect to memory space used, we remark that a called-by-value formula parameter does not always cost much memory space for introduction of a local formula variable; on the contrary, very frequently creation of a local integer variable (costing one word of memory space, instead of three words for a formula variable) is the only effect. It is for efficiency reasons that it is not allowed to assign a value to a formal parameter called-by-value.

For a formal parameter of type formula array, we remark that, if it occurs in the value list, it is wise to use the values of the array elements only, as assignment to the array elements may lead to undefined results. One does not get an error message, however. If it does not occur in the value list, we have the usual situation where the values of the array elements may be used and where assignments to the array elements are allowed.

Concerning the formal procedure specification, e.g. of the form: "procedure proc(,,string,real,array,formula,formula value,,,)" we remark that the actual parameter should be a procedure with 10 parameters, the type of the first two and last three may be anything, the type of the third should be a string, the type of the fourth should be a real, the type of the fifth should be an array, the type of the sixth should be a called-by-name formula and the type of the seventh should be a called-by-value formula.
It is required that, if one has a formal procedure parameter the actual procedure having parameters of type formula, the parameters of this formal procedure are specified by means of a formal procedure specification.

Although not required, it is advised to specify all formal parameters as this leads to shorter and faster running object programs (see section 4.6).

2.3.3.2. Operator identifiers

The procedure headings of formula procedures, defining operators, should be of a very specific form, given below.

For the dyadic + operator:
formula procedure dyadic + (a,b); (value a,b;) formula a,b;
For the dyadic - operator:
formula procedure dyadic - (a,b); (value a,b;) formula a,b;
For the dyadic × operator:
formula procedure dyadic × (a,b); (value a,b;) formula a,b;
For the dyadic / operator:
formula procedure dyadic / (a,b); (value a,b;) formula a,b;
For the dyadic : operator:
formula procedure dyadic : (a,b); (value a,b;) formula a,b;
For the dyadic ↑ operator:
formula procedure dyadic ↑ (a,b); (value a,b;) formula a,b;
For the monadic + operator:
formula procedure monadic + (a); (value a;) formula a;
For the monadic - operator:
formula procedure monadic - (a); (value a;) formula a;
For the monadic integral exponent operator:
formula procedure monadic ↑ (a,n); (value a,n;) formula a; integer n;
For integral numbers in formula expressions:
formula procedure constant + (n); (value n;) integer n;
For real numbers in formula expressions:
formula procedure constant × (r); (value r;) real r;

The identifiers a, b, n and r may be replaced by others and all parameters may be put in the value list or not, indicated by "(value a,b;)".
The meaning of the above operator definitions can be given only in connection with formula expressions.
Consider such an expression; if it is of the form: "expr1 operator expr2", then change it into: "dyadic operator(expr1,expr2)" and do the same for expr1 and expr2. If it is of the form: "operator expr", change it to: "monadic operator(expr)" and do the same for expr. If it is an integer or a real:ir, then change it to "constant + (ir)" or "constant × (ir)".
If the result of the changes is: "dyadic ↑ (expr1,in)", where "in" is an integer consisting of digits, as "2" or "31", then change the result again into: "monadic ↑ (expr1,in)".

The result is a polish form in which all operators are removed and changed into function designators. The identifiers of these function designators take on the form dyadic +, ..., constant ×. The effect of executing the original formula expression is the effect of executing the virtual expression consisting of nested function designators. Execution of the function designators means execution of the procedure bodies of the corresponding formula procedure declarations after applying the parameter substitution mechanism.

It is worth while to describe here what the compiler does. It changes the above operator identifiers into: S, D, P, Q, IQ, E, PL, MI, IE, IN and RN, so that we could think of the following formula procedure declarations:
formula procedure S(a,b); (value a,b;) formula a,b;
...
formula procedure RN(r); (value r;) real r;
Moreover, the compiler changes a formula expression into polish prefix and changes the operators into S(,), ..., RN( ), taking due account of the priority rules.
Example: x + y - (+a) × (-b)/c ↑ 5 ÷ 1024 ↑ 3.14 is treated as:
D(S(x,y),IQ(Q(P(PL(a),MI(b)),IE(c,5)),E(IN(1024),RN(3.14))))

As a matter of fact, if we had forbidden the programmer to use operators, he would have been forced to use functions defined by formula procedure declarations. Now, the compiler is helpful and does the polish prefix translation. It is reasonable to ask the programmer to define his operators by means of some special operator identifiers.
Note that integers and reals are not treated as such when they occur as an integral or real denotation only, i.e. with digits, point and little ten, but also when they occur in the form of a variable or function designator declared of integral or real type. From the above description it follows that the compiler does not combine integers or reals; i.e. "1 + 2 + x" is treated as "S(S(IN(1),IN(2)),x)". This is fortunately the case: it may very well be that the operator + has som very peculiar properties. For example, it is easy to define + in such a way that (1+2) + 3 = 0 and 1 + (2+3) = 10.
Due to some deeply lying reasons (see section 4.5), it is not allowed to declare operator identifiers within the body of formula procedure declarations.

It does not need saying that operator identifiers may not occur in formula expressions.

2.3.3.3. Assignment to formula procedure identifier

The final action in a formula procedure body, when the body is left via the end and not via a jump to a label, should be the assignment to the procedure identifier. Run-time error messages are given when after the assignation to the procedure identifier and before the end is reached something is done involving creation of formula values or calling of formula procedures. Run-time errors are given also when the procedure identifier has not obtained a value anyhow. This restriction does not hold for other type procedures. The reasons for this restriction are: efficiency of memory space (without the restriction one would nearly always need extra local variables introduced by the compiler), the fact that in other languages (e.g. ALGOL 68) this restriction is also present and the fact that the programmer can himself create very easily a local variable if necessary at the cost of memory space.

2.4. Statement

We have the ordinary statements as in ALGOL 60 together with a formula assignment statement. Moreover, function designators and procedure statements may have variables of formula type and formula expressions as actual parameters.
A formula assignment statement is defined as follows:

<formula assignment statement>::= <left part list> <formula expression>
<left part list>::= <left part>|<left part list> <left part>
<left part>::= <variable>:= |<procedure identifier>:= |
<operator identifier>:=

For variable and procedure identifier, we refer to the ALGOL 60 report.
<operator identifier>::= dyadic +|dyadic -|dyadic ×|dyadic /|
dyadic :|dyadic ↑|monadic +|monadic -|monadic ↑|
constant +|constant ×

That a formula assignment statement is useful for assigning a formula value to a formula variable or a formula procedure identifier (either in the form of an ordinary identifier or as an operator identifier) is evident.

With respect to for statements, we remark that the controlled variable may not be of formula type.

Note that it is not allowed to assign a formula value to the lhs, or the rhs, or the i-th el of a formula variable. For a discussion of this restriction and another construction, we refer to the replace mechanism described in section 2.6.

2.5. Expression

There are several expressions: arithmetic, Boolean, designational and formula expressions.
Designational expressions are defined exactly the same as in ALGOL 60.
Boolean expressions are the ALGOL 60 Boolean expressions with one extension: a Boolean primary may also have the form of a category enquiry, to be specified in a moment.
Formula expressions and arithmetic expressions are extensions of the ALGOL 60 arithmetic expressions. The extensions being:
1. an arithmetic primary may also have the form of a formula enquiry, a length enquiry or a type enquiry,
2. a formula primary may also have the form of a formula enquiry, a formula base or a string.
Furthermore, arithmetic primaries may occur in formula expressions and formula primaries may occur in arithmetic expressions. From this it follows that, syntactically, there does not exist any difference between a formula expression and an arithmetic expression. Being a formula expression or an arithmetic expression follows from the context only.

The formula bases and the enquiries are the elementary tools for creating formulae and for inspecting them. Syntactically they are defined as follows:

```
<formula primary>::= <ALGOL 60 primary>|<arithmetic primary>|
        <formula variable>|<formula enquiry>|<formula base>|<string>|
        (<formula expression>)|<formula function designator>
<arithmetic primary>::= <ALGOL 60 primary>|<formula primary>|
        <formula enquiry>|<type enquiry>|<length enquiry>
<Boolean primary>::= <ALGOL 60 Boolean primary>|<category enquiry>
<formula base>::= constant(<type>,<int arith expr>,<int arith expr>)|
        monadic(<type>,<int arith expr>,<formula expression>)|
        dyadic(<type>,<formula expression>,<formula expression>)|
        polyadic(<type>,<int id>,<length>,<formula expression>)|
        rowadic(<type>,<int id>,<length>,<int arith expr>)
<type>::= <int arith expr>
<int arith expr>::= <arithmetic expression>
<int id>::= <identifier>
<length>::= <int arith expr>
<formula enquiry>::= <kind of enquiry> of <formula name>
<kind of enquiry>::= lhs|rhs|el <arithmetic expression>
<type enquiry>::= type of <formula name>
<length enquiry>::= length of <formula name>
<formula name>::= <formula primary>
<category enquiry>::= constant <formula name>|monadic <formula name>|
   dyadic <formula name>|polyadic <formula name>|rowadic <formula name>
```

There are some restrictions to the values of the integral arithmetic expressions occurring above:

1. The value of <type> should be non-negative and not exceed 30.
2. The value of <int arith expr>, occurring as third parameter in a constant base, should, in absolute value, not exceed 2 ↑ 17 - 2.

In polyadic and rowadic bases, the identifier serves as a "Jensen" identifier; i.e. if this identifier is "i" and the actual fourth parameter is "fi", the base consists of a row of elements created as if the following statement were executed:

for i:= length step -1 until 1 do calculate fi.

Example: <u>polyadic</u>(10,i,n,x↑i/i) creates a structure, with type 10 and category polyadic, consisting of the first n terms of the sum x ↑ 1/1 + x↑2/2 + x↑3/3 + ...; it is assumed that x is an algebraic variable.
Another example: <u>rowadic</u>(11,j,m,digit[j]) creates a structure, with type 11 and category rowadic, consisting of m integers: digit[1], ..., digit[m], wich may be, for example, the integer decomposition of a large integer with respect to some base.

The enquiries form the counterpart of the formula bases. With them we can investigate the structure of a given formula value. Their relation to the formula bases is displayed in the following table. Assuming that f is being created by a statement of the form: f:= <u>constant</u>(t,n,m), the <u>lhs</u> <u>of</u> f, the <u>rhs</u> <u>of</u> f and the <u>type</u> <u>of</u> f deliver the integers n, m and t, respectively; when the Boolean primary <u>constant</u> f is used, it delivers the value <u>true</u>, while <u>monadic</u> f, ..., <u>rowadic</u> f deliver the value <u>false</u>. Note that category enquiries might also have been called "predicates".
A similar explanation applies for the other rows of the tables.
Not always is each formula enquiry allowed to be used as formula primary and as arithmetic primary, even if the object being enquired seemingly exists. The table contains, therefore, entries to indicate if both primaries are allowed or that one is forbidden leading to a run-time error message. The entries of the three columns <u>lhs</u>, <u>rhs</u> and <u>el</u> i, consist of two symbols at two rows, the upper symbol refers to the use of the enquiry as a formula primary, the lower one to the use of the enquiry as an arithmetic primary. The symbol ? means erroneous use, which ultimately will lead to a run-time error. The symbol / means: unusual, but allowed. (see the discussion below). The symbols n, m, a, b, fi, ni, t and l stand for quantities being introduced in the first column. The columns <u>type</u> and <u>length</u> contain entries referring to the use of the enquiry as an arithmetic primary only; other use is not possible. The last column denotes which category enquiry will deliver the value <u>true</u>.

f	lhs	rhs	el i	type	length	category
constant(t,n,m)	?	?	?			constant f
	n	m	?	t	?	
monadic(t,n,b)	?	b	?			monadic f
	n	b	?	t	?	
dyadic(t,a,b)	a	b	?			dyadic f
	a	b	?	t	?	
polyadic(t,i,l,fi)	/	/	fi			polyadic f
	/	/	fi	t	l	
rowadic(t,i,l,ni)	/	/	?			rowadic f
	/	/	ni	t	l	

From the table, it follows that one has to be careful in using enquiries in a formula expression; if the result of an enquiry is a number it may not directly be used in a formula expression. Instead, this number should be assigned to an integer variable and this integer variable may then be used in the formula expression.

Example: "f:= constant(1,1024,2048); f:= lhs of f + rhs of f" leads to an error message, whereas:

"f:= constant(1,1024,2048); i:= lhs of f; r:= rhs of f;
f:= i + r"

leads to the creation of a formula "1024 + 2048", provided i and r have been declared as integers.

As has been stated already, an arithmetic primary may appear in a formula expression (except if it is a formula enquiry delivering an integer). This is of course very useful as we normally have that the data structures, which we call formula values, contain as terminal nodes integers (real numbers have to be decomposed into integers) and the ABC ALGOL programmer has to be able to describe these integers by means of arithmetic primaries.

That formula primaries may appear in arithmetic or boolean expressions is not so obvious. For, what is the value of f+g, if f and g are both formula variables; their values are of quite another type: binary trees, so that adding f and g is meaningless. There is, however, one application which makes the possibility to have formula primaries in boolean expressions useful. Consider the statement: "_if_ f = g _then_ ... _else_ ...". If f and g both refer to the same object, i.e. are identically the same, occupying the same memory locations, the value of "f = g" is _true_; if this is not the case the value of "f = g" is always _false_.
In almost all reasonable formula manipulating systems one applies the one-and-zero simplifications. This implies that there is only one "one" and only one "zero" in the system. In such cases it is very meaningful to test on "f = one" or "f = zero", instead of the much more elaborate testing on: "if f is a number and, if so, is the value of that number equal to the value of one".
It must be stated explicitly that the test f = g does not invoke a complete investigation on the two tree structures, referred to by f and g, whether they are equivalent.

Concerning the boxes containing /, we remark that polyadic and rowadic structures are represented by means of dyadic and monadic structures. If f is a polyadic(rowadic) structure, then:
length _of_ f = _lhs_ _of_ f
_el_ i _of_ f = _lhs_ _of_ _rhs_ _of_ ... _rhs_ _of_ f,
where the _rhs_ _of_ enquiry has to be performed i times. If f is polyadic, _el_ i _of_ f delivers the reference to a formula, otherwise it is an integer.

A formula primary may also be a string. For example: "f:= ⊀this is a string⊁" leads to a formula value referred to by f, which we may enquire in the following way:

type _of_ f	results in 31
rowadic f	results in _true_
length _of_ f	results in 6

f evidently turns out to have become a rowadic structure consisting of 6 elements. These elements are obtained by grouping the symbols of the string from left to right in pieces of three symbols: r, t, s and computing the integer r × 256 ↑ 2 + t × 256 + s, where r, t and s are the internal representation of the three symbols. This internal representation is the MC resym value increased by one (in order to be able to mark the absence of a symbol). If the number of symbols in the string is not a multiple of 3, one or two "absent symbols" are added in front of the real symbol. in the above case the 6 elements of f are given by:
IR(thi),IR(s i),IR(s a),IR(st),IR(rin),IR(??g)
where IR computes the integer and ? denotes an absent symbol.

2.6. The procedure replace

As mentioned already in section 2.4, it is not possible to assign something to the lhs, rhs or i-th el of a formula variable. This means that a tree structure, once being the value of a formula name can, in principle, not be changed. In this section we shall see that, if one insists, the tree structure can be changed by means of an almost hidden and secret procedure "replace".
Why are we reluctant to give an ABC ALGOL programmer elegant linguistic means to change his tree. In other languages these are normally available.
The reasons are five-fold:
Firstly; if one allows normal changes of tree structures, cyclic structures will be created. With cyclic structures, however, it is absolutely necessary that the ABC ALGOL programmer can set and test bits in his datastructure. For, if e.g. an output routine must be written, how can one be sure that one does not run into infinite loops, otherwise than by marking those parts of the datastructure which have been treated already.
Secondly; we wanted to provide a watertight system of auxiliary procedures in chapter 3 for general use in formula manipulation. It is, however, very hard, if not impossible, to construct such procedures in which a user can change parts of the value of a formula variable. In chapter 3, the formulae are assumed to have a specific form; desastrous effects would result from changing this form a little bit. Circular structures would ruin the functioning of these procedures completely.

Thirdly; for efficiency reasons, called-by-value parameters are treated in a special way (see chapter 4); i.e. a local integer is created and the reference to the value, calculated once, is assigned to this integer. A special marking bit is inspected to see whether it is necessary to create a new name for saving this value. Assignment in the procedure body is prohibited, either directly by the compiler, or later by the run-time system. If assignment to "lhs of rhs of el 5 of f" were allowed, it is difficult to use this marking-bit technique.
Fourthly; in common practice the following program piece:

"f:= x; g:= f + f; f:= y;

leads to: g = x + x and f = y. With assignment to the lhs of f, however, the outcome of:

"f:= x + y; g:= f + f; lhs of f:= z"

would not be g = (x + y) + (x + y) and f = z + y, but g = (z + y) + (z + y) and f = z + y which is contrary to what one might expect from the first example.

A fifth less sound reason for being hesitant with implementing the assignment to lhs of f was, that we could not well judge all the consequences; although, for the compiler, it would not have given much problems.

Forbidding the change of a given datastructure at all would have been too restrictive; therefore, for an ABC ALGOL programmer, knowing thoroughly the details of the system, there has been included a way out. This way is, however, not provided with any error checks. Following this way, one has to be absolutely sure of what one does, as the system does not give any protection.

The run-time system has been provided with a procedure "replace", which in non-existent ABC ALGOL could have been declared by:

```
"procedure replace(f,left,g); value f,left,g;
              formula f,g; Boolean left;
  if left then lhs of f:= g else rhs of f:= g;"
```

If one wants to change the i-th el of f, this may be done by a statement of the form:

```
a:= f; for i:= i step -1 until 1 do a:= rhs of f;
replace(a,true,g)
```

In chapter 3, the replace mechanisme is used in order to make the simplification process more efficient (less inefficient).

2.7. Error messages and end of compilation

The errors can be divided into the following classes:

1. Syntactic errors with an errormessage of the following format:

    xxxxxxxxxxerrorxxxxxxxxxx <ln> <errortext> <su1> <su2>

where <ln> is the line number of the first symbol (including lay-out) of the syntactic unit under consideration which directly caused the error;

where <errortext> describes the type of error, e.g. "nr of param not O.K." or "for list element of type formula" or "id not declared in form expr";

where <su1> is the syntactic unit under consideration;

and where <su2> is the syntactic unit already read but not yet treated.

A syntactic unit is the smallest piece of text which can be read by simple low-level procedures. (finite state automata). The basic symbols, the identifiers and the numbers are syntactic units.

When the errortext is "synt unit not OK", the above error message is followed by:

"synt unit should be: <su3>"

where <su3> specifies the syntactic unit which the compiler required, but which was not there. The reaction of the compiler is to treat the text as if the required syntactic unit had been inserted.

The compiler goes on to treat the text and produces a (hopefully) correct ALGOL 60 program.

2. Semantic errors with an error message of the above format. The errors are now due to overflow of tables, e.g. "inf list too small" or "too much identifiers". The user is advised to compare chapter 8 for the restrictions imposed by the compiler. The compiler stops translating.

3. Run-time errors occurring during the ALGOL 60 compilation of the compiled program, which are due to the fact that the ABC ALGOL compiler does not check the ABC ALGOL program thoroughly. E.g., Booleans are not tested for declaration.

4. Run-time errors detected by the ABC ALGOL run-time system. The error message has the following format:

<errortext>

linenumbers at previous entries:

<l1> <l2> <l3> <l4> <l5> <l6> <l7> <l8> <l9> <l10>.

where <errortext> describes the type of error, e.g. "protection error", or "type in const not appropriate", or "index in el not appropriate". <l1>,...,<l10> give the line numbers of the first symbol of the 10 blocks executed most recently, <l1> denoting the line number of the last block. A procedure body counts for a block.

The further execution is discontinued.

5. Run-time errors detected by the ALGOL 60 system resulting in a discontinuation of the execution and an error message referring to the line of the ALGOL 60 program causing the error and an error number for which we refer to the manual of the ALGOL 60 system [5].

6. Unforeseen other errors during the compilation due to a non-ideal compiler. These errors should be reported to the author.

The compilation of an ABC ALGOL program is finished by printing the time in millihours (mh) needed for the compilation.

2.8. ABC ALGOL compared with other language

Being an extension of ALGOL 60, ABC ALGOL can best be compared with other derivations of ALGOL 60 such as SIMULA [2], ALGOL W [17], ALGOL 68 [18] or Formula ALGOL [15b].

The extensions of ABC ALGOL with respect to ALGOL 60 are twofold:

1. Creation, change and deletion of arbitrary data structures,
2. The possibility to redefine the operators +, -, ×, / etc.

With respect to the second extension, the language may be compared with ALGOL 68, where the common operators and new operators can be defined together with a priority. The priority does not need to be the same for a given operator defined in several blocks. The method of definition, by means of an ordinary routine declaration, resembles very much our method of definition by means of a formula procedure declaration. We, however, have a fixed set of operators with fixed priority.

With respect to dynamic data structures we take a record declaration in ALGOL W as example:

```
"record node(string info,integer col; reference(node) left,right)".
```

In ABC ALGOL we can not prescribe the format of a data structure by means of this form of declarations; we have to declare a formula procedure:

```
"formula procedure node(info,col,left,right);
    formula info,left,right; integer col;
    node:= dyadic(1,info,monadic(2,col,dyadic(3,
           if type of  left ≠ 1 then error else left,
           if type of right ≠ 1 then error else right)));"
```

and each time we need a new node we can call this procedure: "new node:= node(⊀xyz⊁,5,left node,right node)", where "new node" has been declared of type formula.

In ALGOL 68 one can ask for "col of node" or for "left of node". In ABC ALGOL this is done, as in ALGOL W, by: "col(node)" and "left(node)", where "col" and "left" are declared by:

```
"integer procedure col(n);value n; formula n;
 if type of n ≠ 1 then error else col:= lhs of rhs of n;
 formula procedure left(n); value n; formula n;
 if type of n ≠ 1 then error else left:= lhs of rhs of n;"
```

Note that in ALGOL W and ALGOL 68 the "declaration" of both the structure and the "field selectors" is done immediately with the declaration of the structure, whereas in ABC ALGOL this has to be done separately by means of procedure declarations. On the other hand, the ABC ALGOL method is more flexible, in the sense that the value of one formula variable may have several formats, e.g., it may first be the short integer 0, then the string ⊀xyz⊁, then the dyadic structure ⊀xyz⊁ + ⊀abc⊁. This is accomplished by the following statements: "f:= 0; f:= ⊀xyz⊁; f:= f + ⊀abc⊁". In ALGOL 68 this is also possible by means of uniting, a complicated coercion process, however.

Compared with SIMULA, it is evident that ABC ALGOL lacks the parallel process facility and interprocess equipment. The <u>element</u> and <u>set</u> concepts resemble, however, the ABC ALGOL data structures.

Formula ALGOL has been designed primarily for formula manipulation. Compared with ABC ALGOL, we observe that in both languages dynamically changing data structures and operator definitions are possible; in Formula ALGOL in a much more general form, however.

With respect to the implementation we remark that the efficiency of the run-time system really matters; not the efficiency of the compiler. This run-time system consists of a set of 25 simple ALGOL 60 procedures, occupying a few pages only. They manipulate with one integer array. To transcribe these procedures into machine-code is not a hard problem and the result will be a considerable increase of performance. It is planned to do this for the CDC CYBER computer.

In not using dirty machine-code, but remaining at the high ALGOL 60 level for the run-time system as well as for the compiler, we have achieved a highly portable system (an important concept in software engineering) and a very lucid system which may prove to be of educational value.

3. Simplification and rational numbers

In this chapter, a complete system will be given for automatic simplification of special, but frequently occurring, formulae. The system is provided with a subsystem of ABC ALGOL procedures for automatically handling long rational numbers; i.e. quotients of long integers. Long integers are bounded in size by the total available storage capacity only and not by the word size.
The system is a small one, which may be regarded as the core of a bigger system, not described in this tract.

Problems in formula manipulation most frequently involve simplification of simple sums of simple terms; each term being either a simple factor or a product of simple factors; each simple factor being a number, an algebraic variable or an integral power of an algebraic variable. Moreover, quotients occur; but manipulations involving quotients are normally carried out as manipulations of the numerators and denominators being simple sums.
In an earlier formula manipulation system [11], automatic cancelling of greatest common divisors in quotients has been built in, with the effect that ordinary formula manipulation programs run very slowly while the gcd computations were meaningful in one percent of the cases only.
G. ten Velden [16] constructed a system of simplification procedures emphasizing at the most optimal use of memory space at the cost of execution time.

3.1. The representation of the formulae

The key idea of the system of this chapter is to represent the formulae in their simplified form, in which they have the general form:

$$f = \sum_{i=1}^{n} c[i] \prod_{j=1}^{l[i]} a[i,j] \uparrow e[i,j],$$

where $n \geq 1$, $c[i] \neq 0$, except when $n=1$ and $l[1] = 0$, in which case $c[1]$ may be 0, where $l[i] \geq 0$ ($l[i] = 0$ means we deal with a number only) and where the $a[i,j]$'s are algebraic variables and the $e[i,j]$'s are integers ≥ 1.
The ordering of the algebraic variables in a term is determined by some priority number $pr(a[i,j])$, which is a positive integer different for different algebraic variables.
We adopt the notation:

a[i,j] <. a[k,l] if and only if pr(a[i,j]) < pr(a[k,l])

and we say that a[i,j] "comes before" a[k,l].
The ordering in a term is such that:

a[i,j] <. a[i,j + 1], $1 \leq j < l[i]$.

The ordering of the terms is defined as follows:

$$\text{Let} \quad t[i] = c[i] \prod_{j=1}^{l[i]} a[i,j] \uparrow e[i,j]$$

$$\text{and} \quad t[k] = c[k] \prod_{j=1}^{l[k]} a[k,j] \uparrow e[k,j],$$

then the following algorithm determines the ordering:

```
for j:= 1 step 1 until min(l[i],l[k]) do
if a[i,j] <. a[k,j] then goto before else
if ¬ a[i,j] = a[k,j] then goto after else
if e[i,j] > e[k,j] then goto before else
if e[i,j] < e[k,j] then goto after;
if l[i] > l[k] then goto before else
if l[i] < l[k] then goto after;
equal: apart from numerical factors t[i] equals t[k].
before: t[i] comes before t[k] : t[i] <. t[k].
after: t[i] comes after t[k] : t[k] <. t[i].
```

It does not need saying that the ordering of the terms t[i] of f is such that $t[i] <. t[i+1]$, $1 \le i < n$.
Hence, if a single number occurs, it is the last term.
The above ordering will be called canonical ordering.

The system has to apply the simplification process for every operation of the formulae, which is time-consuming. The big advantage is, however, that the simplification process itself can be made fast due to the fact that we can make full profit of the simplified form of the formulae.

The representation of any formula f is as follows:

```
f = polyadic(simple sum,i,n,
          dyadic(0,c[i],dyadic(0,
                              rowadic(0,j,l[i],pr(a[i,j])),
                              rowadic(0,j,l[i],e[i,j]))
         )      ).
```

In the above representation, l[n] may be equal to zero, which obviously means that the last term is a number.
To get pr(a[i,j]), we simply have to ask for:

```
el[j] of lhs of rhs of el i of f
```

or, if one wants to go along all e[i,j]'s consecutively, one may perform the following algorithm:

```
p:= f; if length of f < i then goto error;
for k:= 1 step 1 until i do p:= rhs of p;
p:= rhs of rhs of lhs of p;
if length of p < j then goto error;
for k:= 1 step 1 until j do
begin p:= rhs of p;e[i,j]:= lhs of p end.
```

A consequence of the fixed chosen format is that a simple algebraic variable or a single number needs a lot of storage space. This is, however, no problem at all since the system is made for situations in which there occur very "big" formulae. Only initially, when the numbers and variables are treated occurring in the expressions, this storage space is wasted. As soon as they are used, however, as building stones in the "big" formulae, they do not occupy any storage space due to the automatic garbage collection.
As one may have observed already, it is not the algebraic variable a[i,j] itself which is used in the representation of the formula f, but pr(a[i,j]) instead. This is for efficiency reasons as the integer pr(a[i,j]) is used very frequently, whereas the algebraic variable itself is used only when output is needed. The output for a[i,j] is defined as a string and is the k-th element of the inverted list "stringstring", k being equal to pr(a[i,j]).

Concerning quotients we remark that it is possible that a formula f is a quotient but then the numerator and denominator are simple sums; therefore, transformations are performed which transform a quotient of quotients, a product of quotients and a sum of quotients into one quotient, of two simple sums. The latter quotients are called simple quotients.

3.2. How to use the system

After declaration of the standard procedures (if they are not yet put in the library of standard procedures), one can write an ordinary ABC ALGOL program, according to the following conventions:

1. Algebraic variables have to be introduced by means of the procedure "av", with as actual parameter a string being used for the output of the algebraic variable.
2. Output must be produced by a call of "output" with as actual parameter the formula to be output.
3. Use of the ordinary arithmetic operators leads to automatically simplified formulae represented in a form described in section 3.1.

4. The system knows three types of numbers: short integers, being integers in absolute value less than the number base "radix", long integers, with a magnitude not restricted by the word length, and rational numbers, being pairs of two common-divisor-free integers, of which the second one is greater than one.
The representation is as follows:

short integer s: constant(short integer,s,0),

long integer l: rowadic(long integer,i,n,d[i]),

with

$$l = \sum_{i=1}^{n} d[i] \times radix \uparrow (i-1),$$

sign(d[i]) = sign(l), except when d[i] equals 0, and abs(d[i]) < radix

rational number r: dyadic(rat numb,num,den),

with num and den two common-divisor-free long or short integers of which den is greater than one.

The radix has been chosen equal to 10 ↑ 6, but this can easily be changed. The only requirement for the radix is that the basic arithmetic of the machine can multiply and add two integers, in absolute value less than the radix, correctly. For the EL X8 the floating-point arithmetic handles the multiplication and the addition of two integers correctly if the result is less than 2 ↑ 40; therefore radix equals approximately 2 ↑ 20. It is emphasized that ordinary integers for the EL X8 are bounded in size by 2 ↑ 26; it is, therefore, possible that an ordinary integer is treated by the system as a long integer.
Occurrence of an integer in an expression (e.g. -1234567) leads to the introduction of a simple sum consisting of one simple term being the product of this integer and an empty list of algebraic variables. (For the example:

```
polyadic(simple sum,i,1,dyadic(0,
   rowadic(long integer,j,2,if j = 1 then -234567 else -1),
   dyadic(0,rowadic(0,j,0,0),rowadic(0,j,0,0)))).)
```

Using the arithmetic operators +, -, ×, / and ↑, the latter one with a (small) integer as exponent, one can operate with the numbers without any troubles. As the ABC ALGOL compiler has no knowledge of long integers or rational numbers, it does not transfer:
a = 123456 789101 112131 415161 718192/122232 425262 728293

into a rational number. This should be done as follows:

```
r:= 1000 000;
a:= ((((123456 × r+789101) × r+112131) × r+415161) × r+718192)/
      ((122232 × r+425262) × r+728293);
```

If the only use of the system is to manipulate long numbers, the overhead in the form of simple sums and empty lists may become awkward and too inefficient. It is then possible to operate directly on the numbers by means of the procedure:

```
    formula procedure oper on num(oper,a,b)
```

where oper specifies the operation and a and b specify the operands. One is advised to consult section 3.4 for further details.

3.3. The procedures of the system

```
begin
  integer simple sum,simple quotient,sum,product,quotient,
          short integer,long integer,rat numb,radix,avcntr,i,j;
  formula zero1 = constant(0,0,0),one1 = constant(0,1,0),
          minone1 = constant(0,-1,0),
          stringstring:= constant(0,0,0),
          dummy variable = dyadic(0,rowadic(0,j,0,0),rowadic(0,j,0,0)),
          zero = polyadic(3,i,1,dyadic(0,zero1,dummy variable)),
          one  = polyadic(3,i,1,dyadic(0,one1,dummy variable));

  avcntr:= short integer:= 0; long integer:= 1; rat numb:= 2;
  radix:= 1000 000; simple sum:= 3; simple quotient:= 4;
  sum:= 5; product:= 6; quotient:= 7;

begin
  formula procedure av(s); value s; formula s;
  begin integer i,j; avcntr:= avcntr + 1;
    stringstring:= dyadic(0,s,stringstring);
    av:= polyadic(simple sum,i,1,dyadic(0,one1,dyadic(0,
                 rowadic(0,j,1,avcntr),
                 rowadic(0,j,1,1))))
  end av;
```

```
formula procedure find string(priority); value priority;
integer priority;
begin integer i; formula ptr; ptr:= stringstring;
  for i:= avcntr-1 step -1 until priority do ptr:= rhs of ptr;
  find string:= lhs of ptr
end find string;

formula procedure constant+(n); value n; integer n;
begin integer i,j;
  constant+:= if n = 0 then zero else if n = 1 then one else
  polyadic(simple sum,i,1,dyadic(0,
          if abs(n) < radix then (if n = -1 then minone1 else
          constant(short integer, n, 0)) else
          rowadic(long integer,j,2,
                 if j = 1 then n - n : radix × radix
                        else n : radix),
          dummy variable))
end constant+;

integer procedure comp terms(a,b); value a,b; formula a,b;
comment comp terms delivers 1 if a <. b, 0 if a = b and
-1 if b <. a, a and b refer to dyadic(0,rowadic(...),rowadic(...));
begin formula avptra,avptrb,expptra,expptrb;
  integer aip,akp,eip,ekp,la,lb,p,min;
  avptra:= lhs of a; avptrb:= lhs of b;
  expptra:= rhs of a; expptrb:= rhs of b;
  la:= length of avptra; lb:= length of avptrb;
  min:= if la < lb then la else lb;
  for p:= 1 step 1 until min do
  begin avptra:= rhs of avptra; avptrb:= rhs of avptrb;
    aip:= lhs of avptra; akp:= lhs of avptrb;
    if aip < akp then goto before else
    if aip ≠ akp then goto after else
    begin expptra:= rhs of expptra; expptrb:= rhs of expptrb;
      eip:= lhs of expptra; ekp:= lhs of expptrb;
      if eip > ekp then goto before else
      if eip ≠ ekp then goto after
  end end;
```

```
  if la > lb then goto before else if lb > la then goto after;
equal:  comp terms:= 0; goto out;
before: comp terms:= 1; goto out;
after:  comp terms:= -1;
out: end comp terms;

comment The next two procedures  serve to put a list of formulae  and a list
of integers, in reversed order, in a polyadic or rowadic structure;

formula procedure reverse formula(f); formula f;
begin reverse formula := lhs of f; f := rhs of f end;

integer procedure reverse integer(f); formula f;
begin reverse integer := lhs of f; f := rhs of f end;

comment The above procedures are applied as follows:
        polyadic(...,i,n,reverse formula(f))
or      rowadic (...,i,n,reverse integer(f)).
Another auxiliary  procedure  is append,  to reverse  a list of the first  n
formulae or integers given by g, into another list and append it to f;

procedure append(f,g,n,dy); value n,dy; formula f,g; integer n;
boolean dy;
if n > 0 then
begin again: f:= if dy then dyadic(0,lhs of g,f) else
                            monadic(0,lhs of g,f);
  if n > 1 then begin n:= n-1; g:= rhs of g; goto again end
end append;

formula procedure dyadic+(a,b); value a,b; formula a,b;
if a = zero then dyadic+:= b else if b = zero then dyadic+:= a else
if type of a = simple quotient then
dyadic+:= if type of b = simple quotient then
      (lhs of a × rhs of b + lhs of b × rhs of a)/
      (rhs of a × rhs of b) else
      (lhs of a + b × rhs of a)/rhs of a else
if type of b = simple quotient then
dyadic+:= (a × rhs of b + lhs of b)/rhs of b else
```

```
begin integer lf,la,lb,v; boolean nexta,nextb;
  formula ta,tb,pa,pb,f,co;
  la:= length of a; lb:= length of b; nexta:= nextb:= true;
  f:= zero; lf:= 0; pa:= a; pb:= b;
  again: if la = 0 ^ nexta then
  begin if nextb then
    begin if lb ≠ 0 then pb := rhs of pb end
    else lb := lb + 1;
    lf := lf + lb; append(f, pb, lb, true)
  end else
  if lb = 0 ^ nextb then
  begin if nexta then pa := rhs of pa else la := la + 1;
    lf := lf + la; append(f, pa, la, true)
  end else
  bein if nexta then
    begin la:= la - 1; pa:= rhs of pa; ta:= lhs of pa; nexta:= false
    end; if nextb then
    begin lb:= lb - 1; pb:= rhs of pb; tb:= lhs of pb; nextb:= false
    end; v:= comp terms(rhs of ta, rhs of tb);
    if v = 1 then begin f:= dyadic(0,ta,f); nexta:= true end else
    if v = -1 then begin f:= dyadic(0,tb,f); nextb:= true end else
    begin co := oper on num(sum, lhs of ta, lhs of tb);
      if co ≠ zero1 then
      f := dyadic(0, dyadic(0, co, rhs of ta), f)
      else lf := lf - 1;
      nexta:= nextb:= true
    end; lf:= lf + 1; goto again
  end;
  dyadic+:= find number(lf,lhs of f,reverse formula(f))
end dyadic+;

formula procedure find number(k,ft,ot); value k;integer k;
        formula ft,ot;
begin integer i; formula f;
  if k = 0 then begin find number := zero; goto out end else
  if k = 1 then
  begin f := ft; if rhs of f = dummy variable then
    begin f:= lhs of f;
```

```
      if f = zero1 then find number:= zero else
      if f =  one1 then find number:= one else
      find number:= polyadic(simple sum,i,1,
      dyadic(0,f,dummy variable));
      goto out
  end end;
  find number:= polyadic(simple sum,i,k,ot);
out: end find number;
```

comment In the above procedure, we profit from the fact that simple sums are canonically ordered. In the same way we define the following procedure for an elementary multiplication of two terms a and b;

```
formula procedure elt mult(a,b); value a,b; formula a,b;
begin integer i,j,lf; integer array l,vi,ei[1:2];
  boolean array next[1:2];
  formula ef,vf; formula array v,e[1:2];
  v[1]:= rhs of a; v[2]:= rhs of b;
  for i:= 1,2 do
  begin e[i]:= rhs of v[i]; v[i]:= lhs of v[i];
    l[i]:= length of v[i]; next[i]:= true
  end;
  lf:= 0; vf:= ef:= zero;
again:for i:= 1,2 do
  if l[i] = 0 ^ next[i] then
  begin j := 3 - i; if next[j] then
    begin if l[j] ≠ 0 then
      begin v[j] := rhs of v[j]; e[j] := rhs of e[j] end
    end else l[j] := l[j] + 1;
    lf := lf + l[j];
    append(vf,v[j],l[j],false); append(ef,e[j],l[j],false);
    goto out
  end;
  for i := 1, 2 do if next[i] then
  begin v[i]:= rhs of v[i]; e[i]:= rhs of e[i]; l[i]:= l[i] - 1;
    vi[i]:= lhs of v[i]; ei[i]:= lhs of e[i]; next[i]:= false
  end;
```

```
  if vi[1] < vi[2] then i:= 1 else if vi[1] > vi[2] then i:= 2 else
  begin i:= 1; ei[1]:= ei[1] + ei[2];
    next[2] := true
  end;
  lf:= lf + 1; vf:= monadic(0,vi[i],vf); ef:= monadic(0,ei[i],ef);
  next[i] := true; goto again;
out: elt mult:= dyadic(0,oper on num(product,lhs of a,lhs of b),
                if lf= 0 then dummy variable else
                dyadic(0,rowadic(0,i,lf,reverse integer(vf)),
                         rowadic(0,i,lf,reverse integer(ef))))
end elt mult;

formula procedure dyadic×(a,b); value a,b; formula a,b;
if a = zero v b = zero then dyadic×:= zero else
if a = one then dyadic×:= b else
if b = one then dyadic×:= a else
if type of a = simple quotient then
    dyadic×:= if type of b = simple quotient then
    (lhs of a × lhs of b)/(rhs of a × rhs of b) else
    (lhs of a × b)/rhs of a else
if type of b = simple quotient then
    dyadic×:= (a × lhs of b)/rhs of b else
begin comment
```

The next algorithm describes the process to form the product of two simple sums and to make one simple sum of it in a highly efficient way. The key idea is again to use the fact that the terms of the simple sums are ordered canonically.

Let

$$a = \sum_{i=1}^{n} ta[i], \quad b = \sum_{j=1}^{m} tb[j]$$

and

$$f = a \times b = \sum_{i=1}^{n} \sum_{j=1}^{m} ta[i] \times tb[j]$$

We consider each term t = ta[i] × tb[j] as to have attached to it two numbers i and j, called the location of t. t is member of one of the following three sets:

N: the set of all not treated terms,

P: the set of all partially treated terms,

C: the set of all completely treated terms.

The algorithm consists of removal of all terms, in a certain order, from N via P to C in such a way that the terms are consecutively put into C in their final canonical ordering. P serves as a reservoir of terms which are candidates for removal to C.

Initially P consists of all terms with location (i,1),i = 1, ...,n, C is empty and N contains all other terms.

Until P is empty, the algorithm repeatedly performs the following operations: the first element of P is moved from P to C. Let the location of this element be(i,j), then, if j < m, the term ta[i] × tb[j+1] is moved from N to P.

The first element of P, defined as the element preceding all other elements of P in canonical ordering, has the interesting property that it not only precedes the other elements of P but also those of N. We will show this by proving that the following property, called the predecessor property, holds:

For every term t, with location (p,q), in N, there exists
a term s in P with location (p,k) such that s <.t.

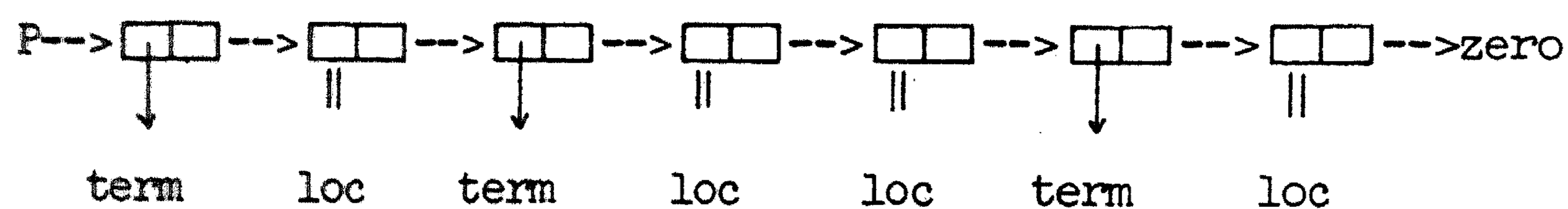
The term s is called the predecessor of t.

From t = ta[p] × tb[k] and s = ta[p] × tb[q] it follows that k < q since tb[k] <. tb[q]. Moreover, from the predecessor property it follows directly that the first element of P has the desired property.

The predecessor property holds initially. We, therefore, have to prove that it is invariant during the course of the algorithm.

Consider the first term f with location (i,j) in P, Remove it from P to C and, if j < m, remove s = ta[i] × tb[j+1] from N to P. This gives the sets P1,C1 and N1. Assuming that the predecessor property holds for P and N, we shall show that it holds for P1 and N1. Take an arbitrary term t from N1. Let pt be the predecessor of t in P. If pt is identical to f we know that the location of t must be (i,k). From f <. t, and f <. s it now follows s <. t, since the value of k must exceed the value of j+1 (from s ≠ t). Hence s is the predecessor of t in P1. If pt is not identical to f the predecessor of t in P is still in P1. This proves the invariance.

From the predecessor property it follows that, when P is empty, N is empty so that C contains all terms. We remind that each time a term is moved to C it precedes all terms which are not in C.
C is implemented as a linear list. The ordering of the list elements is the reversed canonical ordering. P is implemented as a linear list of terms and locations. The location elements follow directly the elements of their corresponding terms. The ordering of the term elements is the canonical ordering. Observe that more than one location may follow a term due to the possibility that two terms can be combined into one. The structure of P is as follows:



The cells, pointing to terms, are dyadic structures and the cells, containing the locations, are monadic structures.;

```
integer i,j,n,m,k,loc,ct; formula a1,b1,p,c,t,tt,f,g;
n:= length of a; m:= length of b;
if n > m then begin i:= n; n:= m; m:= i; a1:= b; b1:= a end
else begin a1:= a; b1:= b end;
if n = 1 then
begin f:= lhs of rhs of a1; g:= b1; c:= zero; k:= m;
  for j:= 1 step 1 until m do
  begin g:= rhs of g; c:= dyadic(0,elt mult(f,lhs of g),c) end
end else
begin formula array ta[1:n],tb[1:m];
  t:= a1; for i:= 1 step 1 until n do
  begin t:= rhs of T; ta[i]:= lhs of t end;
  t:= b1; for j:= 1 step 1 until m do
  begin t:= rhs of t; tb[j]:= lhs of t end;
  c:= p:= zero; k:= 0;
  for i:= n step -1 until 1 do
  p:= dyadic(0,elt mult(ta[i],tb[1]),
      monadic(0,i × 1000 + 1,p));
  for i:= i while p ≠ zero do
  begin if dyadic p then
```

```
    begin t:= lhs of p; if lhs of t ≠ zero1 then
      begin c:= dyadic(0,t,c); k:= k + 1 end;
      p:= rhs of p
    end;
    loc:= lhs of p; i:= loc ÷ 1000; j:= loc - i × 1000;
    if j < m then
    begin t:= elt mult(ta[i],tb[j+1]); tt:= rhs of t;
      f:= rhs of p; g:= p;
      for i:= i while f ≠ zero do
      begin if dyadic f then
        begin ct:= comp terms(rhs of lhs of f,tt);
          if ct = 0 then
          begin replace(g,false,
            dyadic(0,dyadic(0,oper on num(sum,
                   lhs of t,lhs of lhs of f),tt),
                   monadic(0,i×1000 + j+1,rhs of f)));
            goto out
          end else
          if ct = -1 then goto repl
        end;
        g:= f; f:= rhs of f
      end;
    repl:replace(g,false,dyadic(0,t,monadic(0,i×1000 + j+1,f)))
    end;
  out: p:= rhs of p
end end;
dyadic×:= find number(k,lhs of c,reverse formula(c))
end dyadic×;

formula procedure dyadic/(a,b); value a,b; formula a,b;
begin if type of b = simple sum then
  begin if length of b = 1 then
    begin if rhs of el 1 of b = dummy variable then
      begin integer i;
        dyadic/:= polyadic(simple sum,i,1,dyadic(0,
        oper on num(quotient,one1,lhs of el 1 of b),
        dummy variable)) × a;
        goto out
```

```
  end end end;
  dyadic/:= if a = zero then zero else
         if type of a = simple quotient then
        (if type of b = simple quotient then
        (lhs of a × rhs of b)/(rhs of a × lhs of b) else
         lhs of a/(rhs of a × b)) else
         if type of b = simple quotient then
                       (a × rhs of b)/lhs of b
         else dyadic(simple quotient,a,b);
out:
end dyadic/;

formula procedure monadic+(a); value a; formula a; monadic+:= a;

formula procedure monadic-(a); value a; formula a;
begin integer i; i:= -1; monadic-:= i × a end;

formula procedure dyadic-(a,b); value a,b; formula a,b;
dyadic-:= a + (-b);

formula procedure monadic↑(a,n); value a,n;
formula a; integer n;
monadic↑:= int pow(a,n);

formula procedure dyadic↑(a,n); value a,n; formula a,n;
begin if type of n = simple sum then
  begin if length of n = 1 then
    begin if rhs of el 1 of n = dummy variable then
      begin if type of lhs of el 1 of n = short integer then
        begin dyadic↑:= int pow(a,lhs of lhs of el 1 of n);
          goto out
  end end end end;
  error(⊰n in exponentiation not of integral type⊱);
ut:
d dyadic↑;

 mula procedure int pow(a,n); value a,n; formula a; integer n;
 n = 0 then int pow:= one else if n = 1 then int pow:= a else
```

```
if n < 0 then int pow := one / int pow(a, -n) else
if type of a = simple quotient then
    int pow:= int pow(lhs of a,n)/int pow(rhs of a,n) else
if a = zero then int pow:= zero else
if length of a = 1 then
begin integer i,j,l; formula b = el 1 of a,c = rhs of rhs of b;
  l := length of c;
  int pow:= polyadic(simple sum,i,1,dyadic(0,
            int pow num(lhs of b,n),
            if l = 0 then dummy variable else
            dyadic(0,lhs of rhs of b,
            rowadic(0,j,l,el j of c × n))))
end else
begin  formula b; b:= int pow(a,n : 2);
  int pow:= if n = n : 2 × 2 then b × b else a × b × b
end int pow;

formula procedure int pow num(a,n); value a,n;
formula a; integer n;
if n = 0 then int pow num:= one1 else
if n = 1 then int pow num:= a else
begin formula b; b:= int pow num(a,n : 2);
  int pow num:= if n = n : 2 × 2 then
  oper on num(product,b,b) else
  oper on num(product,a,oper on num(product,b,b))
end int pow num;

procedure output(f); value f; formula f;
if f = zero then printtext(⸢ 0 ⸣) else
if f =  one then printtext(⸢ 1 ⸣) else
if type of f = simple quotient then
begin printtext(⸢(⸣); output(lhs of f);
  printtext(⸢) / (⸣); output(rhs of f); printtext(⸢)⸣)
end else
begin integer l,l1,i,j,k,m;
  formula coeff,pointer1,pointer2,pointer3;
  l:= length of f;
  for i:= 1 step 1 until l do
```

```
  begin pointer1:= el i of f;
    coeff:= lhs of pointer1;
    pointer2:= rhs of rhs of pointer1;
    pointer1:= lhs of rhs of pointer1;
    l1:= length of pointer1;
    if l1 = 0 then outputnumber(coeff) else
    begin if coeff = one1 then printtext(⊰ + ⊱) else
      if coeff =  minone1 then printtext(⊰ - ⊱) else
      begin outputnumber(coeff); printtext(⊰×⊱) end;
      for j:= 1 step 1 until l1 do
      begin if j ≠ 1 then printtext(⊰×⊱);
        pointer3:= find string(el j of pointer1);
        k:= length of pointer3;
        for m:= 1 step 1 until k do
        prchar(el m of pointer3);
        m:= el j of pointer2;
        if m ≠ 1 then
        begin printtext(⊰↑⊱); prnum(m) end
      end
    end
  end
end output;

procedure prchar(w); value w; integer w;
if w ≠ 0 then
begin prchar(w : 256); prsym(w - w : 256 × 256 - 1) end;

procedure prnum(n); value n; integer n;
begin if n > 9 then prnum(n : 10);
  prsym(n - n : 10 × 10)
end;

integer procedure error(s); string s;
begin nlcr; printtext(s); if length of constant(0,0,0) = 1 then ;
  error:= 1
end;
```

comment The above system of procedures is complete but for the procedures "oper on num" and "output num", to be described in the next section. In section 3.6 we give an alternative for the simplification procedures.

3.4. The rational-number system

The system knows two types of numbers:
1. the rational number r represented as:
   dyadic(rat numb,num,denom),
   where num and denom are integers representing the common-divisor-free numerator and denominator (> 1), and
2. the integer n either in the form of a
2.1. short integer represented as:
   constant(short integer,n,0)
   or as a
2.2. long integer represented as:
   rowadic(long integer,i,l,d[i]),
   where d[i] is the i - th component in:

$$n = \sum_{i=1}^{l} d[i] \times radix \uparrow (i-1).$$

It is assumed that abs(d[i]) < radix and sign(d[i]) = sign(n).

The procedures to be described are ordered from bottom to top. In order to understand their working we give the following top to bottom description. It should be emphasized that the theory is given in de Roever [13] and Knuth [7].
.oper on num(oper,a,b) transfers control to oper on num1.
.oper on num1(oper,a,b) computes the desired arithmetic function by means of: rnsum, rnprod, rinv, isum and iprod.
.rnsum and rnprod compute the sum and the product of two rationals, using: isum, iprod, iqr, igcd and st rat.
.rinv computes the inverse of a rational number, using isign and invert.

.isum, iprod and iqr compute the sum, product and quotient of two long integers, using: add, mult, iqrs, idif, isign, iabs, signdif, len, sint, lint and get array.
.igcd computes the greatest common divisor of two long integers, using isum, iprod, iqr, igcds,signdif, sint, lint, len and el.
.igcds computes the gcd of two short integers, using iqrs, sint, iabs and signdif.
.add and mult compute the sum and the product of two long integers using the available basic integer arithmetic.
.iqrs computes the quotient and corresponding remainder of a long integer and a short one, using invert, isign, sint, lint, get array and len.
.idif computes the difference of two long integers, using isum and invert.
.isign computes the sign of a long or short integer.
.iabs computes the absolute value of a long or short integer, using invert.
.signdif computes the sign of the difference of two long or short integers, using isign, len and el.
.invert computes the negative of a long or short integer, using sint, lint and get array.
.st rat, lint and sint store a rational, a long integer and a short integer, resp.
.el(l,f) delivers the l-th element of the long or short integer f.
.len(f) delivers the number of elements into which the long or short integer f has been decomposed with respect to the given base radix. Leading zero's are not taken into account.
.get array(f,i,low,up,ai) fills the elements ai of an integer array with the elements of the long or short integer f in reverse order, using Jensen's device.

Next, we have the following procedures for the output:
.outputnumber prints a number, using isign and outputint.
.outputint prints an integer, using prnum and zeroos.
.prnum prints a small integer.
.zeroos prints leading zero's.

Finally, we have the following procedures for inputting a long integer: lo int and li to be used as follows:

The long integer a = 435512 642890 000000 010123 is introduced in the system by means of the statement:

  a:= lo int(4,li(li(li(li(0,435512),642890),0),10123)).

The sign convention asks that all integers have the same sign. Therefore, b = -a has to be introduced as follows:

  b:= lo int(4,li(li(li(li(0,-435512),-642890),0),-10123)).

By writing a as:

  a:= (435512 × 1000000 + 642890) × 1000000 + 10123

the result would not be a long integer but a simple sum consisting of one term being the product of the desired integer and an empty list of algebraic variables. Hence, writing: "a:= lhs of el 1 of a" would have the desired effect.
Observe that "a:= 435512 642890 000000 010123" would invoke a run-time error caused by integer overflow. The run-time system tries to assign this long integer to an ordinary (small) one. As has been said already, the compiler does not do the transfer to long integers.;

```
integer teller;

formula procedure sint(i); value i; integer i;
   sint := if i = 0 then zero1 else
      if i = 1 then one1 else
      if i = -1 then minone1 else
      constant(short integer, i, 0);

formula procedure lint(i, length, ai); value length; integer i, length, ai;
begin boolean b;
   b := true; i := length + 1;
   for i := i - 1 while i ≥ 1 ^ b do if ai = 0 then
   length := length - 1 else b := false;
   i := 1;
   lint := if length > 1 then rowadic (long integer,
   i, length, ai) else
   if length = 1 then sint(ai) else zero1
end;

formula procedure lo int(length,value); value length;
```

```
           integer length, value;
lo int :=  lint(teller, length, value);

integer procedure li(left, right); value right;
           integer left, right;
if teller = 1 then li := right else
begin teller := teller - 1; li := left; teller := teller + 1
end;

integer procedure len(f); value f; formula f;
len := if constant f then 1 else length of f;

integer procedure el(l, f); value l, f; formula f; integer l;
el := if constant f then
           (if l ≠ 1 then 0 else lhs of f) else
           (if l > 0 ^ l ≤ length of f then
              el l of f else 0);

procedure get array(f, i, ow, up, ai); value f, low, up;
           formula f; integer i, low, up, ai;
if constant f then
begin for i := low step 1 until up do
      ai := if i = low then lhs of f else 0
end else
begin formula pointer; integer l, k;
   pointer := f; l := length of f;
   k := if up - low + 1 < l then up else l + low - 1;
   for i := low step 1 until k do
   begin pointer := rhs of pointer; ai := lhs of pointer
   end;
   for i := k + 1 step 1 until up do ai = 0
end;

formula procedure iabs(a); value a; formula a;
iabs := if isign(a) < 0 then invert(a) else a;

integer procedure isign(a); value a; formula a;
isign := if constant a then sign(lhs of a)
```

```
   else sign(el length of a of a);

formula procedure invert(a); value a; formula a;
if constant a then invert := sint(-lhs of a) else
begin integer l, i;
   l := length of a;
   begin integer array b[1:l];
      get array(a, i, 1, l, b[i]);
      invert := lint(i, l, -b[i])
   end
end;

integer procedure signdif(a, b); value a, b; formula a, b;
begin integer la, lb;
   la := len(a); lb := len(b);
   if la > lb then signdif := isign(a) else
   if la < lb then signdif := -isign(b) else
   begin aa: la := sign(el(lb, a) - el(lb, b));
      if la = 0 ^ lb > 1 then
      begin lb := lb - 1; goto aa end;
      signdif:= la
   end
end;

formula procedure isum(a, b); value a, b; formula a, b;
begin integer la, lb, k, i;
   la := len(a); lb := len(b);
   k := if la > lb then la + 1 else lb + 1;
   if k = 2 then
   begin k := lhs of a + lhs of b;
      isum := if abs(k) < radix then sint(k) else
            lint(i, 2, if i = 1 then k - k : radix × radix
                        else k : radix)
   end else
   begin integer array p, q[1:k];
      get array(a, i, 1, k, p[i]);
      get array(b, i, 1, k, q[i]);
      add(p, q, k);
```

```
      isum := lint(i, k, p[i])
   end
end;

procedure add(a, b, k); value k; integer k; integer array a, b;
begin integer s, t, w, carry;
   for w := a[k] + b[k] while w = 0 ^ k > 1 do
   begin a[k] := 0; k := k - 1 end;
   s := sign(w); carry := 0;
   for t := 1 step 1 until k do
   begin w := a[t] + b[t] + carry;
      if s × w < 0 then
      begin a[t] := w + s × radix; carry := -s end else
      if abs(w) ≥ radix then
      begin a[t] := w - s × radix; carry := s end else
      begin a[t] := w; carry := 0 end
   end;
   if carry ≠ 0 then a[k + 1] := carry
end;

formula procedure idif(a, b); value a, b; formula a, b;
idif := isum(a, invert(b));

formula procedure iprod(a, b); value a, b; formula a, b;
begin integer la, lb, l, i;
   la := len(a); lb := len(b); l := la + lb;
   if l = 2 then
   begin real u;
      u := lhs of a × lhs of b;
      if abs(u) ≥ radix then
      begin l := entier(abs(u) / radix) × sign(u); iprod :=
         lint(i, 2, if i = 2 then l else u - l × radix)
      end else iprod := sint(u)
   end else
   begin integer array aa[1:la], bb[1:lb], cc[1:l];
      get array(a, i, 1, la, aa[i]); get array(b, i, 1, lb, bb[i]);
      mult(aa, la, bb, lb, cc, l);
      iprod := lint(i, l, cc[i])
```

```
   end
end;

procedure mult(aa, la, bb, lb, cc, lc); value la, lb, lc; integer la, lb,
            lc; integer array aa, bb, cc;
begin integer ta, tb, tab, carry, btb; real u;
  for ta := 1 step 1 until la do cc[ta] := 0;
   for tb := 1 step 1 until lb do
   begin carry := 0; btb := bb[tb];
      for ta := 1 step 1 until la do
      begin tab := ta + tb - 1; u := btb × aa[ta] + cc[tab] + carry;
         carry := entier(abs(u) / radix) × sign(u);
         cc[tab] := u - carry × radix
      end;
      cc[tb + la] := carry
   end
end;

formula procedure iqrs(x, y, r); value x, y; formula x, y, r;
if x = zero1 v y = zero1 v y = one1 v y = minone1 then
begin if y = zero1 then r := x else r := zero1;
   iqrs := if y = zero1 then zero1 else
         if y = minone1 then invert(x) else x
end else
begin integer lx;
   lx := len(x); if lx = 1 then
   begin integer u, v, w;
      u := lhs of x; v := lhs of y; w := u ÷ v;
      r := sint(u - w × v); iqrs := sint(w)
   end else
   begin integer s, y1, i, n; integer array xx[1:lx], rr[0:lx]; real m;
      s := isign(x);
      rr[0] := rr[lx] := 0;
      get array(if s > 0 then x else invert(x), i, 1, lx, xx[i]);
      y1 := lhs of y; s := sign(y1) × s; y1 := abs(y1);
      for i := lx step -1 until 1 do
      begin m := rr[i] × radix + xx[i];
         xx[i] := n := entier(m / y1);
```

```
        rr[i - 1] := m - (n × y1)
      end;
      r := sint(s × rr[0]);
      iqrs := lint(i, lx, s × xx[i])
   end
end;

formula procedure iqr(x, y, r); value x, y; formula x, y, r;
begin integer lx, ly, lq;
   lx := len(x) + 1; ly := len(y); lq := lx - ly;
   if x = zero1 ∨ y = zero1 ∨ lx ≤ ly then
   begin r := x; iqr := zero1;
      if y = zero1 then error(⟨y equals zero in iqr⟩)
   end else
   if ly = 1 then iqr := iqrs(x, y, r) else
   begin integer s, i, j, yyly, lb;
      formula normfactor, vradixmin1, q, q1, heady, vyyly, dummy, y1;
      integer array xx[1:lx], yy[1:ly], qq[1:lq];
      s := isign(x) × isign(y);
      yyly := abs(el ly of y);
      normfactor := sint(if yyly × (radix ÷ (2 × yyly)) = radix ÷ 2
            then radix ÷ (2 × yyly)
            else radix ÷ (2 × yyly) + 1);
      get array(iabs(iprod(x, normfactor)), i, 1, lx, xx[i]);
      y1 := iabs(iprod(y, normfactor));
      get array(y1, i, 1, ly, yy[i]);
      vradixmin1 := sint(radix - 1); yyly := yy[ly];
      heady := lint(i, 2, if i = 1 then yy[ly - 1] else yyly);
      vyyly := sint(yyly); lb := ly + 1;
      for j := lx step -1 until lb do
      begin q := if xx[j] ≥ yyly then vradixmin1 else
            iqrs(lint(i, 2, xx[j - 2 + i]), vyyly, dummy);
l:       if signdif(lint(i, 3, xx[j - 3 + i]), iprod(q, heady)) = -1 then
         begin q := isum(q, minone1); goto l end;
         q1 := idif(lint(i, lb, xx[j - lb + i]), iprod(q, y1));
         if isign(q1) = -1 then
         begin q := isum(q, minone1); q1 := isum(q1, y1)end;
         get array(q1, i, j - ly, j, xx[i]);
```

```
            qq[j - ly] := lhs of q
         end;
         r := iqrs(lint(i, ly, s × xx[i]), normfactor, dummy);
         iqr := lint(i, lq, s × qq[i])
      end
   end;

formula procedure igcds(x, y); value x, y; formula x, y;
begin integer procedure gcd(a, b); value a, b; integer a, b;
   gcd := if b = 0 then abs(a) else gcd(b, a - ((a ÷ b) × b));
   formula r, x1, y1;
   x1 := iabs(x); y1 := iabs(y);
   if signdif(x1, y1) < 0 then
   begin r := x1; x1 := y1; y1 := r end;
   iqrs(x1, y1, r);
   igcds := sint(gcd(lhs of y1, lhs of r))
end;

formula procedure igcd(x, y); value x, y; formula x, y;
begin formula u, v; integer lu, lv;
   u := iabs(x); v := iabs(y);
   lu := len(u); lv := len(v);
   if lu = 1 ∨ lv = 1 then igcd := igcds(u, v) else
   begin formula hulp, normfactor;
      integer sd, u1, u2, v1, v2, t, i, a, b, c, d, q;
      sd := signdif(u, v);
      if sd = 0 then
      begin igcd := u; goto eindigcd end;
      if sd < 0 then
      begin hulp := u; u := v; v := hulp;
         a := lu; lu := lv; lv := a
      end;
loop: if v = zero1 then igcd := u else
      if lv = 1 then igcd := igcds(u, v) else
      begin a := d := 1; b := c := 0;
         if lu - lv ≤ 1 then
         begin u1 := el(lu - 1, u); u2 := el(lu, u);
            if lu - lv = 1 then
```

```
            begin v1 := el(lv, v); v2 := 0 end else
            begin v1 := el(lv - 1, v); v2 := el(lv, v) end;
            normfactor := sint(
               if u2 × (radix : (2 × u2)) = radix : 2 then
               radix : (2 × u2) else
               (radix : (2 × u2) + 1));
            u2 := el(2, iprod(normfactor, lint(
               i, 2, if i = 1 then u1 else u2)));
            v2 := el(2, iprod(normfactor, lint(
               i, 2, if i = 1 then v1 else v2)));
5:          if v2 + c = 0 v v2 + d = 0 then goto 7;
            q := (u2 + a) : (v2 + c);
            if q ≠ (u2 + b) : (v2 + d) then goto 7;
            t := a - (q × c); a := c; c := t; t := b - (q × d); b := d;
            d := t; t := u2 - (q × v2); u2 := v2; v2 := t; goto 5
         end else
7:       if b = 0 v (lu - lv) > 1 then
         begin iqr(u, v, hulp); u := v; v := hulp end else
         begin hulp := u;
            u := isum(iprod(sint(a), u), iprod(sint(b), v));
            v := isum(iprod(sint(c), hulp), iprod(sint(d), v))
         end;
         lu := len(u); lv := len(v); goto loop
      end
   end;
eindigcd:
end;

formula procedure rinv(a); value a; formula a;
begin integer t;
   t := type of a;
   if t = short integer v t = long integer then rinv :=
      (if isign(a) > 0 then dyadic(rat numb, one1, a)
      else dyadic(rat numb, minone1, invert(a)))
   else
   begin formula l, r;
      l := lhs of a; r := rhs of a;
      rinv := if l = one1 then r else
```

```
        if l = minone1 then invert(r) else
        if isign(l) > 0 then dyadic(rat numb, r, l)
        else dyadic(rat numb, invert(r), invert(l))
    end
end;

formula procedure st rat(a, b); value a, b; formula a, b;
begin integer ta, tb;
    ta := type of a; tb := type of b;
    if b = one1 then st rat := a else
    if b = minone1 then st rat := invert(a) else
    if b = zero1 then
    begin st rat := zero1; nlcr; nlcr;
        printtext(⊰b equals zero in st rat⊱);
        nlcr; nlcr
    end else
    if a = zero1 then st rat := zero1 else
    if (ta = short integer ∨ ta = long integer) ∧
        (tb = short integer ∨ tb = long integer) then
    st rat := if isign(b) > 0 then dyadic(rat numb, a, b)
        else dyadic(rat numb, invert(a), invert(b))
    else
    st rat := error(⊰wrong parameter in st rat⊱)
end;

formula procedure oper on num1(oper, a, b); value oper, a, b; integer oper;
            formula a, b;
begin integer ta, tb;
    ta := type of a; tb := type of b;
    oper on num1 :=
        if (ta = short integer ∨ ta = long integer) ∧
            (tb = short integer ∨ tb = long integer) then
        (if oper = sum then isum(a, b) else
        if oper = product then iprod(a, b) else
        rnprod(a, rinv(b)))
        else
        if oper = sum then rnsum (a, b) else
        if oper = product then rnprod(a, b) else
```

```
      oper on num1(product, a, rinv(b))
end;

formula procedure rnsum(a, b); value a, b; formula a, b;
begin integer ta, tb;
   formula la, ra, l, r, b1;
   ta := type of a; tb := type of b;
   if ta ≠ rat numb then
   begin la := lhs of b; ra := rhs of b; b1 := a
   end else
   begin la := lhs of a; ra := rhs of a; b1 := b end;
   if ta = rat numb ^ tb = rat numb then
   begin formula gcd, lb, rb, hulp;
      lb := lhs of b; rb := rhs of b;
      gcd := igcd(ra, rb);
      if gcd ≠ one1 then
      begin rb := iqr(rb, gcd, hulp);
         l := isum(iprod(la, rb), iprod(lb, iqr(ra, gcd, hulp)));
         r := iprod(ra, rb);
         gcd := igcd(l, r);
         if gcd ≠ one1 then
         begin l := iqr(l, gcd, hulp); r := iqr(r, gcd, hulp)
         end
      end else
      begin l := isum(iprod(la, rb), iprod(lb, ra));
         r := iprod(ra, rb)
      end
   end else
   begin l := isum(la, iprod(ra, b1)); r := ra
   end;
   rnsum := st rat(l, r)
end;

formula procedure rnprod(a, b); value a, b; formula a, b;
begin integer ta, tb;
   formula la, ra, l, r, b1, gcd, hulp;
   ta := type of a; tb := type of b;
   if ta ≠ rat numb then
```

```
    begin ra := rhs of b; la := lhs of b; b1 := a
    end else
    begin ra := rhs of a; la := lhs of a; b1 := b end;
    if ta = rat numb ∧ tb = rat numb then
    begin formula lb, rb;
       lb := lhs of b; rb := rhs of b;
       gcd := igcd(ra, lb);
       if gcd ≠ one1 then
       begin ra := iqr(ra, gcd, hulp);
          lb := iqr(lb, gcd, hulp)
       end;
       gcd := igcd(rb, la);
       if gcd ≠ one1 then
       begin la := iqr(la, gcd, hulp);
          rb := iqr(rb, gcd, hulp)
       end;
       l := iprod(la, lb); r := iprod(ra, rb)
    end else
    begin gcd := igcd(ra, b1);
       if gcd ≠ one1 then
       begin ra := iqr(ra, gcd, hulp);
          b1 := iqr(b1, gcd, hulp)
       end;
       l := iprod(la, b1); r := ra
    end;
    rnprod := st rat(l, r)
end;

formula procedure oper on num(oper, a, b); value oper, a, b; integer oper;
            formula a, b;
if oper = sum then
oper on num := (if a = zero1 then b else
   if b = zero1 then a else oper on num1(sum, a, b))
               else
if oper = product then
oper on num := (if a = zero1 v b = zero1 then zero1 else
   if a = one1 then b else
   if b = one1 then a else
```

```
   oper on num1(product, a, b))
              else
if b = zero1 then
begin nlcr; nlcr; printtext(⌈b = 0 in oper on num⌉);
   oper on num := zero1; nlcr; nlcr
end else
oper on num := (if b = one1 then a else
   if a = zero1 then zero1 else
   if b = minone1 then
      (if type of a = rat numb then st rat(invert(lhs of a), rhs of a)
         else invert(a)) else
   oper on num1(quotient, a, b));

procedure outputint(n, sign); value n, sign; formula n; boolean sign;
begin integer l, t, i;
   if type of n = short integer then
   begin l := lhs of n;
      if sign then
      begin if l < 0 then printtext(⌈ - ⌉) else
         printtext(⌈ + ⌉)
      end;
      prnum(abs(l))
   end else
   begin integer array nn[1:len(n)];
      l := len(n);
      get array(n, i, 1, l, nn[i]);
      if sign then
      begin if nn[l] < 0 then printtext(⌈ - ⌉)
         else printtext(⌈ + ⌉)
      end;
      for i := l step -1 until 1 do
      begin t := abs(nn[i]);
         if i ≠ l then zeroos(t);
         prnum(t)
      end
   end
end;
```

```
procedure zeroos(n); value n; integer n;
begin integer j;
   j := radix;
   space(1);
   for j := j : 10 while n < j ^ j > 1 do prsym(0)
end;

procedure outputnumber(n); value n; formula n;
begin if type of n = rat numb then
   begin if isign(lhs of n) < 0 then
         printtext(⊰ - (⊱) else printtext(⊰ + (⊱);
      outputint(lhs of n, false); printtext(⊰ / ⊱);
      outputint(rhs of n, false); printtext(⊰)⊱)
   end else
   outputint(n, true)
end;

comment
```

3.5. Some examples

In this section we demonstrate the functioning of the simplification system and rational number system with some examples.;

```
begin formula x, y, z, f, g;
real time1;
procedure out(f, s); value f; formula f; string s;
begin integer t;
   t := type of f;
   carriage(3);
   printtext(s); printtext(⊰ = ⊱); nlcr;
   if t = short integer v t = long integer v t = rat numb then
         outputnumber(f) else output(f); nlcr;
   printtext(⊰used time =⊱); absfixt(3, 2, time - time1);
   printtext(⊰sec ⊱); time1 := time
end;
```

```
comment This section can be divided in three parts
1. test of the rational number system
2. test of the formula system
3. a more sophisticated example.

3.5.1. Test of the rational number system

1. input, representation and output of numbers.;

time1 := time;
out(sint(0),⊀sint(0)⊁);
out(zero1, ⊀zero1⊁);
out(sint(-1), ⊀sint(-1)⊁);
out(sint(1), ⊀sint(1)⊁);
out(sint(2), ⊀sint(2)⊁);
out(sint(-3), ⊀sint(-3)⊁);
x := lo int(4, li(li(li(li(0, 435512), 642890), 0), 10123));
out(x, ⊀x = lo int(4, li(li(li(li(0, 435512), 642890), 0), 10123))⊁);
y := lo int(4, li(li(li(li(0, -435512), -642890), 0), -10124));
out(y,
⊀y = lo int(4, li(li(li(li(0, -435512), -642890), 0), -10124))⊁);
out(st rat(sint(2), sint(3)), ⊀st rat(sint(2), sint(3))⊁);
out(st rat(sint(-2), sint(3)), ⊀st rat(sint(-2), sint(3))⊁);
out(st rat(x, y), ⊀st rat(x, y)⊁);
new page;

comment
2. int pow num and igcd.;

x := sint(32); out(x, ⊀x = sint(32)⊁);
y := sint(-20); out(y, ⊀y = sint(-20)⊁);
z := igcd(x, y); out(z, ⊀z = igcd(x, y)⊁);
out(oper on num(quotient, x, y), ⊀oper on num(quotient, x, y)⊁);
x := int pow num(x, 9); out(x, ⊀x = int pow num(x, 9)⊁);
y := int pow num(y, 9); out(y, ⊀y = int pow num(y, 9)⊁);
out(int pow num(z, 9), ⊀int pow num(z, 9)⊁);
out(igcd(x, y), ⊀igcd(x, y)⊁);
out(oper on num(quotient, x, y), ⊀oper on num(quotient, x, y)⊁);
```

```
new page;

comment
3. rational arithmetic;

x := oper on num(quotient,one1, sint(2));
out(x, ⊀x = oper on num(quotient, one1, sint(2))⊁);
y := oper on num(quotient, one1, sint(3));
out(y, ⊀y = oper on num(quotient, one1, sint(3))⊁);
z := oper on num(quotient, one1, sint(6));
out(z,  ⊀z = oper on num(quotient, one1, sint(6))⊁);
out(oper on num(sum, oper on num(sum, x, y), z),
    ⊀oper on num(sum, oper on num(sum, x, y), z)⊁);
out(oper on num(sum, oper on num(sum, x, y), oper on num(
   product, minone1, z)),
   ⊀oper on num(sum, oper on num(sum, x, y),
   oper on num(product, minone1, z))⊁);
new page;

comment
3.5.2. Test of the formula system

1. handling of numbers in the formula system;

f := 32; out(f, ⊀f = 32⊁); g := -20; out(g, ⊀g = -20⊁);
out(f/g, ⊀f/g⊁);
f := f↑9; out(f, ⊀f = f↑9⊁); g := g↑9; out(g, ⊀g = g↑9⊁);
out(f/g, ⊀f/g⊁);
out(1/2 + 1/3 + 1/6, ⊀1/2 + 1/3 + 1/6⊁);
out(1/2 + 1/3 - 1/6, ⊀1/2 + 1/3 - 1/6⊁);
new page;

comment
2. dyadic+, dyadic×, dyadic/, and monadic ↑;

out(av(⊀abc 123 ×+⊁), ⊀av(⊀abc 123 ×+⊁)⊁);
x := av(⊀x⊁); y := av(⊀y⊁); z := av(⊀z⊁);
out(2×x, ⊀2×x⊁); out(x + x, ⊀x+x⊁); out(y + x, ⊀y+x⊁);
```

```
out((x+y)↑5, ‹(x+y)↑5›); out((x-y)↑5, ‹(x-y)↑5›);
out(x/y, ‹x/y›);
f := -3×z + 2×y + x; out(f, ‹f = -3×z + 2×y + x›);
out(f↑5, ‹f↑5›); out(f↑(-5), ‹f↑(-5)›);
out(f/f, ‹f/f›); out(f/5, ‹f/5›); out(5×f/5, ‹5×f/5›);
out((x↑8 + y↑8) × (x↑4 + y↑4) × (x↑2 + y↑2) × (x + y) × (x - y),
‹(x↑8 + y↑8) × (x↑4 + y↑4) × (x↑2 + y↑2) × (x + y) × (x - y)›);
new page;
```

<u>comment</u>

3.5.3. <u>A more sophisticated example</u>

We introduce the following system:

```
   y = x / (2 - 4 × x),
   z = (3 + 2 × y) / 5,
   a = 2 × x / 3,
   b = 3 × x / 4,
   c = y↑2 × (y - x) / x↑2,
   d = z × (z - x) × (z - y) /
        (3 × y × (y - x)),
   e = (20 × x × y - 10 × x - 10 × y + 6) /
        (120 × z × (z - x) × (z - y) × (1 - z)),
   p = e × (1 - z),
```

and we will prove that

```
   f = a × b × c × d × p
```

equals

```
   (5 × x↑2 - 2 × x) / (960 × x - 480).;
```

```
y := x / (2 - 4×x); out(y, ‹y = x / (2 - 4×x)›);
z := (3 + 2×y) / 5; out(z, ‹z = (3 + 2×y) / 5›);
f := 2×x / 3; out(f, ‹a = 2×x / 3›);
g := 3×x / 4; out(g, ‹b = 3×x / 4›);
f := f × g; out(f, ‹f = a × b›);
g := y↑2 × (y - x) / x↑2; out(g, ‹c = y↑2 × (y - x) / x↑2›);
f := f × g; out(f, ‹f = f × c›);
g := z × (z - x) × (z - y) / (3 × y × (y - x));
out(g, ‹d = z × (z - x) × (z - y) / (3 × y × (y - x))›);
f := f × g; out(f, ‹f = f × d›);
```

```
g := (20 × x × y - 10 × x - 10 × y + 6)/ (120 × z × (z - x) × (z - y) ×
      (1 - z));
out(g, ⍃e = (20 × x × y - 10 × x - 10 × y + 6) /
   (120 × z × (z - x) × (z - y) × (1 - z))⍄);
g := g × (1 - z); out(g, ⍃p = e × (1 - z)⍄);
f := f × g; out(f, ⍃f = f × p⍄); carriage(5);
y := lhs of f × (960 × x - 480);
out(y, ⍃numerator of f × (960 × x - 480)⍄);
z := rhs of f × (5 × x↑2 - 2 × x);
out(z, ⍃denominator of f × (5 × x↑2 - 2 × x)⍄);
out(y - z, ⍃the difference⍄)

end
end
end
;
```

This program produced the following output:

Results of 3.5.1.1

```
sint(0) =
 + 0
used time =    .03 sec

zero1 =
 + 0
used time =    .04 sec

sint(-1) =
 - 1
used time =    .04 sec

sint(1) =
 + 1
used time =    .04 sec
```

```
sint(2) =
 + 2
used time =     .04 sec

sint(-3) =
 - 3
used time =     .04 sec

x = lo int(4, li(li(li(li(0, 435512), 642890), 0), 10123)) =
 + 435512 642890 000000 010123
used time =     .20 sec

y = lo int(4, li(li(li(li(0, -435512), -642890), 0), -10124)) =
 - 435512 642890 000000 010124
used time =     .20 sec

st rat(sint(2), sint(3)) =
 + (2 / 3)
used time =     .08 sec

st rat(sint(-2), sint(3)) =
 - (2 / 3)
used time =     .08 sec

st rat(x, y) =
 - (435512 642890 000000 010123 / 435512 642890 000000 010124)
used time =     .37 sec
```

<u>Results of 3.5.1.2</u>

```
x = sint(32) =
 + 32
used time =     .07 sec

y = sint(-20) =
 - 20
used time =     .05 sec
```

```
z = igcd(x, y) =
 + 4
used time =    .16 sec

oper on num(quotient, x, y) =
 - (8 / 5)
used time =    .32 sec

x = int pow num(x, 9) =
 + 35 184372 088832
used time =    .35 sec

y = int pow num(y, 9) =
 - 512000 000000
used time =    .27 sec

int pow num(z, 9) =
 + 262144
used time =    .18 sec

igcd(x, y) =
 + 262144
used time =   2.45 sec

oper on num(quotient, x, y) =
 - (134 217728 / 1 953125)
used time =   2.81 sec
```

Results of 3.5.1.3

```
x = oper on num(quotient, one1, sint(2)) =
 + (1 / 2)
used time =    .28 sec

y = oper on num(quotient, one1, sint(3)) =
 + (1 / 3)
used time =    .25 sec
```

```
z = oper on num(quotient, one1, sint(6)) =
 + (1 / 6)
used time =    .25 sec

oper on num(sum, oper on num(sum, x, y), z) =
 + 1
used time =    .71 sec

oper on num(sum, oper on num(sum, x, y),
   oper on num(product, minone1, z)) =
 + (2 / 3)
used time =    .93 sec
```

<u>Results of 3.5.2.1</u>

```
f = 32 =
 + 32
used time =    .11 sec

g = -20 =
 - 20
used time =    .25 sec

f/g =
 - (8 / 5)
used time =    .66 sec

f = f↑9 =
 + 35 184372 088832
used time =    .41 sec

g = g↑9 =
 - 512000 000000
used time =    .33 sec

f/g =
 - (134 217728 / 1 953125)
```

```
used time =   3.16 sec

1/2 + 1/3 + 1/6 =
 1
used time =   1.56 sec

1/2 + 1/3 - 1/6 =
 + (2 / 3)
used time =   1.96 sec
```

<u>Results of 3.5.2.2</u>

```
av(⋖abc 123 x+⋗) =
 + abc 123 x+
used time =    .21 sec

2×x =
 + 2×x
used time =    .46 sec

x+x =
 + 2×x
used time =    .24 sec

y+x =
 + x + y
used time =    .24 sec

(x+y)↑5 =
 + x↑5 + 5×x↑4×y + 10×x↑3×y↑2 + 10×x↑2×y↑3 + 5×x×y↑4 + y↑5
used time =   8.76 sec

(x-y)↑5 =
 + x↑5 - 5×x↑4×y + 10×x↑3×y↑2 - 10×x↑2×y↑3 + 5×x×y↑4 - y↑5
used time =   5.88 sec

x/y =
( + x) / ( + y)
```

```
used time =    .16 sec


f = -3xz + 2xy + x =
 + x + 2xy - 3xz
used time =   1.04 sec


f↑5 =
 + x↑5 + 10xx↑4xy - 15xx↑4xz + 40xx↑3xy↑2 - 120xx↑3xyxz + 90xx↑3xz↑2 + 80xx
↑2xy↑3 - 360xx↑2xy↑2xz + 540xx↑2xyxz↑2 - 270xx↑2xz↑3 + 80xxxy↑4 - 480xxxy↑3
xz + 1080xxxy↑2xz↑2 - 1080xxxyxz↑3 + 405xxxz↑4 + 32xy↑5 - 240xy↑4xz + 720xy
↑3xz↑2 - 1080xy↑2xz↑3 + 810xyxz↑4 - 243xz↑5
used time =  29.13 sec


f↑(-5) =
( 1 ) / ( + x↑5 + 10xx↑4xy - 15xx↑4xz + 40xx↑3xy↑2 - 120xx↑3xyxz + 90xx↑3xz
↑2 + 80xx↑2xy↑3 - 360xx↑2xy↑2xz + 540xx↑2xyxz↑2 - 270xx↑2xz↑3 + 80xxxy↑4 -
480xxxy↑3xz + 1080xxxy↑2xz↑2 - 1080xxxyxz↑3 + 405xxxz↑4 + 32xy↑5 - 240xy↑4x
z + 720xy↑3xz↑2 - 1080xy↑2xz↑3 + 810xyxz↑4 - 243xz↑5)
used time =  29.38 sec


f/f =
( + x + 2xy - 3xz) / ( + x + 2xy - 3xz)
used time =    .36 sec


f/5 =
 + (1 / 5)xx + (2 / 5)xy - (3 / 5)xz
used time =   1.23 sec


5xf/5 =
 + x + 2xy - 3xz
used time =   1.95 sec


(x↑8 + y↑8) x (x↑4 + y↑4) x (x↑2 + y↑2) x (x + y) x (x - y) =
 + x↑16 - y↑16
used time =  17.27 sec
```

<u>Results of 3.5.3</u>

```
y  = x / (2 - 4xx) =
(  + x) / ( - 4xx + 2)
used time =    .77 sec

z  = (3 + 2xy) / 5 =
(  - 2xx + (6 / 5)) / ( - 4xx + 2)
used time =   1.82 sec

a  = 2xx / 3 =
 + (2 / 3)xx
used time =    .88 sec

b  = 3xx / 4 =
 + (3 / 4)xx
used time =    .87 sec

f  = a × b =
 + (1 / 2)xx↑2
used time =    .71 sec

c   = y↑2 × (y - x) / x↑2 =
(  + 4xx↑4 - x↑3) / ( - 64xx↑5 + 96xx↑4 - 48xx↑3 + 8xx↑2)
used time =   6.49 sec

f  = f × c =
(  + 2xx↑6 - (1 / 2)xx↑5) / ( - 64xx↑5 + 96xx↑4 - 48xx↑3 + 8xx↑2)
used time =   1.16 sec

d  = z × (z - x) × (z - y) / (3 × y × (y - x)) =
(  - 1536xx↑7 + 5376xx↑6 - (203136 / 25)xx↑5 + (34368 / 5)xx↑4 - (439104 / 1
25)xx↑3 + (135456 / 125)xx↑2 - (23328 / 125)xx + (1728 / 125)) / ( + 3072xx
↑7 - 6912xx↑6 + 6144xx↑5 - 2688xx↑4 + 576xx↑3 - 48xx↑2)
used time =  31.75 sec

f  = f × d =
(  - 3072xx↑13 + 11520xx↑12 - (473472 / 25)xx↑11 + (445248 / 25)xx↑10 - (1 3
07808 / 125)xx↑9 + (490464 / 125)xx↑8 - (114384 / 125)xx↑7 + (3024 / 25)xx↑
6 - (864 / 125)xx↑5) / ( - 196608xx↑12 + 737280xx↑11 - 1 204224xx↑10 + 1 11
```

```
8208xx↑9 - 645120xx↑8 + 236544xx↑7 - 53760xx↑6 + 6912xx↑5 - 384xx↑4)
used time =  15.79 sec

e = (20 × x × y - 10 × x - 10 × y + 6) /
   (120 × z × (z - x) × (z - y) × (1 - z)) = -
( + 245760xx↑8 - 958464xx↑7 + 1 634304xx↑6 - 1 591296xx↑5 + 967680xx↑4 - 37
6320xx↑3 + 91392xx↑2 - 12672xx + 768) / ( + 368640xx↑8 - 1 437696xx↑7 + (12
 331008 / 5)xx↑6 - (60 742656 / 25)xx↑5 + (37 573632 / 25)xx↑4 - (74 663424
 / 125)xx↑3 + (18 602496 / 125)xx↑2 - (2 654208 / 125)xx + (165888 / 125))
used time =  53.17 sec

p = e × (1 - z) =
( - 491520xx↑9 + 2 113536xx↑8 - (20 176896 / 5)xx↑7 + (22 450176 / 5)xx↑6 -
 (16 041984 / 5)xx↑5 + 1 526784xx↑4 - 483840xx↑3 + (492288 / 5)xx↑2 - (5836
8 / 5)xx + (3072 / 5)) / ( - 1 474560xx↑9 + 6 488064xx↑8 - (63 700992 / 5)x
x↑7 + (366 280704 / 25)xx↑6 - (54 355968 / 5)xx↑5 + (674 390016 / 125)xx↑4
- (223 736832 / 125)xx↑3 + (47 821824 / 125)xx↑2 - (5 971968 / 125)xx + (33
1776 / 125))
used time =  19.14 sec

f = f × p =
( + 1509 949440xx↑22 - 12155 092992xx↑21 + 46053 457920xx↑20 - (2 726572 32
6912 / 25)xx↑19 + (22 598922 534912 / 125)xx↑18 - (27 824800 923648 / 125)x
x↑17 + (131 730131 386368 / 625)xx↑16 - (97 960723 218432 / 625)xx↑15 + (2
316560 891904 / 25)xx↑14 - (5 474047 131648 / 125)xx↑13 + (413535 633408 /
25)xx↑12 - (3 102712 455168 / 625)xx↑11 + (730625 605632 / 625)xx↑10 - (528
5 154816 / 25)xx↑9 + (3543 330816 / 125)xx↑8 - (331 849728 / 125)xx↑7 + (96
 878592 / 625)xx↑6 - (2 654208 / 625)xx↑5) / ( + 289910 292480xx↑21 - 2 362
768 883712xx↑20 + (45 320226 471936 / 5)xx↑19 - (543 389732 831232 / 25)xx↑
18 + (182 438735 118336 / 5)xx↑17 - (5687 470989 508608 / 125)xx↑16 + (5454
 629538 103296 / 125)xx↑15 - (4109 058478 964736 / 125)xx↑14 + (492 224190
087168 / 25)xx↑13 - 9 428143 177728xx↑12 + (451 095573 823488 / 125)xx↑11 -
 (137 203032 784896 / 125)xx↑10 + (32 746889 281536 / 125)xx↑9 - (1 200600
907776 / 25)xx↑8 + (32639 680512 / 5)xx↑7 - (77478 100992 / 125)xx↑6 + (458
6 471424 / 125)xx↑5 - (127 401984 / 125)xx↑4)
used time = 177.43 sec
```

```
numerator of f x (960 x x - 480) =
 + 1 449551 462400xx↑23 - 12 393665 003520xx↑22 + 50 045764 239360xx↑21 - (
634 030185 775104 / 5)xx↑20 + (5647 747843 620864 / 25)xx↑19 - (7511 858340
 691968 / 25)xx↑18 + (38648 089669 533696 / 125)xx↑17 - (31454 551471 03027
2 / 125)xx↑16 + (20523 721710 108672 / 125)xx↑15 - (2162 966277 390336 / 25
)xx↑14 + (922 502732 709888 / 25)xx↑13 - (1588 206311 571456 / 125)xx↑12 +
(438 140511 977472 / 125)xx↑11 - (95 508801 257472 / 125)xx↑10 + (3 217193
828352 / 25)xx↑9 - (403874 906112 / 25)xx↑8 + (177888 559104 / 125)xx↑7 - (
9809 952768 / 125)xx↑6 + (254 803968 / 125)xx↑5
used time =  29.70 sec

denominator of f x (5 x x↑2 - 2 x x) =
 + 1 449551 462400xx↑23 - 12 393665 003520xx↑22 + 50 045764 239360xx↑21 - (
634 030185 775104 / 5)xx↑20 + (5647 747843 620864 / 25)xx↑19 - (7511 858340
 691968 / 25)xx↑18 + (38648 089669 533696 / 125)xx↑17 - (31454 551471 03027
2 / 125)xx↑16 + (20523 721710 108672 / 125)xx↑15 - (2162 966277 390336 / 25
)xx↑14 + (922 502732 709888 / 25)xx↑13 - (1588 206311 571456 / 125)xx↑12 +
(438 140511 977472 / 125)xx↑11 - (95 508801 257472 / 125)xx↑10 + (3 217193
828352 / 25)xx↑9 - (403874 906112 / 25)xx↑8 + (177888 559104 / 125)xx↑7 - (
9809 952768 / 125)xx↑6 + (254 803968 / 125)xx↑5
used time =  30.85 sec

the difference =
 0
used time =  15.94 sec
```

3.6. An alternative simplification algorithm

Simplification is nothing but a complicated sorting technique. It is well-known that sorting of real numbers can be done very rapidly. In this section we, therefore, investigate the process in which the terms of the manipulated sums are mapped in a one-to-one fashion on real numbers. The real numbers are sorted and via the inverse mapping the original terms are reconstructed.

We use only those real numbers r being integral and satisfying $0 \leq r < 2 \uparrow 40$.

The mapping is not a fixed one but depends on the occurrence of the algebraic variables and the particular simplification to be performed.

Let the algebraic variables that are involved be v[p], $1 \leq p \leq m$.

Let the maximal exponents with which v[p] can occur be em[p], then the mapping is defined as follows:

Let a term t be given as:

$$t = \text{coef} \times \prod_{p=1}^{m} v[p] \uparrow e[p], 0 \leq e[p] \leq em[p],$$

then, the real number, called its <u>code</u>, is determined by:

$$c(t) = \sum_{p=1}^{m} e[p] \times 2 \uparrow vs[p-1],$$

$$\text{where } vs[p] = \sum_{q=1}^{p-1} vs[q] + 1 + \text{entier}(\ln(em[p]) / .69131\ 47180\ 56),$$

$1 \leq p \leq m$, and where vs[0] = 0.

The latter contribution to vs[p] is the number of bits to store the maximal exponent em[p].

Given a sum of terms f, it is the task of the procedure "avs and exps in list" to indicate in the formal array varf[1:40], whether the variable with index p occurs in f and, if this is the case, to make varf[p] = 1 and expf[p] = the maximal exponent with which this variable occurs in f, and to make varf[p] = expf[p] = 0 in the other case.
Note that the index of an algebraic variable is used to identify the variable in a term. Thus el j of lhs of rhs of el i of f is the index of the algebraic variable a[i,j] in:

$$f = \sum_{i=1}^{n} coef[i] \times \prod_{j=1}^{l[i]} a[i,j] \uparrow e[i,j]$$

and e[i,j] is given by el j of rhs of rhs of el i of f.

Not all the variables have to occur in a given sum f, it is, therefore, necessary to renumber the variables. This is the task of "avshifts", which also calculates the shiftfactors vs[p]. This procedure uses the following global integer arrays:
.var[0:40]: originally indicating, by means of var[p] that the variable with index p does or does not occur (var[p] = 1 or var[p] = 0). Afterwards, when the variable has got a new index q, var[q] = p.
.var2[0:40]: for holding the new index var2[p] = q.
.varshift[0:40]: determining the shiftfactor vs[p].
.exp[0:40]: containing the maximal exponents em[p].

Computing the code of the k-th term of a sum f is the task of the real procedure "make code (f,k)".
The inverse mapping is performed by the procedure "rowav", while the sorting process is carried out by "sort". The latter procedure is based on the "quicksort" method of C.A.R. Hoare [6].

```
integer array var,var2,exp,exp2,varshift[0:40];
formula procedure av(s); value s; formula s;
begin integer i,j; avcntr:= avcntr + 1;
  if avcntr > 40 then error(⌈too much algebraic variables⌉);
  stringstring:= dyadic(0,s,stringstring);
  av:= polyadic(simple sum,i,1,dyadic(0,one1,dyadic(0,
        rowadic(0,j,1,avcntr),rowadic(0,j,1,1))))
end av;

formula procedure dyadic+(a,b); value a,b; formula a,b;
if a = zero then dyadic+:= b else if b = zero then dyadic+:= a
else if type of a = simple quotient then
        dyadic+:= if type of b = simple quotient then
        (lhs of a × rhs of b + lhs of b × rhs of a)/
                                  (rhs of a × rhs of b) else
        (lhs of a + b × rhs of a)/rhs of a else
if type of b = simple quotient then
        dyadic+:= (a × rhs of b + lhs of b)/rhs of b else
begin integer la,lb,l,up; la:= length of a;
  lb:= length of b; l:= la + lb;
  begin array code[1:l]; integer i,j,d;
    formula array coef[1:l];
    avs and exps in list(a,la,var,exp);
    avs and exps in list(b,lb,var2,exp2);
    for i:= 1 step 1 until avcntr do
    if var2[i] = 1 then
    begin var[i]:= 1;
      if exp2[i] > exp[i] then exp[i]:= exp2[i]
    end;
    avshifts(up);
    for i:= 1 step 1 until la do
    begin code[i]:= make code(a,i);
      coef[i]:= lhs of el i of a
    end;
    for i:= 1 step 1 until lb do
    begin code[la+i]:= make code(b,i);
```

```
      coef[la+i]:= lhs of el i of b
    end;
    sort(code,coef,l);
    dyadic+:= if l = 0 then zero else
              if l = 1 ^ code[1] = 0 then
             (if coef[1] = one1 then one else
              polyadic(simple sum,i,1,dyadic(0,coef[1],
                       dummy variable))) else
              polyadic(simple sum,i,l,dyadic(0,coef[i],
              dyadic(0,rowav(code[i],d,up),
                       rowadic(0,j,d,exp[j]))))
end end dyadic+;

formula procedure dyadic×(a,b); value a,b; formula a,b;
if a = zero v b = zero then dyadic×:= zero else
if a = one then dyadic×:= b else if b = one then dyadic×:= a
else if type of a = simple quotient then
        dyadic×:= if type of b = simple quotient then
        (lhs of a × lhs of b)/(rhs of a × rhs of b) else
        (lhs of a × b)/rhs of a else
if type of b = simple quotient then
        dyadic×:= (a × lhs of b)/rhs of b else
begin integer la,lb,l,up,i,d,j,ind;
  la:= length of a; lb:= length of b; l:= la × lb;
  begin array code[1:l]; formula array coef[1:l];
    real codei; formula coefi;
    avs and exps in list(a,la,var,exp);
    avs and exps in list(b,lb,var2,exp2);
    for i:= 1 step 1 until avcntr do
    if var2[i] = 1 then
    begin var[i]:= 1; exp[i]:= exp[i] + exp2[i] end;
    avshifts(up);
    for i:= 1 step 1 until la do
    begin codei:= make code(a,i); coefi:= lhs of el i of a;
      ind:= (i-1) × lb; for j:= 1 step 1 until lb do
      begin code[ind+j]:= make code(b,j) + codei;
        coef[ind+j]:= oper on num(product,
               lhs of el j of b,coefi);
```

```
        comment By means of these two statements the actual
        multiplication has been performed;
    end end;
    sort(code,coef,l);
    dyadicx:= if l = 1 ^ code[1] = 0 then
             (if coef[1] = one1 then one else
              polyadic(simple sum,i,1,dyadic(0,coef[1],
                       dummy variable))) else
              polyadic(simple sum,i,l,dyadic(0,coef[i],
              dyadic(0,rowav(code[i],d,up),
                       rowadic(0,j,d,exp[j]))))
end end dyadicx;

procedure avs and exps in list(f,lf,varf,expf);
value lf; integer lf; formula f;
integer array varf,expf;
begin formula refav,refexp;
  integer i,k,var,exp;
  for i:= 0 step 1 until avcntr do
  varf[i]:= expf[i]:= 0;
  for k:= 1 step 1 until lf do
  begin refav:= lhs of rhs of el k of f;
    refexp:= rhs of rhs of el k of f;
    for i:= length of refav step -1 until 1 do
    begin var:= el i of refav;
      exp:= el i of refexp;
      if varf[var] = 0 then
      begin varf[var]:= 1; expf[var]:= exp end else
      if exp > expf[var] then expf[var]:= exp
    end
  end
end avs and exps in list;

procedure avshifts(up);
begin integer i,bits,varctr;
  varctr:= bits:= var2[0]:= varshift[0]:= 0;
  for i:= 1 step 1 until avcntr do
  if var[i] = 1 then
```

```
  begin var2[i]:= varctr:= varctr + 1;
    var[varctr]:= i; varshift[varctr]:= bits:=
    bits + 1 + entier(ln(exp[i]) / .691314718056);
    if bits > 40 then error(⊰too much bits needed⊱);
  end;
  up:= varctr
end avshifts;

real procedure make code(f,k);
value k; integer k; formula f;
begin formula refav,refexp;
  integer i; real code; code:= 0;
  refav:= lhs of rhs of el k of f;
  refexp:= rhs of rhs of el k of f;
  for i:= length of refav step -1 until 1 do
    code:= code + 2 ↑ varshift[var2[el i of refav] -1] ×
    el i of refexp;
  make code:= code
end make code;

formula procedure rowav(c,ub,up);
value c,up; integer up,ub; real c;
begin integer i,varctr; real shift,oldshift,tail;
  varctr:= 0; oldshift:= 1;
  for i:= 1 step 1 until up do
  if c ≠ 0 then
  begin shift:= 2 ↑ varshift[i];
    tail:= c - entier(c/shift) × shift;
    if tail ≠ 0 then
    begin varctr:= varctr + 1;
      c:= c - tail; exp2[varctr]:= var[i];
      exp[varctr]:= tail/oldshift
    end;
    oldshift:= shift
  end;
  rowav:= rowadic(0,i,varctr,exp2[i]);
  ub:= varctr
end rowav;
```

```
procedure sort(code,coef,orde);
real array code; integer orde;
formula array coef;
begin integer i,pivot,j,ncode,ub; real max,s;
  formula f;
  procedure sor(l,r); value l,r; integer l,r;
L1: if l ≥ r then else
  begin real h,y; integer p,q,s; p:= l-1; q:= r+1;
    y:= (code[l] + code[r])/2;
  L2: for p:= p+1 while p < q do
    if code[p] > y then
    begin for q:= q-1 while q > p do
      if code[q] ≤ y then
      begin h:= code[p]; code[p]:= code[q]; code[q] := h;
        f:= coef[p]; coef[p]:= coef[q]; coef[q]:= f;
        goto L2
      end; s:= p; goto L3
    end; s:= q;
  L3: if s = r+1 then s:= r; sor(l,s-1); l:= s; goto L1
  end sor;
  ub:= orde; sor(1,ub);
  s:= code[1]; ncode:= 1;
  for i:= 2 step 1 until ub do
  if abs(code[i] - s) <.1 then
  begin coef[ncode]:= oper on num(sum,coef[ncode],coef[i]);
    if coef[ncode] = zero1 then
    begin ncode:= ncode - 1; s:= - 1 end
  end else
  begin ncode:= ncode + 1; s:= code[ncode]:= code[i];
    coef[ncode]:= coef[i]
  end;
  orde:= ncode
end sort;
```

The above procedures have been used to run the same test program as the test program of section 3.5. It is, of course, not meaningful to reproduce this test program and the results as they are the same except for the computation times used. We therefore summarize the computation times only. In general, there was a considerable difference in computation times when the manipulations concerned much multiplications of large sums of terms. In particular, the method of this section turned out to be two to three times more rapid in these cases. For some examples we give a table of computation times:

	section 3.5	section 3.6
-3xz+2xy+x	1.04	1.19
x/(2-4xx)	.77	.75
y↑2x(y-x)/x↑2	6.49	3.14
(x+y)↑5	8.76	2.83
(-3xz+2xy+x)↑5	29.13	10.58
example 3.5.3.	386.47	285.09

It should be noted that the procedures of section 3.6 are by far not optimal. In particular, frequent use has been made of the <u>el</u> operator. This is an expensive operator however. Considerable gain can be obtained by eliminating this operator as has also been done in section 3.5.

In a forth-coming report of A. de Bruin and R. Wiggers the two simplification methods will be compared in more details. Moreover, a comparison will be made with the already mentioned method of G. ten Velden.

As a final remark, note again that the method of this section is restricted to problems with not too much variables and not too high exponents.

4. The translated program

In this chapter we treat the desired form of the translated ABC ALGOL program, together with the running system.
The running system consists of a collection of ALGOL 60 procedures which govern the dynamic storage allocation processes.
The translation of the ABC ALGOL program and the running system are not so simple that they can be treated immediately in section 4.1. Their treatment has to be preceded by a section on a simple but unrealistic translation, a section on the dynamic storage allocation system and a section on a feasible but very (run-time) inefficient system.
Before entering into these sections, it is necessary to introduce some nomenclature. Moreover, we make some general observations concerning storage space management.
As we shall go into the details of the implementation, where addresses, pointers and references play an important role, we define the following terms:
A formula value is a structured collection of (at least one) boxes. Each formula value has a reference. A formula value and its reference are defined as follows:

A simple formula value is a box, called its own box, which does not contain a reference; the reference of this simple formula value is the reference of its box.

A formula value is either a simple formula value, in which case its reference is the reference of that simple formula value, or there exists a box, called its own box, containing at least one reference, in which case the formula value is the collection of this box and the boxes of the formula values referred to by the references contained in that box; the reference of the formula value is the reference of that box.

We say that a reference refers to a formula value, if it is the reference of that formula value. Each reference refers to one and only one formula value.
The definition of a formula value is done in terms of boxes; we therefore have to define boxes:

A box contains three items:

1. The first one is called its type, which is a (small) natural number, and which can be of five different categories: constant, monadic, dyadic, polyadic and rowadic.

2. The second one is called the lhs, which may either be an integral number or a reference.

3. The third one is called the rhs, which may either be an integral number or a reference.

Furthermore, each box has one and only one reference (which may be thought of as its address or the pointer pointing to it).
Each <u>formula</u> occurring in ABC ALGOL is represented by means of a formula value; i.e. a structured collection of boxes. Simple formulas, as algebraic variables or numbers, are represented as simple formula values; tree-like formulas are represented as formula values. Example: the formula x×x + x, with x an algebraic variable, may be represented as follows:

box 1 with reference r1:	1	33	0
box 2 with reference r2:	2	r1	r1
box 3 with reference r3:	3	r2	r1

The numbers 1, 2, 3 and 33 are codes chosen for "algebraic variable", "product", "sum" and "x", respectively.

During the computation, each <u>formula</u> <u>variable</u>, subscripted or not subscripted, can have several formulas as value (at different times). The space occupied by these different formulas, i.e. the number of boxes of their formula values, is not necessarily the same. Upon block-entry, when a formula variable is introduced by means of its declaration, it is thus not possible to reserve the space needed for its formula value. Neither is it possible to determine this space upon block-exit in order to make it available for other use.

Looking more closely at the relation between a formula variable and its formula value, we observe that a formula variable may very well have as value the reference of its formula value. The amount of space for this reference is fixed; this space can thus be reserved on block-entry and released on block-exit.

4.1. The simplest but unrealistic translation

Consider the following ABC ALGOL program:

```
begin   formula g;
        begin   formula f,x;
                x:= av(⊀x⊁); f:= x×x + x;
                x:= f + f×x; g:= f
        end;
        g:= g + 10
end
```

Assuming the existence of an integer array C, we can represent a box with reference k by the three array elements: C[k,1],C[k,2] and C[k,3]. The run-time system may now consist of:

```
integer procedure STORE(type,lhs,rhs);
        value type,lhs,rhs; integer type,lhs,rhs;
    begin   STORE:= k:= k+1; C[k,1]:= type;
            C[k,2]:= lhs; C[k,3]:= rhs
    end;
    integer procedure  SUM(a,b);  SUM:= STORE(3,a,b);
    integer procedure PROD(a,b); PROD:= STORE(2,a,b);
    integer procedure ALG VAR(letter);
                      ALG VAR:= STORE(1,letter,0);
    integer procedure INT NUM(value);
                      INT NUM:= STORE(0,value,0);
```

and the translation of the ABC ALGOL program above can simply be:

```
    begin  integer G;
           begin   integer F,X;
                   X:= ALG VAR(33); F:= SUM(PROD(X,X),X);
                   X:= SUM(F,PROD(F,X)); G:= F
           end;
           G:= SUM(G,INT NUM(10))
    end
```

Here, and in the sequel we shall adopt the convention of using lower-case letters for identifiers in the ABC ALGOL program and upper-case letters for the corresponding identifiers in the translated program. In the actual translation, however, another mechanism is used so that an ABC ALGOL programmer may choose his identifiers without any restriction.

Giving the integer k the initial value 0, we see that the values of the references r1, r2 and r3 of the preceding section are 1, 2 and 3, respectively.

A compiler for this translation could be very small: it has to change "formula" into "integer" and a formula expression into its equivalent Polish prefix form. And indeed this is the rough structure of the compiler of chapter 6.

The reason that the simple translation scheme does not fulfill the needs is hidden in the size of the array C, which was, until now, tacitly assumed to be indefinitely large. In practice this size is bounded, and it is our task to provide the means for using the available space as well as possible. Referring to our example, we see that after the second block is completed, the formula value for f will still be used in the surrounding block; the final formula value of x will, however, partly not be used any longer. This means that two boxes are used for unreachable formula values. At any moment we can divide the formula values into two classes: the interesting and the non-interesting class. Members of the interesting class are those formula values the references of which are either the values of "active" formula variables or are contained in the boxes of formula values belonging to this interesting class. Formula values not in the interesting class are members of the non-interesting class.

It is necessary to provide the procedure STORE with the means to be able to separate the two classes. For the above case STORE has to know that the integers G, F and X have values being the references of formula variables; and it has to know the lifetime of these integers. How this can be done, is the subject of the next section.

4.2. The dynamic-storage allocation system

. At an arbitrary moment, during the computation process, the procedure STORE is called to allocate space (a box), for a new formula value. Assume that there is no free space left. Now we may have two implementations: one in which there may be left space occupied by non-interesting formula values and one in which the class of non-interesting formula values is always empty. We will discuss the second implementation later on.
There are two methods to distinguish interesting and non-interesting formula variables: the external and the internal method.
The internal method consists of putting the information about a formula variable being interesting or not inside its own box; which means that STORE can simply inspect all boxes in a straightforward manner.
In the external method we use a list of all references to formula values being the values of "active" formula variables; STORE can separate the interesting from the non-interesting formula values in a recursive fashion.
As far as STORE is concerned, the internal method is preferable. Other operations are block-entry, assignation and block-exit. If we assign a formula value f2 to a formula variable r, which originally had the formula value f1, then we have to do something with the administration in the boxes of f1 and f2 when we follow the internal method. Indeed, f1 may have become non-interesting and f2 surely remains or becomes interesting. How do we know that f1 really has become non-interesting in another way than by inspecting all other interesting formula values and seeing whether they refer to f1 at the cost of much computing time. There is a way: the information does not only tell whether a formula value is interesting but also how interesting it is. In particular, this information can be a counter, called reference counter, which counts the number of times the formula value is referenced. In our case, the reference counter of f1 has to be decreased by one and the reference counter of f2 has to be increased by one.
With the external method, we only have to change the list of all references being the value of formula variables; i.e. the reference to f1 in this list has to be replaced by a reference to f2.
With respect to assignments, we arrive at the conclusion that the internal method has the following disadvantage: each box needs space for the reference counter and, as some values may be referenced very often, this space has to be more than just a few bits.

Considering the block-entry, we observe that upon declarations of formula variables, the list of references of values of formula variables has to be enlarged in the internal method, whereas nothing has to be done in the internal method.
Finally, during block-exit the actions necessary in the external method comprise the shrinking of the list of references of values of formula variables only, whereas the actions in the internal method consist of decreasing the reference counters of the values of the formula variables to be deleted.

Concluding, we remark that the internal method is advantageous only when STORE has to collect the non-interesting formula values (called a garbage collection) often. Thus in a case that 90 percent of the memory is needed. However, in such a case we can better use the 10 percent space, needed for reference counters, for more boxes.
The two methods do not exclude each other. It is also possible to have a mixed method: The space for the reference counter is kept small; after its maximum has been reached and another reference has to be dealt with, a special bit is turned on; the property "being interesting" for formula values for which this bit is on, is dependent now on a list of references of values of formula variables.
The advantages are that the number of garbage collections can be kept to a minimum while the space needed for reference counters does not need to be exceedingly large; in fact some spare bits, which frequently are available, can be used. The disadvantages are that we still need space, which in our case can not be found in spare bits and that it is rather complicated.

More or less orthogonal to the internal-external controversy, stands the question: relocation or no relocation, with which we mean the following. After STORE has determined which formula values are interesting and which are not, it can proceed in two directions: the interesting formula values are relocated such that the free space forms a contiguous space, or the interesting formula values are not relocated but the non-contiguous free space is chained together to form a free list (i.e. a linear chain of free boxes).

Sometimes it is unavoidable to relocate; e.g. in a case where we also have a stack, growing upward and downward in memory, or in a case that the simple formula values are of different size.
In our case, where we use a fixed-size linear array in which the formula values need to be stored only, and where we are not forced to have simple formula values of different size, we choose the non-relocation version. The main argument is that in the relocation version the values of references may change after a garbage collection. The effect is that it is very dangerous to store such a value into an integer variable since the value of the reference can change while this integer variable remains unchanged. In the following sections, we will see that integer variables play an important role. For a detailed comparison between the relocation and the non-relocation methods we refer to [13], in which complete ALGOL programs are given.

We close this section by returning to the beginning, when we introduced a system in which non-interesting formula values never occur. This system can easily be implemented as a non-relocation and internal system in which each box, whose reference counter has become equal to zero, is immediately chained to the free list.

4.3. The external non-relocation method

The procedure STORE should always have at its disposal the list of references of formula values of formula variables. It is, of course, impossible that STORE can be made aware of a set of integer variables since these are known to the ALGOL 60 compiler system only. Therefore, we have to make an explicit list, of items referring to formula values. Each such item is called a <u>name</u>. A name is a box and it has a reference, the reference of the box. The box either contains no reference, in which case it contains NULL, or it contains a reference to a formula value.
A formula variable f, declared as "<u>formula</u> f", has two counterparts in the translated program: the integer variable F, declared as "<u>integer</u> F", and a name F', created by a call of DE. Immediately after the declaration, the value of F is the reference to F' (i.e. F points to F') and F' contains NULL.

After an assignment "f:= expr", translated into: "ASSIGN(F,EXPR)", the value of F is still the reference to F' but the reference contained in F' is the reference of the formula value defined by EXPR.
On block-exit the integer variable F disappears automatically by the ALGOL 60 system and the name F' disappears by a call of ERASE. To be more specific, the following program:

```
        begin   formula f;
                f:= expr
        end
```

is translated into:

```
        begin   integer F,fnn; fnn:= gnn; DE(F,NULL);
                ASSIGN(F,EXPR)
                ; ERASE(fnn)
        end
```

In order to understand this translation, it is necessary to observe that the list of names can be organized in a Last In First Out fashion; i.e. as a stack. This is a consequence of the simple scheme:

upon block-entry create n names

upon block-exit delete the n lastly created names.

We use a global integer gnn for counting the number of names and, for each block, a local integer fnn, which, as indicated above, gets the value of gnn at the moment just preceding the creation of the names. It is now easy to imagine what happens after a call "ERASE(fnn)": precisely gnn - fnn lastly created names are erased and gnn gets the value of fnn.
Each block may have an integer with the same identifier "fnn" since the scope of this integer always includes the statement "ERASE(fnn)", as it is the last statement of the block. The problems, concerning a jump out of the block without passing the block-exit, will be dealt with in section 4.6.

For reasons of efficiency and elegancy, the list of names is implemented in the same array C which is used for the formula values. The array element C[k,3] serves for internal linking of the list, while C[k,2] is used for storing either NULL or the reference to a formula value. Again the procedure STORE puts the three values: type, lhs and rhs in C and it performs a garbage collection if necessary, its precise declaration will be given in the next section.
Now we give, for illustrative reasons only, the following procedures in a non-definite form.

```
procedure DE(f,E); value E; integer f,E;
begin   last name:= STORE(name list type,E,last name);
        f:= last name; gnn:= gnn + 1
end;
procedure ASSIGN(f,E); value f; integer f,E;
begin   C[f,2]:= E end;
procedure ERASE(old gnn); value old gnn; integer old gnn;
for old gnn:= old gnn while gnn > old gnn do
begin   last name:= C[last name,3]; gnn:= gnn - 1 end
```

4.3.1. Translating an Expression

The translation of an expression will now be considered. Take the expression "x × y + u × v", in which x, y, u and v are simple formula variables for which the integer variables X, Y, U and V have been declared in the translated program. As in section 4.1 the translation has the form "S(P(...,...), S(...,...))"; but we still have to decide how the primaries x, y, u and v shall be translated. In fact, this decision depends on the nature of the quantity which is being delivered by the procedure identifiers themselves. The nature can be:

1. a reference to a formula value;
2. a reference to a name (which itself refers to a formula value).

4.3.1.1. The reference-to-a-name approach

If the procedure S has to deliver a reference to a name, which itself refers to the formula value, then its declaration has to be of the form:

```
integer procedure S(a,b); value a,b; integer a,b;
begin   integer Sname; DE(Sname,STORE(3,C[a,2],C[b,2]));
        S:= Sname
end;
```

However, the translation of "x × y + u + v" into "S(P(X,Y), S(U,V))" then leads to the introduction of 3 different names, of which only one is needed to refer to the formula value. It is obligatory to have, besides an automatism for creating names, an automatic erasure of names, as is done in the following declaration:

```
integer procedure S(a,b); value a,b; integer a,b;
begin   integer Sname,fnn; Sname:= STORE(3,C[a,2],C[b,2]);
        fnn:= gnn - 2; ERASE(fnn); DE(Sname,Sname); S:= Sname
end
```

This procedure needs a little explanation: After STORE is called for storing the formula value, the last two created names are erased and then a new name is created in which the reference to the formula value is stored. During the creation of that new name, a garbage collection is in principle possible and this could destroy the newly created formula value (there is still no name referring to it). First, we remark that we can explicitly inform STORE about the existence of a special formula value which is interesting and, second, we observe that DE is called immediately after the erasure of two names. In the next section we will see that space becoming free as a result of erasing names can immediately be returned to free space and this means that a garbage collection can not occur. Of course, the second line of the procedure body could be made more simple as follows: "fnn:= gnn - 1; ERASE(fnn); S:= C[a,2]:= Sname", where we used the fact that the order of execution of called-by-value parameters is the order in which they occur in the value part.
Coming back to the translation of the primaries x, y, u and v, we see that the translation can not be "S(P(X,Y),P(U,V))" since S, and also P, assume that just prior to the evaluation of their actual parameters two new names have been created, which may be erased as soon as their values have been used. Therefore, we need the procedure:

```
integer procedure EN(N); value N; integer N;
begin   DE(N,C[N,2]); EN:= N end
```

for creating an Extra Name; we now get the translation:
"S(P(EN(X),EN(Y)),S(EN(U),EN(V)))"
The whole should be put as actual parameter of the following procedure which erases the lastly created name and delivers the reference to the formula value:

```
integer procedure EXPR(actual expr); value actual expr;
integer actual expr;
begin integer fnn; EXPR:= C[actual expr,2]; fnn:= gnn - 1;
      ERASE(fnn)
end
```

The translation of "f:= x × y + u + v" then takes the form:

"ASSIGN(F,EXPR(S(P(EN(X),EN(Y)),S(EN(U),EN(V)))))"

Evaluating the above approach, we conclude that it is complicated and inefficient with respect to the number of names used.
It is complicated since the creation and erasure of names is being done in different procedure calls. It needs a good insight into the dynamics of the process to see that the system works correctly.
It is inefficient as may be seen already from the translation of the simple example: "f:= x + y" into: "ASSIGN(F,EXPR(S(EN(X),EN(Y))))" in which three names are created and erased although none was necessary to save a formula value.

4.3.1.2. The reference-to-a-value approach

The procedure S delivers a reference to a formula value. This may be a saved or a possibly not saved one. By saved we mean: there does exist a name referring to this value. If S has to deliver the reference of a saved value, we are almost in the same position as in the preceding approach; therefore, we follow the second approach. S delivers a reference of a not necessarily saved value and we have the following procedure declaration:

```
integer procedure S(a,b); integer a,b;
begin integer an,bn,fnn; fnn:= gnn; DE(an,a); bn:= b;
      S:= STORE(3,C[an,2],bn);
      ERASE(fnn)
end
```

The actual parameters of S should also deliver references to possibly non-saved formula values; therefore, the formula value, to which a refers, has to be saved explicitly before the actual parameter b is evaluated, since this evaluation could lead to a garbage collection. The primaries x, y, u and v should now be translated such that the references to their formula values are delivered and we thus have as translation of: "f:= x × y + u + v":

"ASSIGN(F,S(P(C[X,2],C[Y,2]),S(C[U,2],C[V,2])))".

Comparing this approach with the preceding one, we see that the creation and deletion of names is far less complicated: there is a static correspondence between each DE and its ERASE counterpart. Also with respect to efficiency this approach is superior: for "f:= x + y" it uses one name. Furthermore, it will be shown later on that it is even possible to make a safe system in which this (superfluous) name is not used. The latter system is a modification of the system which is now being described.

As an example, we shall now give the procedures S and P which apply the distributive laws and perform the simple 0 and 1 simplifications; in ABC ALGOL they have the following form:

```
formula procedure dyadic + (a,b); value a,b; formula a,b;
dyadic +:= if a = zero then b else if b = zero then a else
dyadic(3,a,b);

formula procedure dyadic × (a,b); value a,b; formula a,b;
dyadic ×:= if a = zero ∨ b = zero then zero else
      if a = one then b else if b = one then a else
      if type of a = 3 then lhs of a × b + rhs of a × b else
      if type of b = 3 then a × lhs of b + a × rhs of b else
```

```
        dyadic(2,a,b).
```

While in ALGOL 60 their form is given by:

```
integer procedure S(A,B); integer A,B;
begin   integer An,Bn,fnn; fnn:= gnn; DE(An,A); DE(Bn,B);
        S:= if C[An,2] = C[ZERO,2] then C[Bn,2] else
            if C[Bn,2] = C[ZERO,2] then C[An,2] else
            STORE(3,C[An,2],C[Bn,2]);
        ERASE(fnn)
end;

integer procedure P(A,B); integer A,B;
begin   integer An,Bn,fnn; fnn:= gnn; DE(An,A); DE(Bn,B);
        P:= if C[An,2] = C[ZERO,2] v C[Bn,2] = C[ZERO,2] then
                                                C[ZERO,2] else
            if C[An,2] = C[ONE,2] then C[Bn,2] else
            if C[Bn,2] = C[ONE,2] then C[An,2] else
            if C[C[An,2],1] = 3 then S(P(C[C[An,2],2],C[Bn,2]),
                                      P(C[C[An,2],3],C[Bn,2]))
                                 else
            if C[C[Bn,2],1] = 3 then S(P(C[An,2],C[C[Bn,2],2]),
                                      P(C[An,2],C[C[Bn,2],3]))
                                 else
            STORE(2,C[An,2],C[Bn,2]);
        ERASE(fnn)
end.
```

Observe that it is necessary for P to introduce names for both A and B and not only for A. The reason is that the evaluation of "lhs of a × b" may involve a garbage collection so that b might be erased if there was no name saving the formula value of b. For S it would have been sufficient to create a name for a only.

Due to the fact that no assignments take place to the parameters a and b of the above procedures, we can give a much more efficient (with respect to time) version by means of the following procedure:

```
procedure DEVAL(F,E); integer F,E;
begin   DE(F,E); F:= C[F,2] end
```

The effect of DEVAL is that a name is created for saving a formula value defined by E. It is not possible to give this created name a reference to another formula value. Assuming that ONE and ZERO have been introduced by means of DEVAL, i.e. that their values are references to formula values instead of names, we may declare S and P as follows:

```
integer procedure S(A,B); integer A,B;
begin   integer Av,Bv,fnn; fnn:= gnn; DEVAL(Av,A); DEVAL(Bv,B);
        S:= if Av = ZERO then Bv else
            if Bv = ZERO then Av else STORE(3,Av,Bv);
        ERASE(fnn)
end;
integer procedure P(A,B); integer A,B;
begin   integer Av,Bv,fnn; fnn:= gnn; DEVAL(Av,A); DEVAL(Bv,B);
        P:= if Av = ZERO v Bv = ZERO then ZERO else
            if Av = ONE then Bv else if Bv = ONE then Av else
            if C[Av,1] = 3 then S(P(C[Av,2],Bv),P(C[Av,3],Bv))
                           else
            if C[Bv,1] = 3 then S(P(Av,C[Bv,2]),P(Av,C[Bv,3]))
                           else
            STORE(2,Av,Bv);
        ERASE(fnn)
end.
```

Let us close this section by showing that the translation of any ABC ALGOL expression, in which the only operators are + and × and in which the only primaries are formula variables, leads to an ALGOL 60 expression which, when executed, has the property that garbage collections can not destroy partially formed results still being necessary. We call this translation a *safe* translation.

Consider an arbitrary product expression: "expr1 × expr2", translated into: "P(EXPR1,EXPR2)". From the moment P is called until the moment the value of P is computed and delivered we shall follow the execution precisely.
Through "DEVAL(Av,A)" EXPR1 is evaluated and we assume that it is evaluated correctly. A name is created to which the formula value is assigned, so that, until "ERASE(fnn)", the value is safe. Through "DEVAL(Bv,B)" EXPR2 is evaluated and the remarks made for EXPR1 pertain also to EXPR2. After P has got a value the following possibilities are open:
1. The value of EXPR1 (or EXPR2) is the value of P.
2. This is not so.
After the execution of "ERASE(fnn)" the values of EXPR1 and EXPR2 are no longer safe. Only in the first case, however, we have to see what happens. If "expr1 × expr2" is itself part of another expression, then a name will be introduced to save the value; if "expr1 × expr2" is the right-hand side of an assignment statement or appears as actual parameter, it is the responsibility of the assignment statement mechanism or the parameter mechanism to save the value (if necessary).

We thus arrive at the conclusion that during the execution of the procedure body of P, including the evaluation of its actual parameters, the derived partial results remain safe until P has got its value. It is the responsibility of the mechanism that called P to save the formula value to which the value of P refers.
In passing, we remark that during the execution of the translation of (x+y) × (u+v), the formula values x + y and u + v are formed but, after P is finished, they belong to the garbage.
Similar arguments may be applied to the execution of the procedure S.

4.3.2. <u>The translation of parameters of type formula</u>

A formal parameter a of a procedure p specified of type formula and called by value is translated in the same way as the parameters a and b of the preceding section: An integer Av is declared, the statement "DEVAL(Av,A)" is executed and each occurrence of a in the procedure body is translated as Av. It is not possible to assign a value to a parameter, called by value. This simple and natural restriction enables us to make the efficient translation with Av instead of C[An,2]. A formal parameter b called by name is treated analogously as in ALGOL 60. We now have to face the following problems: b may occur at the left-hand side of an assignment statement, in which case the actual parameter should deliver a reference to a name, or b may occur in expressions only when the reference to a value is needed.
On the other hand, the actual parameter may deliver a reference to a name or the reference to a value, and this can not be investigated syntactically. Hence, it is necessary to build in such means that it dynamically can be seen whether a reference is a reference to a name or to a value. An occurrence of b in an expression is, therefore, translated as "V(B)", where the procedure V is defined as follows:

<u>integer</u> <u>procedure</u> V(N); <u>value</u> N; <u>integer</u> N;
V:= <u>if</u> N is a reference to a name <u>then</u> C[N,2] <u>else</u> N.

At the left-hand side of an assignment statement, b is translated as B; in the procedure ASSIGN we check whether B is the reference to a name, which it should be.

The price for the freedom, to use the actual called-by-name parameter for names as well as for values, is inefficiency in time as we have to distinguish at run-time between two categories of references.

4.3.3. The translation of a formula procedure

As we have seen already in section 4.3.1.2, the heading of a formula procedure is changed into an integer procedure. If the last executed statement of the procedure body can syntactically be shown to be always the assignment to the procedure identifier the translation of the assignment to the procedure identifier can be simple: "p:= expr" is translated into "P:= EXPR". If this cannot be shown syntactically, the translation has to be more complex due to the fact that we want to be able to give error messages in the case that the procedure identifier does not obtain a value, or in the case that after the assignment statement to the procedure identifier and before the block-exit an action takes place which may invoke a garbage collection.

4.3.4. An example

With the aids discussed until now, we can easily translate the following example:

```
formula procedure der(f,x); value f,x; formula f,x;
begin   integer t; t:= type of f;
        der:= if f = x then one else
              if t = 3 then der(lhs of f,x) + der(rhs of f,x)
                       else
              if t = 2 then der(lhs of f,x) × rhs of f +
                               lhs of f × der(rhs of f,x)
                       else zero
end
into:
integer procedure DER(F,X); integer F,X;
begin   integer T,Fv,Xv,fnn; fnn:= gnn; DEVAL(Fv,F); DEVAL(Xv,X);
        T:= C[Fv,1];
        DER:= if Fv = Xv then ONE else
        if T = 3 then S(DER(C[Fv,2],Xv),DER(C[Fv,3],Xv)) else
        if T = 2 then S(P(DER(C[Fv,2],Xv),C[Fv,3]),
                        P(C[Fv,2],DER(C[Fv,3],Xv)))
        else ZERO;
        ERASE(fnn)
end.
```

Let us compute the number of names which are simultaneously in existence for calculating the derivative of f; denote it by n(f).
When f is x or f is not a sum or a product then n(f) = 2. When f is a sum, a + b, then n(f) = max(2 + n(a), 3 + n(b)). When f is a product, a × b, then we have in the simple case that P does not apply the distributive law: n(f) = max(2 + n(a), 4 + n(a)). For f = 1 + (x + (x × x + x × (x × x))), we thus have at a certain moment 19 names in use. This should be compared with the 6 formula values which ultimately result from the computation.
The conclusion is that, for this example, we need 3 times more space for saving partially formed results than we need for the final value. Knowing that the number of names which is really needed for saving partial results as x + x, another x + x and x × (x + x) is 3, we seriously have to try to remedy this situation. Also in another, less sophisticated, example, experiments showed that the number of simultaneously existing names is approximately 6 times more than the number which is really necessary. The reason for this probibitive large amount of names is that the same formula value is saved simultaneously several times in different calls of DER, S and P.

4.4. A more clever and more efficient system

We turn our attention again to the formal parameters of S and P, and we ask whether it is possible to create names conditionally instead of always. It must, therefore, be possible to exchange information about a formula value whether it is saved or possibly not saved. This information must be exchanged between several procedure calls and it is simply possible to put this information in the value of the reference to the formula value.
A first solution seems to be marking a reference if it is the reference to a probably non-saved formula value. The procedure S may then take on the form:

```
integer procedure S(A,B); integer A,B;
begin   integer Av,Bv,fnn; fnn:= gnn;
        Av:= A; if Av < 0 then DEVAL(Av,Av);
        Bv:= B; if Bv < 0 then DEVAL(Bv,Bv);
        S:= if Av = ZERO then Bv else Bv = ZERO then Av
            else - STORE(3,Av,Bv);
        ERASE(fnn)
end.
```

The above form is not suitable, however, due to the fact that S may get as value the value of Av when Bv = ZERO. The value of Av refers to a formula value which is just recently being saved; i.e. the actual parameter A was negative. The corresponding name, referring to Av, is erased immediately after S has got a value; hence, the value of S indicates that S refers to a saved formula value while this is not necessarily the case.
One remedy would be to investigate the value of S during the erasure of the names. This means that all names, all formula values and sub formula values which are erased are compared with the value of S. If one is found equal to the value of S, S is made negative. This is very time-consuming a process; moreover, it is not watertight in the general case. Consider for example:

```
formula procedure p(a); value a; formula a;
begin ... begin formula b; ... p:= if random < .5 then
                lhs of a else rhs of b;
          end; ...
end
```

It would be necessary to translate each block-exit such that it compares the values of a number of procedure identifiers with the values of the to be erased names and formula values.

Another solution would be to compare the value of the procedure identifier with the values of still existing names and formula values. This is more easy to compile, of course, since this action need only be performed at the block-exit of the procedure body of the formula procedure involved. But again it is a time-consuming process. Let us call this the comparison method.

A third solution, called the minus method,is by changing the assignment statement into:
"S:= <u>if</u> Av = ZERO <u>then</u> -Bv <u>else</u> <u>if</u> Bv = ZERO <u>then</u> -Av
    <u>else</u> -STORE(3,Av,Bv)".

This is, of course, not an optimal solution, as may be seen in the example:

"<u>begin</u> <u>formula</u> f; ...(f + 0) + 0 ... <u>end</u>" translated into:
"<u>begin</u> <u>integer</u> F; DE(F,NULL); ... S(S(C[F,2],ZERO),ZERO) ... <u>end</u>",
in which one superfluous name is created for f.

In order to make a comparison, we did some experiments with calculating the first 7 Taylorcoefficients with a program, similar to the one in section 1.2, in which numbers are not combined automatically. The four methods are:
1. the simple method of section 4.3; no marking of references.
2. the comparison method.
3. the comparison method, in which the value of the procedure identifier is always made negative.
4. the minusmethod.

The quantities measured are:
1. the computation time in millihours (mh), excluding the time necessary for printing the results (6 mh);
2. the maximum number of names simultaneously in use, excluding the names for the declarations;
3. the number of times STORE is called (to be compared with the number of values being created: 3069 in total);
4. the number of garbage collections.

	method1	2	3	4
time (in mh)	55	56	33	31
number of names	82	9	13	9

number of STORE calls	44450	6700	13650	10770
number of garbage collections	4	4	4	4

On both frontiers: computation time and storage space, the minus method is the favourite one.

Alas, such as the method stands now, it is not watertight. Consider the example:

```
begin  formula f;
       formula procedure p(g); value g; formula g;
       begin  f:= 20; ... p:= g end;
       f:= 10;
       f:= p(f)
end
```

We observe that after execution of "f:= 20", the value of the integer G, as it is translated from g, refers to the old value "10" of f and the reference is marked safe; the formula value "10" is, however, far from safe; in fact there is no name referring to it, so that a garbage collection during "..." would be disastrous.
The reason for the trouble is that the status of an arbitrary formula value can very easily be changed from safe to non-safe, by simply assigning another value to the name which referred to it. For the compiler it is almost impossible to see whether an arbitrary formula variable will change its value as this may be done in deeply hidden procedures.
There is, on the other hand, a category of names which will never change their values: those names which are introduced by means of DEVAL and which, anonymously, serve only to save formula values. The statuses of the latter formula values can change only on block-exit: a well defined place and easily recognizable by the compiler.

Call a formula variable f, declared as "_formula_ f = expr", and a formal parameter, called-by-value and specified as formula, a _fixed variable_. The property of a fixed variable then is that it obtains a value immediately after its creation which it holds until its death. If, upon creation of a fixed variable f1, its value turns out to be the value of another fixed variable f2, it is not necessary to create a name since the lifetime of f2 is at least as large as the lifetime of f1 and, hence, the value of f1 is safe during the lifetime of f1, so that no name need be created to save it.

The procedure DEVAL may now have the form:

```
procedure DEVAL(F,E); integer F,E;
begin   F:= E; if ¬ is marked fixed(F) then
        begin  DE(F,F); F:= C[F,2]; mark as fixed(F) end
end,
```

and the introduction of the fixed variable f with a value defined by expr is performed by: "DEVAL(F,EXPR)". After its introduction, f can occur as primary in an expression only. Its appearance elsewhere, at the left-hand side of an assignment or as an actual parameter of a formal call-by-name parameter appearing at the left-hand side of an assignment, leads to an error-message either at compiletime or at runtime.

As primary in an expression we only need to consider the case that f appears as actual parameter of a procedure, as the operators are translated into function designators. Take, therefore, "p(f)" as a procedure call to be translated. Let the formal parameter be a call-by-value parameter; "f" may then be translated into "F", i.e. the reference to its formula value is delivered including the marking. The direct effect is that, within the procedure body of p, no name is introduced for the formal call-by-value parameter. During the lifetime of p, f exists; hence, during the execution of p we may consider the value of f to be safe since an assignment to f is impossible.

Assume now that the formal parameter of p is a call-by-name parameter. In this case, f should be passed through to the place in p where the formal parameter appears. Again "f" should be translated as "F", i.e. including its marking, as this knowledge may be worth-while within the procedure body of p.

The other case which needs consideration is that the expression is nothing but "f" itself, or that the expression is a conditional expression in which a simple expression consists of "f" only. What need be done now is dependent on the context in which the expression is placed.

1. As the right-hand side of the assignment to a formula variable. The action of ASSIGN is to peel off the "fixed" marking and to put the reference of f without this marking into the name of the formula variable.

2. As the right-hand side of the assignment to a formal parameter. This case is precisely the same as the first case.

3. As the right-hand side of the assignment to a procedure identifier. The discussion about this case will be postponed a little while.

4. As an actual parameter of a procedure call. This has been considered already above.

5. As the right-hand side in a formula declaration; e.g. in: "formula g:= if random < .5 then f else 3.14, h = f". Now the expression appears as second actual parameter of DE or DEVAL. In the DE case, a name is introduced and the expression is treated analogously to case 1.; in the DEVAL case, the "fixed" marking is finally being taken care of.

Within an arithmetic or Boolean expression, we may also encounter the fixed variable "f", as e.g. in: "if f = g then ...". The reference which now should being delivered has to be free from the "fixed" marking, otherwise the test in the following example would fail: "begin formula f = one, g:= one; if f = g then ... end".

A special discussion is necessary for the case "p:= f" or "p:= if random < .5 then f else ...", where p is the procedure identifier of a formula procedure. The value calculated by p, i.e. the reference to the formula value produced within the procedure body of p, is brought out of the procedure body of p and is still being used after the execution of p is done. This reference has a lifetime which is surely longer than the lifetime of formula variables declared in the body of p or of local variables, introduced for call-by-value parameters.

Hence, if f is a fixed variable, declared inside p, a translation into: "P:= if random < .5 then F else ... " would be erroneous: the reference, being the value of P, would be marked "fixed", even after f is erased at the block-exit of p.

In the case that f is declared inside p, or is a formal parameter of p or of a procedure declared inside p, the "fixed" marking has to be peeled off before P gets a value. It would be most easy to peel off the "fixed" marking always, but this would also exclude an efficient translation of constructions like: "der:= ... <u>then</u> one <u>else</u> zero", where we assume that one and zero are globally defined fixed variables.
Therefore, we peel the "fixed" marking off when f is a local fixed variable only. This has, however, more consequences: we allow the transfer of the "fixed" marking only once; namely where it is syntactically clear that no danger exists; i.e. in a case "p:= ... f" as described above. We do not allow this transfer from procedure to procedure or from formal parameter called-by- name to procedure. The reason for this is exemplified in the following:

```
begin   formula procedure p(f); value f; formula f;
        begin   formula p1 = f;
                procedure p2(g); formula g; p:= g;
                if random < .5 then p:= p1 else p2(f)
        end;
        ...
end
```

In this example, p1 gets a value including the "fixed" marking, so that this marking could be transferred to p, if no measures were taken. The same pertains to g, the actual parameter of which delivers the reference including the "fixed" marking.

We concentrate now a little bit more on the translation of "p:= f", where f is global to the procedure body of p, so that the "fixed" marking is transferred outside this body, to the place where the procedure p is called. At this place f exists already, otherwise f could not be global to the procedure body of p and exist within it, Therefore, we could replace this call of p by f itself without changing the result of this call (apart from side-effects by the body of p of course; these can have no effect, however, on f itself as f is fixed). Maintaining the "fixed" marking is, therefore, subject to the same condition as for f itself:

if the call of p is the simple expression in the right-hand side of an assignment to a formula-procedure identifier q, then the "fixed" marking is maintained when p is global to q, otherwise it is removed.

Removing the marking, results in a specific translator action such as putting "abs(...)" in the translated program. Taking into account that this removing is only necessary for local procedures, and that it seldomly occurs that formula procedures are declared within formula procedures. We conclude that in the translated result we will normally not find any such removals. Only in far-fetched and very sophisticated programs they will be put (See example 6.1 of chapter 7).

The silly, but non-serious, restriction, that the declarations of the formula procedures for the operators are forbidden within a formula procedure body, is a direct consequence of the considerations above. If the procedure identifier p, in the above example, were "<u>dyadic</u>+", say, then the translation of a sum a + b, occurring in the body of p should involve the removing of the "fixed" marking. This gave some troubles within the compiler, which could have been overcome for a price which was considered too high in view of the very improbable construction. Therefore, a simple restriction and a check on the appearance of such a declaration within a procedure body has been built in. After all, defining an operator is different from defining a procedure.

The above discussions do not only pertain to fixed variables but also to formula enquiries in which the kernel is a fixed variable, as e.g. "<u>lhs</u> <u>of</u> f" or "<u>rhs</u> <u>of</u> <u>el</u> 5 <u>of</u> <u>el</u> 7 <u>of</u> f". The reference of a formula enquiry is marked fixed if its kernel is fixed.

Discussing the marking itself, we remark that there are only two possibilities:

1. The reference of a fixed variable is positive while other references are negative. This was the way the original marking, concerning the saved or not saved property, was performed.
2. The other way around: the reference of a fixed variable is negative.

It does seem that the second approach has some slight advantages: a) the ordinary references do not have to be changed before they are put into the boxes; b) "abs(...)" is cheaper than "-abs(...)"; c) a reference corresponding to a fixed variable may be more exceptional.
Therefore, a reference is marked "fixed" by making it negative.

An actual call-by-name parameter p, may have four different values.
a) a reference to a name;
b) a reference to a formula value being the value of a fixed variable;
c) a reference to a formula value for which it is not known whether it is the value of a fixed variable;
d) NULL; i.e. the "value" of a formula variable which did not obtain a formula value as value.
The reference of such a parameter p is defined by:

ref:= _if_ p < N _then_ C[N-p,2] _else_ abs(p).

Here N is a negative number, in absolute value larger than the maximum number of possible references to formula values.

4.5. Formula procedures

Until now we assumed that the assignment to a formula-procedure identifier: "fp:= expr" is translated into "FP:= EXPR"; i.e. a direct assignment to the procedure identifier "FP" of the translated procedure. This simple translation scheme cannot be maintained, however, due to two problems:
1. At the moment the formula value has been created, the reference of which is assigned to FP, this value may be not saved, so that a garbage collection may destroy it before the value of FP is used in the expression where FP occurred as a function designator. Example:

"_formula_ _procedure_ fp; _begin_ fp:= x + y; f:= x × y _end_;

if translated into:

"_integer_ _procedure_ FP; _begin_ FP:= S(X,Y); ASSIGN(F,P(X,Y))
end",

then the execution of "ASSIGN(F,P(X,Y))" may destroy the value "x + y".

2. A formula procedure may not have got a value at all with the effect that its value is undefined, i.e. an arbitrary value. In the complicated system we work with, where dynamically changing datastructures are involved with references to names, to values of fixed variables and to other values, with three layers of languages: ABC ALGOL -> ALGOL 60 -> machinecode, in such a system it would be very difficult to find the error in:

```
"begin formula procedure fp; if 1 < 0 then fp:= x + y;
        f:= fp
end",
```

where "1 < 0" should have been written as "1 < d" and where FP gets the value 52 which just happens to be the reference of the formula value x × y.

A simple and straightforward solution of the problem is to require that the last statement of the procedure body is the procedure-identifier-assignment-statement (piass) in such a way that the compiler can check it.
Another, expensive, solution is to introduce a local formula variable lfp and to treat the piass as if the procedure identifier were replaced by this ordinary variable. The procedure identifier should then obtain a value by means of "FP:= LFP" at the end of the body. With this solution we run into another problem: In the preceding section we carefully designed a system which introduced as few as names possible. A lot of the profit would be lost due to the introduction of a name for lfp; observe, moreover, that it is impossible to transfer the information, about the reference being the reference of a formula value of a fixed variable or not, through formula procedures. The result would be that, e.g., the procedure "der" for differentation, would introduce a lot of names for "saving" one's and zero's.
The solution chosen is a modus between the two solutions above. Instead of a formula variable, an integer, LFP, is introduced as a local variable. This integer gets the initial value NULL immediately after its introduction; it is used in the left-hand side of the piass; while the end of the procedure body is translated as:
"if LFP = NULL then ERR(⌈no assignment to proc ident⌉); FP:= LFP".

This solves the second problem in a cheap way; we really do not have to pay for the integer since it does not take up space for storing formula values and names. The first problem is solved by requiring that the last action of the procedure body, in which the creation of a formula value or a name is involved, is the piass. This makes constructions like:

"for ... do begin ... if i < 0 then begin fp:= expr; goto end
end end"

possible. However, constructions like:

"fp:= expr; if i < 0 then fp:= expr2"

are forbidden (assuming that i can take on a negative value). The requirement is not very restrictive as the ABC ALGOL programmer can always introduce himself, at the cost of space, a local formula variable. The requirement is, moreover, in the light of ALGOL 68 developments, a rather natural one.

It is, of course, not sufficient to state the requirement, it is also necessary to build in an automatic test. The translated program has therefore at its disposal a global Boolean variable "protect", getting the initial value false. Immediately after the piass, protect is made equal to true; and at the end of the procedure body, protect is changed to false again. It is not hard to see that by building the error message:

"if protect then ERR(⊀protection error⊁);

into the run-time procedure STORE, and by inserting this error message at the beginning of the translated procedure body, the translation scheme is watertight. Indeed, if protect has the value true, we only have to be careful that it does not obtain the value false before the end of the procedure body is reached. It can, however, take on this value only by a formula procedure call. But this leads immediately to an error message.

Concluding, the translation of:

"formula procedure fp; begin ... fp:= expr ... end" can be:

```
"integer procedure FP;
begin   integer LFP; if protect then ERR(
        ⊀protection error in form proc⊁);
        LFP:= NULL; ... begin LFP:= EXP; protect:= true end ...
        ; if LFP = NULL then ERR(⊀no assignment to proc ident⊁);
        FP:= LFP; protect:= false
end".
```

This looks rather awkward indeed, although it is necessary in the general case. In a very frequently occurring special case, namely where the compiler can see that the last action is the piass, the translation produced is simply:
"integer procedure FP; begin ... FP:= EXPR ... end".

In the following examples the compiler produces the simple translation:

```
begin    ... if ... then begin ... fp:= expr end else
             if ... then begin ... fp:= expr end
                    else begin ... fp:= expr end
end
```

and begin ... fp:= expr;; end
but it gives the "protect-with-local-integer" version in:

```
begin    ... for i:= 1 do fp:= expr end
```

and in

```
begin    ... if 0 < 1 then fp:= expr end
```

See also the examples of section 7.5.

Another solution would have been to provide the end of the procedure body with the label "EOP" and translate "fp:= expr" into "begin LFP:= EXPR; goto EOP end". This gives, however, problems with formula procedures declared within each other.
Example:

```
"formula procedure fp1;
begin   formula procedure fp2; begin ... fp1:= fp2:= expr ...
                                 end;
        f:= fp2
end", which when translated into:
"integer procedure FP1;
begin   ... integer procedure FP2;
        begin   ... begin LFP1:= LFP2:= EXPR; goto EOPFP2
                    end ...
        ; EOPFP2: ... end;
        ASSIGN(F,FP2)
;EOPFP1: ... end"
```

leads to a non-watertight translation. In the "protect" method it is simply possible to produce a translation in which "protect:= false" is omitted at the end of the body of fp2. It must be said, however, that special administration in the compiler is necessary for this purpose.

4.6. Jumps leading out of a block

Referring to section 4.3, we recall that the block-entry - block-exit: "begin declarations; statements end" is translated into:

```
"begin  integer ..., fnn; fnn:= gnn; DECLARATIONS;
        STATEMENTS;
        ERASE(fnn)
end
```

The global integer gnn counts the number of names currently in use. ERASE destroys the names introduced in the block, i.e. the names introduced by the declarations and those introduced by the statements.

Normally, the number of names may temporarily be enlarged during execution of a statement, but after the execution, this number is again the same as it was before. This is so since each block-entry has its block-exit counter part; thus each "fnn:= gnn" has its "ERASE(fnn)" counter part. However, this is only statically the case. Dynamically we may easily miss an "ERASE(fnn)". Hence, the execution of a statement may involve the introduction of extra names. The only way this can occur is that a block is left not via the normal block-exit. Thus a goto statement is executed, leading out of the block to a statements s in an outer block. Consider therefore:

l:s; ... ; begin ... goto l ... end ... "

At two places we can undertake the action to erase superfluous names:

1. The goto statement can be translated as: "erase the superfluous names; goto l". The serious problem is, however, how many superfluous names are to be erased; just "ERASE(fnn)" is not sufficient, as the jump may be to an outer-outer block or the label l may be a formal parameter of a procedure, or l could be a switch as e.g. in:

   "k:s; begin switch sw:= k,l; ... goto sw[1 + random] ... end"

2. "l:s" could be translated as: "L: erase the superfluous names; S".

The number gnn gives the total number of names when L is reached. The number of names which are of interest only for the execution of s is the number fnn, plus the names which are introduced in the block head of the block of s. Call this number "snn". With "the block of s" we mean: the block which contains s but which does not contain a block also containing s.

We thus arrive at the following construction:

"<u>begin</u> declarations; statements <u>end</u>" is translated into:

```
"begin     integer ...,fnn,snn; fnn:= gnn; DECLARATIONS;
           snn:= gnn;
           STATEMENTS
           ; ERASE(fnn)
end"
```

Now a labelled statement: "l:s" is translated into:

"L: <u>begin</u> ERASE(snn); S <u>end</u>"

Per definition, snn is the number of names accessible by the statements of the block; i.e. the names of enveloping (static and dynamic) blocks and the names of the block itself. The function of "ERASE(snn)" is to erase names if there are more than snn, so that after its execution, the number of names is again snn. Each statement could be provided with "ERASE(snn)". An unlabelled statement s, however, can only be reached after its textual predecessor is finished without ending with a goto. Therefore, the "ERASE(snn)" in front of its predecessor serves s equally well; an "ERASE(snn)" in front of s would thus be superfluous. Repeating the argument, we see that labelled statements need this treatment only.

Not always is the "ERASE(snn)" insertion necessary. If the block does not have an innerblock, or if it has an innerblock but this innerblock does not contain goto statements, or if it has an innerblock with gotos but this innerblock does not contain declarations of formula variables, in these cases "ERASE(snn)" may, in general, be left out. Not, however, if there are procedures with label parameters or, what is equally worse, with unspecified parameters and goto statements in their bodies.

Although the following analysis is a more syntactical analysis, and might, therefore, also be given in the part describing the compiler, we prefer to give it here due to its intimate connections with the subject of this section.

We want to determine the criteria to be used when deciding that there is no danger in omitting the "ERASE(snn)", or in omitting the declaration of the integer snn and its initialization.

First, we need some definitions: For the remainder of this section, we consider a procedure body as a block with fictitious block-entry and block-exit. Moreover the initializations in a declaration as in "formula f = 3.14,g:= 2.7" are considered as part of the statements; i.e. the above example is considered to be equivalent with "formula f,g; f:= 3.14; g:= 2.7" although it is not equivalent.

The own statements of a block b are those statements s, contained in b, for which b is the block; i.e. there is no other block contained in b containing s. Note that the statements of a compound statement are own statements of the block of the compound statement.

The envelope of a block b is the block e for which b is an own statement. We write e = env(b).

An innerblock of a block b is a block i such that b = env(i).

A block b has the goto-property, goto(b), if at least one of its own statements is a goto statement, or it has an innerblock i with the goto-property. Or formally:

goto(b) = one of its own statements is a goto statement
          v∃ i: (b = env(i) ^ goto(i))

A block b has the formula-property, form(b), if formula variables are declared in the block (including local formula names for formal formula parameters called by value).

A block b has the label-property, label(b), if one of its own statements is labelled.

A block b has the dangerous-property, dang(b), if:

form(b) ^ goto(b) v (∃i:(env(i) = b ^ dang(i))).

A program has the property: dangerous procedures, dang proc, if it contains a procedure declaration with an unspecified parameter or with a parameter specified as label, and its procedure body p has the property: goto(p).

It is now easy to formulate the criterium, with which we may decide that it can not be seen that "ERASE(snn)" is superfluous in the translation of the labelled statement: "l:s", where s is an own statement of the block b. The criterium is:

(∃ i:(b = env(i) ^ dang(i))) v dang proc.

Stated positively, if there is no innerblock which is dangerous and if there are no dangerous procedures, then we may omit "ERASE(snn)". From the definition we see that an innerblock which is not dangerous has the property that it does not have itself dangerous innerblocks and that it does not have both the goto- and the formula property. Hence, if all the innerblocks are not dangerous then we may omit "ERASE(snn)" indeed, since the labelled statement can only be reached from other own statements of the block and statements of innerblocks.

It will now be investigated when it can <u>not</u> be seen that the introduction of snn is superfluous (so that the compiler will introduce it).
Firstly, we remark that if a block does not have the formula- property, it is not necessary to introduce snn, as the snn of the enveloping block may well serve for this purpose. In other words, with respect to formulas, the block is not really a block; it might be considered as a compound statement so that the own statements of this block may in fact be considered as own statements of its enveloping block.
Secondly, we remark that we can say that a block has the responsibility for introducing snn if it is necessary to insert "ERASE(snn)" in one of its own statements or if one of its innerblocks has this responsibility but does not introduce snn since it does not have the formula-property.
There are two cases to be considered: one case where we have dangerous procedures and one case where these are not present.
A block b has the <u>snn1-property</u>, snn1(b), if:
$L(b) \wedge (\exists\, i: (b = env(i) \wedge dang(i))) \vee$
$\exists\, j: (b = env(j) \wedge snn1(j) \wedge \neg\, form(j)),$
A block b has the <u>snn2-property</u>, snn2(b), if:
$L(b) \wedge dang\ proc \vee \exists\, j: (b = env(j) \wedge snn2(j) \wedge \neg\, form(j)).$
If $(snn1(b) \vee snn2(b)) \wedge form(b)$, then snn will be introduced. If this is not the case, snn will not be introduced, since we can see that this would be superfluous. For, if $\neg\, form(b)$, it surely is superfluous; if $form(b) \wedge \neg\, snn1(b)$, it follows that no labels are present for which "ERASE(snn)" is necessary and that all the innerblocks j have the property that either form(j), in which case "snn" would have been introduced in the innerblock if also snn1(j), or $\neg\, form(j)$ but then $\neg\, snn1(j)$.
It is not difficult to see now, by means of an inductive argument, that, indeed, "snn" would be introduced superfluously.

same reasoning can be done in the case that there are dangerous edures, in which case property snn2 is used.

effect of the analysis is that normally one does not encounter "snn" and SE(snn)" in the translated program. This is a nice result and we cannot st the temptation to refer to the beautiful way this analysis has been t into the compiler. (See section 6.3.3)

The basic structures

elements with which formula values are built up are boxes; each box isting of three items: the type-category, the left-hand side (lhs) the right-hand side (rhs). There are five basic structures with which structures of an arbitrary formula value can be built up. The number, tifying the specific basic structure, is called the category, cat, and s a part of the type-category. The basic structures are:

onstant structures; cat = 0; the lhs and the rhs are integer: $(|\text{lhs}| < 2$ and $|\text{rhs}| < 2 \uparrow 17)$.

onadic structures; cat = 1; the lhs is an integer $(|\text{lhs}| < 2 \uparrow 26)$; the is a reference to a formula value.

yadic structures; cat = 2; the lhs and the rhs are references to formula es.

polyadic structures; cat = 3; the lhs is a non-negative integer n; rhs is a reference to a dyadic formula value r1. The lhs of ri is ference to a formula value vi, the rhs of ri is, in case $i < n$, a rence to a dyadic formula value ri + 1, for $1 \leq i \leq n$.

owadic structures; cat = 4; the lhs is a non-negative integer n; the rhs reference to a monadic formula value r1. The lhs of ri is an integer ci $| < 2 \uparrow 26)$, the rhs of ri is, in case $i < n$, a reference to a monadic ula value ri + 1, for $1 \leq i \leq n$.

visually the structures have the form:

```
              t    l    r
constant:  (0    [//] [//])

monadic:   (0    [//] [→])

dyadic:    (0    [→]  [→])

polyadic:  (0    [n]  [→]→(0 [→]   [→]→(0 [→]   [→]→..(0[→]   [ ])

rowadic:   (0    [n]  [→]→(0 [//]  [→]→(0 [//]  [→]→..(0[//]  [ ])
```

The box items: [//] contain an integer, [→] contain a reference,
[n] contains the value of n and [ ] contains NULL.

There are no internal reasons for differentiating polyadic and rowadic structures from the other structures since they are basically built up from monadic and dyadic structures. In fact, STORE does not even use the value of n, since, in the general case, this is not possible. The reason is that a garbage collection might occur in the middle of building up a polyadic structure; then the part which has been formed already should be saved and at that time the value of n may not be appropriate.
It is for external reasons that we have introduced the polyadic and rowadic structures including the difference in the categories. The reasons being to be able to safeguard the programmer for errors, which otherwise could lead to almost undiscoverably complicated situations.
As a basic principle, the system is provided with the maximum amount of error-checking possible. The backlying theory is that:
1. ABC ALGOL programs are complicated due to the complexities involved with dynamically changing structures and due to the two languages involved: ABC ALGOL and ALGOL 60;
2. it is easy to throw out the testing, much easier, indeed, than to insert testing.

In view of the second point, we remark that the compiler can easily be changed such that procedure-calls like TYPE, LHS and RHS do not occur, but instead the direct operations on the array elements are produced.

Instead of using a two-dimensional array, as has been suggested in the preceding section, we declare two arrays C1 and C2 to be used as the space for the boxes. C1[k] is used for the lhs, while C2[k] has to store both the type-category and the rhs:
$C2[k] = (cat \times 32 + type) \times 2 \uparrow 18 + rhs$.
This imposes the restrictions:
$0 \leq type < 32$ and $0 \leq rhs < 2 \uparrow 18$;
The value of the rhs of a constant structure is defined by:
<u>if</u> EVEN(rhs) = 1 <u>then</u> rhs $\underline{:}$ 2 <u>else</u> -rhs $\underline{:}$ 2.

The list of names is organized by means of a dyadic structure: the integer variable "last name" has as value either NULL, in which case the list is empty, or the reference to the last name called name[gnn]. The lhs of each name[i] g,$1 \leq i \leq gnn$, is either NULL, or is the reference to the formula value being saved; then the rhs of name[i] contains a reference to a dyadic structure name[i-1]. The rhs of name[1] contains NULL.
Within STORE, only two handles are used to which are attached all the names and formula values to be saved during a garbage collection; these are "last name" and the reference to the box into which STORE has just stored its three actual parameters. The value of this reference is the value of the integer "free cell", which at the beginning of STORE points to the first free box and at the end of STORE again points to the first free box. It is essential that STORE can always deliver its actual parameters in a free box, otherwise the simple statement "TRACE(free cell)" has to be changed into a write-out of TRACE for the triple: t, a and b.
The list of free cells is organized as follows: "free cell" points to the first free cell denoted by the index k while C1[k] points to the next; until a pointer equaling zero is found.

The garbage collection process, performed in STORE, makes heavy use of a procedure TRACE, which, for dyadic structures only, acts in a recursive fashion to trace the interesting, non-garbage, boxes. A direct iteration through the memory establishes the new list of free cells and undoes the cells from the markings produced by the tracing.

It is emphasized that we made use freely of space provided by the ALGOL 60 system for the recursive tracing. It would not be difficult to implement a more sophisticated scheme in which the space of C1 and C2 is used only for the tracing. For these techniques we refer to the literature: Knuth[7] and Van der Mey[9].

Of particular interest is the method used for the procedures DE, DEVAL and ERASE. While experimenting with the old procedures of sections 4.3 and 4.4 we discovered that a call of ERASE was, very frequently, followed immediately by a call of DE or DEVAL so that space given free by ERASE was reclaimed instantaneously by DE or DEVAL for storing names. The name list is, therefore, organized as a stack of which the top elements are, upon ERASE, not immediately given back to free space but only after a while so that DE or DEVAL get a chance to reuse them, without the aid of STORE. The effect of this implementation was a 15 percent increase in speed. As only two names at most are used for this purpose, the cost in space is negligible.

Concerning the other procedures of the run-time system, it suffices to make the following remarks:
For each of the formula enquiries: lhs of, rhs of and el ... of there are two run-time procedures: LHS, ARLHS, RHS, ARRHS, EL and AREL. The "AR" version appears when the formula enquiry is situated in an arithmetic or Boolean expression in which case we are interested in integral values; the version without "AR" appears in a formula expression, or inside a formula enquiry only (e.g. "rhs of" in "lhs of rhs of f"). Each version has its own error diagnostics.

Observe that LHS, RHS and EL give error messages if the type is such that the result would be a constant. Consider the following example:
f:= constant(5,123,321); h:= lhs of f;
g:= if random < .5 then lhs of f else rhs of f",
Where f,g and h are variables of type formula.

The procedures LHS and RHS give error messages which are somewhat crude; let us investigate the alternatives. It is evident that to h and g references to formula values must be assigned. There are thus two possibilities: either a new formula value in created with as value 123 or 321 and the reference to this formula value is given as value to h and g, or the value of f itself is given to h and g. The latter possibility gives problems as it is impossible to see later on from the value of g which of the two choices has been made lhs or rhs. The first possibility has the draw-back that values are created, not under the direct responsibility of the programmer, with types chosen rather arbitrarily. For the above case the effect would be the same as:

```
"f:= constant(5,123,321); h:= constant(5,123,0);
g:= if random < .5 then constant(5,123,0) else constant(5,0,321)"
```

For standard procedures the draw-back would be very serious as these procedures also have to cope with such unintentionally created formula values.

Another alternative would be to give no error message anyhow. This is, however, contrary to the general principle to warn the programmer each time something is unclear and probably due to a programming error.

There being no real alternatives, we were forced to make LHS, RHS and EL as they are, with the somewhat awkward consequence that simple constructions like:

```
"begin  formula l,r; l:= lhs of f; r:= rhs of f;
        if dyadic f then begin ... l ... l ... r ... r end else
        if monadic f then begin ... r ... r ... end else
        if constant f then begin ... end
        ...
end" are in general not admitted.
```

Rowadic and polyadic structures are built up by means of monadic and dyadic structures, respectively. It is thus meaningful to allow the use of the lhs and rhs operators also for these structures. The i-th element can be reached by the following algorithm:

```
if i < r v i > lhs of f then goto no element;
g:= rhs of f;
for i:= i-1 while i > 0 do g:= rhs of g;
if polyadic f then g:= lhs of g else i:= lhs of g;
```

The result is either that a jump is made to the label no element if the value of i is not appropriate, or that the value is delivered in g, if f is a polyadic structure and in i, if f is a rowadic structure.

It is remarked that in the "enquiry" procedures as TYPE, TYPE CAT, LHS, ... , AREL, it is not tested whether F has the value NULL since this is done automatically by the ALGOL 60 run-time system and it is very simple to reconstruct the error message from the line number.
The procedure STRING is used, together with its cooperating procedure st, to store a string as a rowadic structure with a type equal to 31. The working of these procedures is rather complicated. The translation of the string "⊀01234567⊁" is: "STRING(st(66051,st(263430,st(1800,0))),3)". The global integer "ici", used as "Jensen-variable" in a call of STORE ROW by STRING, obtains in STORE ROW the values 3, 2 and 1, respectively.
Evaluation of the first actual parameter of STRING leads to execution of st, which, when di > 1, leads to evaluation of the second actual parameter of st after a decrease of di. The first actual parameter of STRING is peeled off a number of times indicated by the value of di. The first element stored by STORE ROW is, therefore, 1800, the second one is 263430 and the third and last one is 66051.

4.8. The run-time system

We now reproduce the run-time system.

```
begin comment Most simple run-time system for efficient ABC ALGOL;
  integer max of memory,free cell,last name,gnn,snn,N,NULL,
  constant,monadic,dyadic,polyadic,rowadic,t23,t18,t17,linectr,
  di,kn,prev last name,gci,max gnn,nr of names,nr of values;
  Boolean protect; real time1;
  integer array linebuf[1:10];

  procedure ERR(s); string s;
  begin NLCR; PRINTTEXT(s);
    if linectr ≠ 0 then
    begin integer i,ub,line; ub:= linectr + 1; CARRIAGE(2);
      PRINTTEXT(⌈linenumbers at previous entries:⌉);
      for i:= linectr step -1 until 1,10 step -1 until ub do
      begin line:= linebuf[i];
        if line ≠ -1 then absfixt(4,0,line) else goto out
      end
    end;
  out: EXIT
  end ERR;

  procedure lnr(linenumber); integer linenumber;
  begin linectr:= linectr + 1;
    if linectr = 11 then linectr:= 1;
    linebuf[linectr]:= linenumber
  end lnr;

  integer procedure ERROR(b,s); boolean b; string s;
  begin if b then ERR(s); ERROR:= 1 end;

  max of memory:= read; gci:= nr of names:= nr of values:= 0;
  time1:= time; last name:= kn:= prev last name:= 0;
  gnn:= snn:= max gnn:= 0; linectr:= 0;
  for di:= 1 step 1 until 10 do linebuf[di]:= -1;
  t23:= 2 ↑ 23; t18:= 2 ↑ 18; t17:= 2 ↑ 17; N:= -2 ↑ 16 + 1;
  constant:= 0; monadic:= 1; dyadic:= 2; polyadic:= 3;
  rowadic:= 4; NULL:= 0; protect:= false;
```

```
begin integer array C1,C2[1:max of memory];

  integer procedure STORE(t,A,B); value t,A,B; integer t,A,B;
  begin integer k,i; k:= C1[free cell];
    comment "free cell" points to the first free cell of the
    free list. The list is never empty when we enter STORE;
    C1[free cell]:= A; C2[free cell]:= t × t18 + B;
    STORE:= free cell;
    if protect then ERR(⋠protection error⋡);
    if t = 64 then nr of names:= nr of names + 1 else
      nr of values:= nr of values + 1;

    if k = 0 then
    begin comment A garbage collection is necessary now;
      procedure TRACE(F); value F; integer F;
    L: if F ≤ 0 then else
      begin i:= C2[F]; if i ≥ 0 then
        begin integer t; t:= i ÷ t23; C2[F]:= -i - 1;
          if t ≥ polyadic then t:= monadic;
          if t = monadic then
          begin F:= i - i ÷ t18 × t18; goto L end else
          if t = dyadic then
          begin t:= i; TRACE(C1[F]); F:= t - t ÷ t18 × t18;
            goto L
      end end end TRACE;
      TRACE(free cell); TRACE(last name); gci:= gci + 1;
      for i:= max of memory step -1 until 1 do
      if C2[i] < 0 then C2[i]:= -C2[i] - 1 else
      begin C1[i]:= k; k:= i end;
      if k = 0 then ERR(⋠no space left⋡)
    end garbage collection;
    free cell:= k
  end STORE;

  integer procedure STORE CONST(t,l,r); value t,l,r; integer t,l,r;
  begin if t < 0 ∨ t > 31 then ERR(⋠type in const not appropriate⋡);
    if abs(r) ≥ t17 - 1 then ERR(⋠rhs value in const too large⋡);
    STORE CONST:= STORE(t,l,(if r < 0 then t17 else 0) + abs(r))
```

```
end STORE CONST;

integer procedure STORE MONADIC(t,A,B); value t,A,B; integer t,A,B;
begin if t < 0 v t > 31 then ERR(⌈type in monadic not appropriate⌉);
  STORE MONADIC:= STORE(t + 32,A,abs(B))
end STORE MONADIC;

integer procedure STORE DYADIC(t,A,B); value t; integer t,A,B;
begin integer A1,B1,fnn; fnn:= gnn;
  if t < 0 v t > 31 then ERR(⌈type in dyadic not appropriate⌉);
  DEVAL(A1,A); B1:= B; ERASE(fnn);
  STORE DYADIC:= STORE(t + 64,abs(A1),abs(B1))
end STORE DYADIC;

comment  It  is not necessary  to test whether  the references  to be stored
are references to values  and not references to names.  The reason is that a
formula expression always delivers a reference to a value;
integer procedure STORE ROW(c,t,i,nr,Ai); value c,t,nr;
  integer c,t,i,nr,Ai;
begin integer row,fnn; fnn:= gnn;
  if t < 0 v t > 31 then ERR(⌈type not appropriate in row⌉);
  if nr < 0 then ERR(⌈nr of elements is negative⌉);
  t:= t + c; DE(row,NULL);
  for i:= nr step -1 until 1 do
  ASSIGN(row,STORE(160 - c,if c = 96 then abs(Ai) else Ai,
        if i = nr then 0 else V(row)));
  ASSIGN(row,STORE(t,nr,if nr = 0 then 0 else V(row)));
  STORE ROW:= V(row); ERASE(fnn)
end STORE ROW;

procedure DE(f,E); value E; integer f,E;
begin E:= abs(E); if kn > 1 then
  begin C1[prev last name]:= E; f:= prev last name; kn:= 1; end else
  if kn > 0 then
  begin C1[last name]:= E; f:= last name; kn:= 0 end else
  begin prev last name:= last name;
    last name:= f:= STORE(64,E,last name)
  end; gnn:= gnn + 1; f:= N - f; if max gnn < gnn then max gnn:= gnn
```

```
end DE;

procedure DEVAL(F,E); value E; integer  F,E;
begin F:= E; if E > 0 then
  begin F:= - E; if kn > 1 then
    begin C1[prev last name]:= E; kn:= 1; end else
    if kn > 0 then
    begin C1[last name]:= E; kn:= 0 end else
    begin prev last name:= last name;
      last name:= STORE(64,E,last name)
    end;
    gnn:= gnn + 1; if max gnn < gnn then max gnn:= gnn
end end DEVAL;

procedure ERASE(n); value n; integer n;
begin gnn:= gnn - n - 1; if gnn ≥ 0 then
  begin for gnn:=gnn - 1 while gnn > - kn do
    begin C1[last name]:= free cell; free cell:= last name;
      last name:= prev last name; if last name ≠ 0 then
      begin prev last name:= C2[last name]; prev last name:=
        prev last name - prev last name : t18 × t18
    end end; kn:= kn + gnn + 2;
    if kn = 2 then C1[prev last name]:= 0; C1[last name]:= 0
  end; gnn:= n
end ERASE;

integer procedure ASSIGN(f,E); value f,E; integer f,E;
begin if f ≥ N then ERR(⊰assignment to a value⊱);
  if E = NULL then ERR(⊰undef rhs in ass stat⊱);
  ASSIGN:= C1[N - f]:= abs(E)
end ASSIGN;

integer procedure V(f); value f; integer f;
begin f:= V:= C1[N - f];
  if f = NULL then ERR(⊰val of undef var in expr⊱)
end V;

integer procedure VN(f); value f; integer f;
```

```
begin if f < N then f:= C1[N - f];
  if f = NULL then ERR(⊀val of undef name var in expr⊁); VN:= f
end VN;

integer procedure TYPE(F); value F; integer F;
TYPE:= remainder(C2[abs(F)] : t18,32);

integer procedure TYPE CAT(F); value F; integer F;
TYPE CAT:= C2[abs(F)] : t23;

integer procedure LENGTH(F); value F; integer F;
begin integer t; t:= C2[abs(F)] : t23;
  if t < polyadic then ERR(⊀type in length not appropriate⊁);
  LENGTH:= C1[abs(F)]
end LENGTH;

integer procedure LHS(F); value F; integer F;
begin if C2[abs(F)] : t23 = constant then ERR(
  ⊀type in lhs not appropriate⊁); LHS:= sign(F) × C1[abs(F)]
end LHS;

integer procedure ARLHS(F); value F; integer F;
  ARLHS:= C1[abs(F)];

integer procedure RHS(F); value F; integer F;
begin integer r; r:= C2[abs(F)];
  if r : t23 = constant then ERR(
  ⊀type in rhs not appropriate⊁);
  RHS:= sign(F) × (r - r : t18 × t18)
end RHS;

integer procedure ARRHS(F); value F; integer F;
begin integer r; r:= C2[abs(F)];
  r:= r - r : t18 × t18;
  ARRHS:= if r ≥ t17 then t17 - r else r
end ARRHS;

integer procedure EL(i,F); value i,F; integer i,F;
```

```
begin integer s; s:= sign(F); F:= abs(F);
  if C2[F] : t23 ≠ polyadic then ERR(⊰type in el not appropriate⊱);
  if i < 1 v i > C1[F] then ERR(⊰index in el not appropriate⊱);
  for i:= i step -1 until 1 do
  begin F:= C2[F]; F:= F - F : t18 × t18 end;
  EL:= s × C1[F]
end EL;

integer procedure AREL(i,F); value i,F; integer i,F;
begin integer t; F:= abs(F); t:= C2[F] : t23;
  if t < polyadic then ERR(⊰type in arel not appropriate⊱);
  if i < 1 v i > C1[F] then ERR(⊰index in arel not appropriate⊱);
  for i:= i step -1 until 1 do
  begin F:= C2[F]; F:= F - F : t18 × t18 end;
  AREL:= C1[F]
end AREL;

integer procedure STRING(ST,m); value m; integer ST,m;
STRING:= STORE ROW(128,31,di,m,ST);
integer procedure st(head,tail); value head; integer head,tail;
if di = 1 then st:= head else
begin di:= di - 1; st:= tail; di:= di + 1 end;

procedure Zreplace(f,left,g);
value f,left,g;
integer f,g; Boolean left;
if left then C1[f]:= g else
begin integer t; t:= C2[f] : t18;
  C2[f]:= t × t18 + g
end;

for di:= max of memory step -1 until 1 do
begin C2[di]:= 0; C1[di]:= di + 1 end;
C1[max of memory]:= 0; free cell:= 1;
```

The compiled ABC ALGOL program should be placed here followed by two end's and an integer defining the length of the arrays C1 and C2.

5. The compiling process

The ABC ALGOL compiler is, in all details, reproduced in chapter 6 as an ALGOL 60 program. The description is amply provided with comment, the identifiers are chosen as appropriately as possible (we hope) and the structure of the description itself, from top to bottom, should lend itself most conveniently for easy reading and studying.
As an aid, section 5.3 contains the alphabetic listing of the identifiers and the places where they are called in the compiler.
Albeit all these nice features, we consider it necessary to make some remarks on the compiling process.
From the first ABC ALGOL compiler, running in 1969, until the final one, we aimed at a two-pass clearly written compiler using a top-to-bottom syntax-analyzing technique. The features provided for in the final version and not contained in the first version range from error-recovery and packing of symbols to static-scope checking and optimality analyses.

5.1. The overall strategy

There is a hierarchy of procedures for the reading process: the highest procedure being "envelope of block" reading a complete program with heavy syntactical means. The lowest procedure being the MC standard procedure RESYM reading one symbol from input tape and delivering the internal representation of this symbol. In between we have a succession of other procedures. We may have e.g. the following sequence of procedures, where each procedure, except for the first one, has been called by its predecessor:

> envelope of block, block or statement, statement, unconditional statement, envelope of block, block or statement, declaration, procedure declaration, envelope of block, block or statement, statement, conditional statement, statement, unconditional statement, assignment statement, formula expression, simple formula expression, primary, RE, READ synt unit, NS, RESYM.

This calling sequence is pertinent to the following example:

"begin real x; begin formula procedure p; begin if x < 0 then
p:= 5.4 +"

5.1.1. The lexical scan

The last four procedures are designed for what might be called the *lexical scan*; in fact, the procedure RE reads one so-called syntactical unit (su), using READ synt unit, while the first procedures are all designed for reading high syntactical structures. These syntactical structures are built up from su's as building stones. In the preceding example the respective su's are:

begin symbol, real symbol, identifier, semicolon symbol,
begin symbol, formula symbol, procedure symbol,identifier,
semicolon symbol, begin symbol, if symbol, identifier,
smaller than symbol, integral number, then symbol,
identifier, becomes symbol, real number, plus symbol.

The compiler is not interested in the value of a number; it only wants to know whether it is a real or an integral number. Identifiers are of course objects of more interest; if they are read, two global variables: "nr of ident" and "ptr to first letgit" obtain as values the identification number of the identifier and the entry in the text array where the identifier occurs, respectively. No automatic look-up in the name list is done; this should be asked for explicitly by a call of "Search for identifier(n)", which, if this procedure delivers the value *true*, gives in n the entry in the name list where more information about the identifier is stored.
For a language like ABC ALGOL it is very desirable to look one su ahead. This makes constructions like:

'L:", "L:=", "L(" or "L["

easily recognizable. Therefore, we have two global variables: "synt unit", which designates the su under treatment now and "next synt unit", which designates the su to be treated next. This is reflected in the procedure RE where we have the statements: "synt unit:= next synt unit; next synt unit:= READ synt unit". Besides the procedure for reading a su, we have a printing and punching procedure "PR synt unit" which prints and punches the symbols of the su under treatment. Moreover, there is a procedure which combines both actions: "PR and RE" for printing and punching the su under treatment and for reading a new su.
The above-mentioned procedures are described in section 6.6.2.

5.1.2. The syntactical scans

The two-pass compiler is written as a single-pass compiler in that there exists only one procedure for each syntactical structure. Two Boolean variables: "first scan" and "second scan" serve as a switch. Except at the place where they change values, it is everywhere true that first scan $\equiv \neg$ second scan. The rough structure of the syntax-translating procedures is the following:

```
"procedure envelope of block;
 begin integer local integers; Boolean local Booleans;
       procedure block or statement; ...;
       procedure declaration; ...;
       procedure statement; ...;
       procedure expression; ...;
       procedure block and name list procedures; ...;
       block or statement
 end"
```

Each block, including the procedure-body statement promoted to become a block, is translated by means of a call of "envelope of block", which then calls the other syntax-translating procedures. The effect is that all the syntax-translating procedures have the same environment consisting of the local integers and the local Booleans. If an inner block has to be translated, a (recursive) call of 'envelope of block" preserves the current local variables and introduces automatically a new set of local variables. Finishing the inner block, we automatically get back our original local variables. Between the first and the second scan the values of these variables are retained in the information list.

5.1.3. The information list

In the information list, the integer array "contents of", information cells are stored from bottom to top while identifiers, i.e. the characters forming the identifiers, are stored from top to bottom.
During the first scan, identifiers, occurring in the text, are replaced by unique numbers in order to reduce the size of the text array and to speed up the search-for-identifier- process. The lexical scan delivers for each identifier read two pieces of information: "nr of ident" identifying the identifier and "ptr to first letgit" defining the place in the text array where the identifier is put.
For almost all identifiers, declared in the program, an information cell (ic) is created containing the relevant information about the identifier: "nr of ident" for the identification, "ptr to first letgit" denoting the place in the text array of the defining occurrence, the type, other information as number of parameters or dimension and space for link pointers.
Ic's are, moreover, provided for block-begin's and block-end's; firstly in order to save information pertinent to a block and secondly for the search-for-identifier-process where the block structure has to be taken into account with. For the latter purpose, the ic's of identifiers declared in a block are physically placed between the ic's of the block-begin and the block-end. (See also section 5.2.3.5).
There are three lists formed by identifier ic's:

- a list of integers (to which a formula procedure identifier may be attached);
- a list of formula variables (to which parameters may be attached);
- a list of formula array segments.

Moreover, the begin- and end-ic's are linked.
The following example shows the structure of the information cells, in particular with respect to the several lists.

```
begin formula f1,f2; integer i1,i2; real r1,r2;
      Boolean b1,b2;
      formula procedure fp(par1,par2,par3); value par2;
              formula par1,par2; real par3;
      begin   integer i1,i2; formula f1,f2;
              formula array fa1,fa2[1:2],fa3,fa4[1:2,3:4];
              integer i3,i4; formula f3,f4;
              formula array fa5[1:2,1:2];
              ...
      end;
      procedure pr;
      begin   integer i1,i2,i3; ...
              begin   formula g1,g2; ... end
      end;
      ...
end
```

Now follows the information list consisting of the ic's in the order indicated by the order of the ic's in the following picture from left to right and from top to bottom.

begin f1 f2

i1 i2

r1 r2

fp

begin par1 par2 par3

i1 i2

f1 f2

fa1 fa2 fa3 fa4

i3 i4

f3 f4

fa5

end

pr

begin i1 i2 i3

begin g1 g2

end

end

end

Note that during the second scan, the order of the list of formula identifiers is reversed in order to treat the declarations appropriately.

Identifiers of quantities other than of arithmetic or formula type are considered by the compiler as non-interesting and no ic's are created for them.

By means of a call of "STORE into information list" and "st" an ic, consisting of the n items: item1, item2, ..., item n, is created:

```
"p:= STORE into information list(st(st(...st(0,
     item1),
     item2),
     ...
     item n))".
```

To get back the i-th item we simply have to ask for the value of:

```
"contents of[p+i]".
```

The format of the identifier ic's is the following:

	Arithmetic but non-integer	integer	formula-array segment	formula	procedure
item1	nr of ident	same	same	same	same
item2	ptr to first letgit	same	same	same	same
item3	type	same	same	same	same
item4		link	link	link	link
item5			dimension		nr of param
item6					form proc

Identifiers of formula arrays not occurring as the first identifier in a formula-array segment have an ic not containing the link.

Item6 for procedures contains information relevant for the protect mechanism of formula procedures.

The _begin_-ic contains 9 items, the first one, identifying the ic as a _begin_-ic, being equal to -begin symbol. The end-ic contains 2 items, the first one being equal to -end symbol. As the values of "nr of ident" are normally positive, an easy distinction between identifier ic's and _begin_ and _end_ ic's has been achieved. (See, however, the discussion on operator identifiers in section 5.1.6).

5.1.4. The reading mechanism

During the first scan, the text is read from input tape and, after some more or less trivial changes, put into the text array. The changes concern: deletion of comment, change of ")text:(" into a comma, replacement of word delimiters by one-symbol-codes and substitution of identifiers by unique codes.

During the second scan, the text is almost entirely read in a linear fashion. Sometimes, use is made of a back-track or a look-forward technique. For example in the case of "a[1]:= b[1]:= c[1] + d[1]", the ":=" symbol after "b[1]" is read before "b[1]" is treated as the lhs part of an assignment statement. Use is made of three procedures:

1. "SAVE reading ptrs", which puts all the reading pointers, as "synt unit", "next synt unit", "ptr of text" or "line number" on a stack and stops the counting of _begin_'s, _end_'s and lines.
2. "SKIP text until", which skips text until a Boolean condition, being its actual parameter, is fulfilled, taking into account the brackets structure. Example: "SKIP text until (synt unit = comma symbol)" leads, for the following text (part of an array declaration):

   'x[1,2]: p(1,2,3) × y[q(1,2)],27]; ...",

   to the skipping of the text until: ",27]; ...".
3. "RESET reading ptrs", which resets the reading pointers to the values being placed on the stack by the first procedure. If the stack is empty, it reenables the counting of _begin_'s, _end_'s and lines.

For the multiple assignment above, the compiler investigates the situation as follows (section 6.4.4):

```
"L:     treat lhs part of assignment statement;
        if synt unit = identifier then
        begin if next synt unit = becomes symbol then goto L else
          if next synt unit = sub symbol then
          begin SAVE reading ptrs; RE; RE;
            SKIP text until (synt unit = bus symbol);
            if next synt unit = becomes symbol then
            begin RESET reading ptrs; goto L end else
            RESET reading ptrs
        end end;
        treat rhs part of assignment statement"
```

In a situation like: "formula f,g:= x+y × z;
formula array fa[1:lhs of h]",
it is necessary to treat the expressions "x+y × z","1" and "lhs of h" not at the places where they occur in the text. In the information cells of g and of fa, the pointers to their defining declarations in the text are put so that we can find back the text of the expressions. For this purpose we use the procedure "SET reading ptrs on". If the ic of g has the address "a",

"SET reading ptrs on (contents of[a+2])"

has as effect that after two calls of RE, to skip "g" and ":=", the expression "x+y × z" can be read.

5.1.5. Error detection and error recovery

The errors in an ABC ALGOL program can be divided into errors discoverable by the compiler (almost all syntactic errors) and errors becoming manifest during the running of the translated program only. Besides these errors, there is an unknown category of non-detectable errors which we will not discuss. Let us call a piece of text, which has the pretention being an ABC ALGOL program, a Pretentious ABC ALGOL Program (PAAP). We assume that a PAAP consists of syntactic units.

Apart from some exceptions, only fatal errors are detected in reading syntactic units. Fatal errors are the result of overflow of tables and arrays and lead to a discontinuation of the compile process. The exceptions are trivial errors as in: "for comment" or "constant ↑". Note that there is an important syntactic unit: "unknown symbol".
If a PAAP does not begin with a begin, this is corrected by putting a begin in front of it. After such a possible correction each PAAP has a blockstructure and its first syntactic unit is begin. The end of the PAAP must explicitly be indicated by a special symbol for which we have taken ";". If this symbol is part of a string or of comment, it automatically serves for as many right string quotes as are necessary or as a semicolon; moreover, it serves for as many end's that are missing.
If a PAAP has a blockstructure such that the symbol after the final end is not equal to ";", the superfluous symbols are deleted. The result is a PAAP of which the blockstructure obeys the laws of an ABC ALGOL program.
One level deeper, we find declarations and statements. For the time being we will consider a compound statement to be a block. Statements and declarations are separated by semicolon's.
Except for procedure declarations, declarations do not contain semicolon's. Statements which are no blocks do not contain semicolon's. Apart from procedure declarations and blocks we thus arrive at the following error-recovery scheme: let the compiler read a pretentious declaration or a pretentious statement; there are two possibilities: either during this process a semicolon or an end is encountered, upon which as many symbols as are necessary are virtually supplied to correct the PAAP (missing right brackets e.g.), or the semicolon or end is not encountered. In the latter case we may delete all symbols possibly present before the semicolon or end. The first case needs some more considerations.
Firstly, it is necessary that the compiler does not bypass a semicolon or end without being aware of this fact. To this end, a Boolean variable: "no semicolon or begin end allowed" is being used, which, when true, prohibits actual reading (i.e. a call for RE does not result in any change of the reading pointers) if synt unit is equal to a semicolon, a begin or an end. Only when its value is false, the reading process continues also for a semicolon, a begin or an end. There are special procedures: " RE semicolon", "RE begin" and "RE end" for this purpose.

Secondly, suppose that the compiler requires a certain symbol to be present in the PAAP, but it is not there. That symbol is then virtually (= not in the text array) inserted; moreover, the reading process is halted for exactly one synt unit. This means that the next call of RE has no effect on the value of the reading pointers, except that the switch, in the form of the Boolean variable "delay one RE", is turned off.

Example: "(a × (b+c[(a+c[d × (e+f)] + c[g];" is corrected into:

"(a × (b+c[(a+c[d × (e+f)])] + c[g]));"

The interesting feature of this scheme is that it fixes quite a number of errors; while, on the other hand, the apparatus used in the compiler is minimal: some lines of program in the reading mechanism and explicit calls of the procedure "CHECK", every time the compiler requires a certain symbol to be present.

Besides calls of "CHECK", we encounter calls of the procedure "ERR" within the syntax-analyzing procedures. These calls are invoked when errors of more semantic nature are encountered, such as: identifier not declared, too much parameters or type of actual parameter not in correspondence with type of formal parameter. These errors can easily be repaired such that the compiling process may go on.

For procedure declarations, the situation is a little bit different: we have to be aware of some extra semicolons.

A statement, in the form of a block, can not give any troubles as the block structure itself is error free (which does not mean that the compiler necessarily has the same opinion about the block structure as the programmer); the block then automatically decomposes into declarations and statements, for which the above considerations hold.

An error message has to include a line number. For the first scan, this is no problem at all, except for the fact that one syntactic unit is read in advance. During the second scan, however, the situation is more complicated due to actions in which the reading pointers are changed; in particular when they are changed by means of "SET reading ptrs on" to point to an arbitrary piece of text. The line number of that piece of text has to be available. It is part of the information contained in "ptr of first letgit".

Also error messages during execution of the translated ABC ALGOL program have to be provided with a line number. Each block-begin is, therefore, translated such that a call of "LNR(...)" is inserted. The actual parameter is the line number of the corresponding begin. The procedure LNR of the running system has a cyclic buffer, so that upon an errormessage the line numbers of the last 10 activated blocks can be printed.

On the proper termination of the compilation of a PAAP, we can make the following remarks. Firstly it can be seen (with some pains) that the compiler behaves neatly for a correct ABC ALGOL program. Secondly, we have described the way the compiler corrects the PAAP. The blockstructure is certainly correct. The form of the statements could be incorrect. Assuming that the statement-analyzing procedures terminate, it can simply be seen that the block procedure terminates. If a statement procedure is called anyhow, the first syntactic unit must fit and the next syntactic unit is read. The only possible danger is an infinite loop either on account of a "L: ... goto L" situation, which is not possible unless we had an infinity of comma's, or on account of an infinite recursion.
In the latter case no danger is present if each time a procedure is called at least one syntactic unit is actually read. The danger occurs when virtual syntactic units are created. This is only possible in a "CHECK" situation where the syntactic unit is not what it should be. A situation like: "...; CHECK(... symbol); RE; proc; ..." with: "procedure proc; begin CHECK(... symbol); RE; ... end" does not occur, however.

5.1.6. Identifiers

Identifiers are picked up from the text and, if not already present, placed in an alphabetically ordered binary tree situated in the higher part of the information list. The organisation is as follows: The cell containing the identifier consists of the following words:

- a number of words containing the letters and digits, packed four in a word, of which the last word is negative;
- a word containing a pointer to the tree containing identifiers following, in alphabetical order, this identifier;

. a word containing a pointer to the tree containing identifiers preceding, in alphabetical order, this identifier;

. a word containing the number which internally represents the identifier (nr of ident).

To find the letters of an identifier with a given internal number as code, a second small sized array is used containing the address of the cell in the information list. The letters in the text array are replaced by the internal code. Possible occurring new line symbols are taken care of by inserting them afterwards. Due to the way symbols are packed in the text array, i.e. 3 in one X8 word, the internal code of any symbol may not exceed 399. A consequence is that if more than 218 different identifiers occur, another strategy has to be followed for the replacement (There are roughly 180 different symbols to be coded, including the word delimiters). The identifier then occupies three symbols: s1, s2 and s3, the first one being equal to 399, the identifying number being equal to: s2 × 400 + s3.

The identifiers of standard procedures are all placed, together with information on type, number of parameters and possible appearance in upper and lower case, in alphabetical order in a string. The search technique uses the alphabetical order.

It is interesting to observe that the search technique for word delimiters uses a hash-code technique in view of the fact that we wanted the word delimiters to be a stored in a special, non-alphabetical, order (Moreover,we wanted to apply three techniques: binary tree, alphabetical order and hash code).

The treatment of operator identifiers, as "dyadic+" or "constant-", needs some extra clarification. The underlying principle is to handle them, despite their appearance, as much as possible as ordinary identifiers. That means that at a low level (RE) they have to be recognized, at the higher level they are only taken into account, if the compiler is explicitly aware of them (e.g. after "formula procedure" or at the lhs of an assignment statement). Normally an operator identifier may not occur at a place where an ordinary identifier may occur. In order to prevent explicit testing upon the absence of an operator identifier, the following simple method is used. In normal cases each time an identifier occurs it is being printed by means of "PR and RE" or "PR synt unit". Within these low-level procedures an error message is given when an operator identifier has to be printed. Any time

the compiler is aware of an operator identifier which is to be printed, a special printing procedure "PR operator" is called. As the printed version of the operator identifier is rather peculiar ("S" and "MI" for the example above), this special treatment had to be given anyhow. Now we have achieved that we are not hindered in the normal case with operator identifiers.
To let operator identifiers behave as much as possible as normal identifiers, the values of "nr of ident" and "ptr to first letgit" have to be defined. The latter one can point, as usual, to the place in the text array where the formula procedure is defined. The first one is made negative, equalling:

"- (synt unit × 1024 + next synt unit)"

assuming that "synt unit" denotes the internal representation of <u>constant</u>, <u>monadic</u>, <u>dyadic</u>, <u>rowadic</u> or <u>polyadic</u> and "next synt unit" the operator. All identifiers being identified by means of the value of "nr of ident", it follows that all operator identifiers are different from normal identifiers, but we can use the normal apparatus to treat them; such as "Search for identifier" or "procedure declaration". In the latter procedure an exhaustive test is being performed for formula procedures defining operators with respect to number and type of the formal parameters.

The translated identifiers should be such that a clash with existing identifiers of the procedures and variables of the running system and with standard identifiers is impossible, while, at the same time, the ABC ALGOL programmer should be as free as possible in his choice. Therefore, each identifier (except those of standard procedures) is translated by putting infront of it either a "Z" or a "Y". The "Y" is put if an extra local variable is necessary to be used simultaneously with the original identifier (the local integer for a formula procedure identifier or the local variable for a called-by-value formula parameter). The "Z" is used in the normal case of a direct translation.

5.2. <u>The compiler from begin to end</u>

In this section the several parts of the compiler will be discussed. The number 5.2.i of each sub section refers to the number 6.i of the corresponding section of the next chapter, where the ALGOL 60 text of the compiler is given. As this text itself is amply provided with comment, it suffices to make a few remarks only.

5.2.1. Prerequisites for translation process

Local to the procedure "envelope of block" are all the syntax- analyzing procedures from section 6.2 to 6.5, and the following variables, the meaning and usage of which will now be described.
The parameter "procedure body" indicates, when equal to zero, that a block or a compound statement has to be treated. The latter case can occur only when the program itself is a compound statement. When not equal to zero, a procedure body has to be treated and the information cell for the block-begin of this procedure body has been created already (within the procedure "procedure declaration"; see section 6.3.4). The value of "procedure body" defines the information cell of that block-begin.
The integers "ptr to integer", "ptr to formula" and "ptr to array segment" are used to point to the last cells of linked lists of information cells for identifiers. The linking process always has the following form:

```
 "initialization: ptr to something:= 0;
      ...
  create a new something:
  ptr to something:= STORE into information list(st(... st(... st(0,
              ...),
              ptr to something),
              ...))".
```

The list of formula identifiers, e.g., is created by several calls of "formula declaration", one for each occurrence of the declarator formula and by internal loops within "formula declaration", one for each identifier in a formula list (as in "formula f1, f2, f3;"). It is the task of "formula declaration" to go on with extending the already existing list of formula identifiers.
The integers "nr of array segments" and "max dimension in array declarations" will be discussed in section 5.2.3.2.

The Boolean "formula block" indicates whether the block contains declarations involving the creation of formula names such that, upon execution, the number of names changes. Its value determines whether the statement "ERASE(fnn)" will appear at the end of the block and whether a jump out of the block (see section 4.6) is possibly dangerous.
The Boolean "dangerous inner block" determines whether the block contains an inner block, while the latter one contains itself a dangerous inner block or a goto statement and formula declarations (see section 4.6).
The Boolean "snn necessary" determines whether in the declaration of integer variables, also the variable "snn" should be declared followed by the statement "snn:= gnn" after the treatment of the declarations (section 4.6).
The Booleans "block contains labels" and "block contains gotos" are self-clarifying.
The Boolean "proc id ass stat" is used to determine whether the last statement of the block (except for some dummy statements at the end) is an assignment statement, the left-hand part of which contains the procedure identifier of the formula procedure of which this block is the procedure body. If such is the case, several safety measures can be skipped. The process is rather simple: before translating a statement the variable gets the value <u>false</u>; only the procedure assignment statement can change this value and it does so when it encounters the procedure identifier.
The Boolean "interested in proc id" indicates whether the block is the procedure body of a formula procedure.
The Boolean "block" determines whether we have a block (or compound statement) or a statement.

The only statement of the procedure body of "envelope of block" is: "block or statement".

5.2.2. Translation of a block

The rough structure is: translate the declarations (taking into account that each declaration is followed by a semicolon) and translate the statements.

5.2.3. Translation of a declaration

Concerning the item type in the information cell of an identifier we remark that it normally consists of two parts:

- the type part, tp, which defines the "plain" type
  e.g. tp = real symbol × 2 ↑ 15, for a _real_
  tp = formula symbol × 2 ↑ 15, for a _formula_
  tp = formula symbol × 2 ↑ 15 + array symbol × 2 ↑ 5,
  for a _formula_ _array_
- the mode part, mp, which gives some more information;
  e.g. it may be equal to: "declared as value" or "declared as name", "specified as value" or "specified as name", "with local" or "without local".
  These are all small (< 32) integers.

The type item itself equals tp + mp.

5.2.3.1. Translation of a formula declaration

The procedure "Declare integers" reverses the list of formula variables, such that the procedure "Introduce names for formulae" produces the explicit declarations with "DE" and "DEVAL" in the right order; i.e. in the order written down by the programmer. With the effect that "_formula_ x = ∢x≻,f:= x;" is executed nicely. Note that the same declaration in the other order would lead to a run-time error message. In order to give that error message, "Zx" and "Zf" get the initial value "NULL". The necessary code is produced by "Introduce names for formulae".

5.2.3.2. Translation of a formula-array declaration

A formula-array declaration: "_formula_ _array_ fa[1:n]", roughly has to be translated into: "_integer_ _array_ FA[1:N]", followed by:
"_for_ i:= 1 _step_ 1 _until_ N _do_ DE(FA[i],NULL)".
However, the lower- and upper-bound expressions may be evaluated only once. We, therefore, arrive at the introduction of a surrounding block in which the expressions are evaluated.
The following program is then translated correctly:

```
begin   integer procedure n(f); value f; formula f;
        begin   f:= der(f,x) + der(f,y); n:= degree(f) end;
        ...
        begin   formula array fa[1:n(g)]; formula h; ... end
end
```

the translation of the second block being:

```
        begin   integer l,u,i; l:= 1; u:= Zn(Zg);
                begin   integer array Zfa[l:u];
                        integer Zh,fnn; fnn:= gnn;
                        DE(Zh,NULL);
                        for i:= 1 step 1 until u do DE(Zfa[i],NULL);
                        ...
                ; ERASE(fnn) end
        end
```

Consider now the above program in which the fourth line is replaced by:

```
        procedure proc(par); value par; formula par;
        begin formula array fa[1:n(par)]; formula h; ... end
```

The treatment of "par" being performed together with the treatment of "h", troubles arise concerning the proper translation of "n(par)". In fact, the statement "u:= Zn(Zpar)" would be executed too early. It is for this reason that the procedure "arraybounds" changes the list of identifiers to be declared and for which names should be introduced by means of DE or DEVAL. In fact the list is broken into two lists: the first one containing identifiers of formal parameters specified as formula and called by value, the second one containing the rest of the identifiers, i.e. the identifiers declared as formula variables.
Before the arraybounds are computed the identifiers of the first list are treated, by declaring local variables and introducing names by means of DEVAL. The original list of identifiers is then changed such that it becomes the second list, which is treated when the declarations of the procedure body are treated.

If several lower- and upper-bounds occur they are identified as "low i c j" and "up i c j", where i and j stand for natural numbers. To produce the translation, use is made of the integers: "max dimension in array declarations" and "nr of array segments", In the procedures "arraybounds", "Declare formula arrays" and "Introduce names for formula array elements" a rather peculiar procedure "PRsn" outputs a sequence of strings alternated by numbers.

5.2.3.3. <u>Opening and closing a block</u>

In this section the apparatus for the treatment of the block-begin, the block-end and for the closely related Search- for-identifier process is given.

5.2.3.5. <u>Translation of a procedure declaration</u>

A note-worthy observation is that an information cell for a virtual block-begin is created between the procedure identifier and the procedure parameters. In behalf of the procedure "envelope of block", called to translate the procedure body, this block-begin information cell may contain pointers to a list of integers (consisting of maximally one information cell for the formula procedure identifier) and to a list of formula variables (consisting of the information cells for the formula parameters called by value).
Note that the type of the i-th parameter of a normal procedure may be found in: contents of[a+14 + ix6], where a is the pointer to the information cell of the procedure identifier; for a formal procedure, i.e. occurring as formal parameter, this type is equal to:

- 1 - contents of[contents of[a+4] - 1 + ix2]

In fact, for each parameter of a formal procedure an information cell is created containing as information the type only, as a negative quantity. A more elegant method would have been to put the information on the types in the information cell of the procedure identifier; as this information cell is created already when the identifier appears as a parameter, while the types are inspected in the specification part appearing later on, this method is unperformable.

We have to investigate if the negative values of the types overlap the negative values of other information cells i.e. -beginsymbol, -endsymbol and - ("adic symbol" × 1024 + "operator symbol") for operator identifiers. Either a negative value of a type equals -1, in which case there certainly is no overlap, or this value equals:

$$-(\text{synt unit} \times 2 \uparrow 15 + \text{next synt unit} \times 2 \uparrow 5{+}s),$$

where s equals "specified as name" +1 (=3) or "specified as value" +1 (=2). The values of "synt unit" and "next synt unit" range from 64 (+ symbol) to 399. Thus, in this case the absolute value of the type is larger than 2 ↑ 21, a value not reachable by the operator identifier whose largest absolute value is smaller than 400 × 2 ↑ 10 + 400.
After the treatment of the specification part, it is checked whether the procedure has parameters of type label, of type switch or unspecified. This information, together with the information on the presence of goto statements in the procedure body, serves to give the global variable "dangerous procedures" a value (see also section 4.6).
In behalf of the procedure "assignment statement" and in behalf of the test on the appearance of operator-identifier declarations, the global integer "in formula procedure body" gets as value the number of formula-procedure bodies declared within each other.

5.2.4. Translation of a statement

The procedure "statement" is called when a possibly labelled statement is to be translated. Depending on the environment, the translation may or may not be translated in more than one statements. This is determined by the Boolean variable "divisible".
For example, within: "i:= i+1; l:stat; ...", l:stat may be translated into: "L:ERASE(snn); STAT", however, in "do l:stat", it must be translated as "do begin L:ERASE(snn); STAT end", at least if "ERASE(snn)" is necessary.

5.2.4.1. Translation of an assignment statement

Only the assigment statement to variables of type formula and to formula-procedure identifiers is of interest here. The statement: "f1:= f2:= fp1:= f3:= fp2:= f4:= expr" is translated into:

"(<u>begin</u>) FP1:= FP2:= FP3:= ASSIGN(F1,ASSIGN(F2,ASSIGN(F3,
ASSIGN(F4,EXPR)))) (;protect:= <u>true</u>)(<u>end</u>)".

We assume that f1, f2, f3 and f4 are formula variables (possibly subscripted) and that fp1,fp2 and fp3 are formula-procedure variables. Use is made of a mechanism to link the information cells of fp1, fp2 and fp3 together. The ";protect:= <u>true</u>" is provided if, for at least one of the formula-procedure identifiers, it is true that the last statement of its procedure body is not the assignment statement to the procedure identifier, which is investigated during the first scan by means of the variable "proc id ass stat".

The procedure is also helpful for the investigations mentioned in section 4.5. It has to be detected whether in a formula- procedure body an assignment to another formula-procedure identifier occurs, in which case the "protect:= <u>false</u>" statement at the end of this procedure body may not be produced. Consider the following example:

```
1: begin ...
        formula procedure fp1;
      2: begin ...
           3: begin ... begin ... end ...
                   formula procedure fp2;
                 4: begin ...
                      5: begin ... 6: fp1:= ... end; ...
                   end; ...
              end; ...
        end; ...
  end
```

The block-end of the fourth block may not consist of "protect:= <u>false</u>", if the "protect:= <u>true</u>" is necessary for "fp1". The way this is programmed shows the strength of the block-structured information list.

First, it is determined how deep, with respect to "6", the procedure "fp1" is declared (in our case block depth = -4); next, three block-begin's are inspected to see whether their blocks are procedure bodies of formula procedures, in which case a special notice is made that "protect:= <u>false</u>" may not be produced. This notice is made by using the 6-th item of the information cell of the procedure identifier. Contents of[a+6] = 0 means: no protection needed, ≥ 1 means: protection is needed, but if = 2 then "protect:= <u>false</u>" must be suppressed. See also the procedure "Block exit" of section 6.3.3.
In order to enable the procedure "simple formula expression", i.c. the procedure "primary", to see whether a formula-procedure identifier is used, declared in a block with a block depth deeper than the block depth of the formula-procedure identifier occurring at the left-hand side, the global variable "min block depth" is being used together with the global variable "proc id ass stat".

5.2.4.2. <u>Translation of a procedure statement</u>

The procedure has a parameter "IRN necessary" which determines if the procedure is called as a function designator in a formula expression.

5.2.5. <u>Translation of an expression</u>

The compiler knows two types of expressions: "formula expression" and "other expression". The first one is investigated in all details; while the second one is scanned only to take into account the bracket structure, to pick up function designators and to treat the identifiers.
An important "other expression" is the Boolean expression: "f = g", where "f" and "g" both are formula variables.

At first sight, this seems meaningless; for the translation: "abs(V(F)) = abs(V(G))" says: are the references of f and g the same. A useful application is the case that "f" or "g" is "one" or "zero", assuming that the system is made such that only one "one" and only one "zero" are present. In that case equality of references means equality of values and then the test "f = one" (Note, not f = 1) is much faster than: "if constant f then lhs of f = r". In order to be able to use the test "lhs of f = one", the procedure ARLHS gives, in the case that the type of f is not constant or monadic, no error message but the (absolute) value of the reference instead. See also the discussion in section 2.5 on this subject.

The treatment of identifiers in a formula expression can only be understood if section 4.4 is understood.
In a formula-procedure-identifier-assignment-statement: "fp:= exp", we have to be aware of a direct transfer of a value from a variable or a function designator to "fp". To this end "proc id ass stat" is used; it is made true if the above case occurs and it remains true until, by means of a function designator or an operator (also a form of a function designator), the identifiers of the expression are shielded away from "fp". That "proc id ass stat" may reobtain its old value can be seen from the example:
"fp:= if f = one then a else if f = zero then b+c else d".
One might ask whether it would not have been easier to have "proc id ass stat" as parameter of "formula procedure" and thus of "simple formula procedure", instead of saving its value a number of times by means of local auxiliary Booleans "piass". The answer is that, originally, the compiler was not aware of special assignment statements anyhow. The above construction grew gradually. Its advantage was that calls for "formula expression" did not have to be changed.

5.2.6. The auxiliary equipment

Except for one remark on standard identifiers, we refer to the ALGOL text. The standard identifiers are stored as a string. The meaning of the + is: the identifier may occur in lower and in upper case, the meaning of the last bracket is:) real procedure,] integer procedure, > non-type procedure.

5.2.7. Initialization

The initialization consists of several parts.
Firstly, the initialization at the very beginning before any ABC ALGOL program is treated; this consists of the procedure "INITIALIZE symbols" and the filling of the array "adic op" for the adic operators.
Secondly, the initialization at the beginning of each ABC ALGOL program; this consists of the procedure "INITIALIZE inf list ptrs".
Thirdly, the initialization at each scan for each ABC ALGOL program; this consists of the procedure "INITIALIZE reading ptrs".
The whole process of how the several ABC ALGOL programs are treated and how the state variables are set can best be studied by inspecting section 6.6.5 of the main program. Note that the main loop is:

```
        fill adic op; INITIALIZE symbols;
    BEGIN: set state variables; INITIALIZE inf list ptrs;
        L: INITIALIZE reading ptrs; ...
           begin:= 1; preceding begin:= 4;
           envelope of block(0);
           if first scan then
           begin ... first scan:= false; goto L end;
           EXIT
```

and in EXIT a jump to BEGIN is made.
The same EXIT is called if the compiler decides to discontinue the further compilation through FATAL ERR.

About the initialization of the information list we want to say something more in the next section.

5.2.8. A procedure library

Each compiler system has the possibility of adding and deleting procedures in a library of precompiled procedures. Adding or deleting precompiled procedures is, as far as the run-time system is concerned, no problem at all: the precompiled procedures just have to be added or deleted by adding or deleting a few cards or a piece of papertape or by changing the file containing the run-time system.
Quite another story concerns the compiler, which should be aware of standard procedures, in particular of the procedure headings of standard procedures.

Partly, this has been implemented by supplying the compiler with a string of standard identifiers of the noninteresting procedures for input and output and similar procedures which are standard in the MC ALGOL 60 system. The information provided can be minimal: the type of the procedure (real, integer or otherwise) and the number of parameters (the parameters not being of type formula, it is not necessary to know the type of the parameters).
For general ABC ALGOL procedures, with all sorts of parameters and of any type, including even operator definitions, it is necessary to have a special catalogue. This catalogue is organized as a part of the information list in the following way:
Consider the following program:

```
        begin   ... procedure standard proc1 ...;;
                ... procedure standard proc2 ...;;
                ...
                ... procedure standard procn ...;;
                the actual program:
                begin   ... end
        end
```

The actual program may be thought of as any ABC ALGOL program using the standard procedures.
Take the information list (including the code table for the identifiers and the information-list pointers) at the moment that the compiler begins at the translation (during the first scan) of the actual program. It contains all the information needed for the translation of calls in the actual program of the standard procedures. It is also complete, in the sense that it will not be changed by the actual program, except for the information cell of the first <u>begin</u>.
The contents of the information list at this moment is called the <u>catalogue</u>.
As part of the initialization of the compiling process for an arbitrary ABC ALGOL program, the information list is filled with the catalogue. At the end of the first scan an information cell for an extra <u>end</u> is added, in order to complete the block structure of the information list. During the second scan the information list then consists of the catalogue and the information about the particular program under consideration. We thus come at the conclusion that there must at least be two different compilers:

1. The catalogue compiler, which has as input the ABC ALGOL program consisting of the headings of the standard procedures (each provided with a dummy procedure statement) and which delivers as output a string of numbers specifying the contents of the catalogue, including all the information-list pointers.
2. The normal compiler, which is provided with a string as produced by the catalogue compiler, which initializes its information list by means of information contained in this string, which treats the first scan of the ABC ALGOL program to be compiled, which inserts an extra end-information cell and which, finally, compiles the ABC ALGOL program in the second scan.
Both compilers do not differ much; the difference is the value of the boolean variable: "normal compilation".

The compiler, as reproduced in chapter 6, is a normal compiler; it can be changed into a catalogue compiler by simply changing the statement: "normal compilation := true" into: "normal compilation:= false", in section 6.6.5 line 2977.

The catalogue of this compiler has been produced by means of the following program:

```
        begin   procedure replace(f,left,g); value f,left,g;
                formula f,g; Boolean left;;
        end
```

which resulted in the procedure "catalogue symbol".

5.3. Alphabetic listing of important identifiers

In this section we give an alphabetic listing of all the identifiers which are important in the compiler. These identifiers have some global significance. Most identifiers are preceded by a number, denoting the line where they are declared. The exceptions are the MC standard procedures; they are preceded by "-".
In addition, either a list of numbers is given, denoting all lines where the identifier occurs, or "++++" is printed, indicating that the list was uninterestingly long.
A program of D. Grune and L. Meertens has been used for listing the identifiers.

-	ABSFIXT	2121	2760	2854	2857	2867	2x2870	2957
2068	additional synt unit	2068	2542	2692	2776			
1947	adic op	1947	2432	2962	2963	2964	2965	2966
		2967	2968	2969	2970	2971	2972	2973
2070	adic sym	2070	2432	2433	2x2754	2x2755		
1750	again	1748	1750	1758				
2072	alf	2042	2052	2x2056	2072	2575		
386	array bounds	68	386					
190	array declaration	157	161	190				
1245	assignment statement	1208	1245					
29	block	29	61	74	97	106	730	756
29	block contains gotos	29	632	665	701	1204		
29	block contains labels	29	632	664	700	1090		
2807	block depth	2x614	1264	1285	2x1286	1289	2x1294	1295
		1418	1441	1846	1859	2807	2834	
728	Block exit	112	728					
60	block or statement	60	1920					
-	CARRIAGE	3004						
2906	catalogue symbol	2898	2901	2906	2908			
2770	CHECK	++++						
670	Close block in information list			112	670			
2810	code table	2274	2572	2678	2679	2810	2844	2870
		2890						
1125	conditional statement	1103	1125					
2810	contents of	++++						
2372	count nr of begins	2372	2406	2417	2460	2487	2643	
28	dangerous inner block	28	666	1094				
2948	dangerous procedures	667	1060	1094	2948	2978		
149	declaration	76	149					
2805	declared as name	++++						
2805	declared as value	++++						
437	Declare formula arrays		85	437				
249	Declare integers	85	86	249	408			
2372	delay one RE	2372	2x2395	2488	2496	2517	2539	2551
		2642	2691	2775				
2072	delimiter array	2072	2106	2330	2354			
2069	delimiter array ptr	2069	2x2105	2106	2328	2x2329	2330	2335
		2354	2355					
2129	early end of program	2113	2129					
1740	elevator	1740	1741	1745	1749	1758	1759	1906
		1914						
1566	enquiry	1566	1627	1896	1897			
24	envelope of block	24	1057	1190	2987			
2758	ERR	++++						
-	EVEN	1059						
2767	FATAL ERR	++++						
2030	fill alf	2027	2030					
-	FIXT	2854						
325	formula array declaration		151	325				
28	formula block	28	94	97	270	311	663	699
		738						
225	formula declaration	153	225					
1533	formula expression	299	1368	1495	1533	1540	1810	1820
		1823	1901					
788	formula procedure	788	801	804	814	817	832	
1150	for statement	1105	1130	1150				
2071	from delimiter array	2071	2104	2108	2351	2642		

```
2933 get bits                 ++++
 789 Identifier in paramlist        789    790    795    796    875    967
                              1009  1025
2807 in formula procedure body      834   1051   1052 2x1061  1288   2807
                              2835
2827 INITIALIZE inf list ptrs       2827   2979
2637 INITIALIZE reading ptrs        2637   2981
 310 Initialize snn             92   310
1949 INITIALIZE symbols       1949  2976
  30 interested in proc id      30    78     89    731    739   1248
 473 Introduce names for formula array elements   91    473
 277 Introduce names for formulae    90    277    410
2363 is adic symbol           ++++
2366 is declarator            ++++
2379 is digit                 ++++
2382 is enquiry               ++++
2377 is lay out               ++++
2375 is letter                ++++
2385 is operator              ++++
2369 is specifier             ++++
2068 line number                63   388   2068   2403   2469   2483   2640
                              2760
2068 line number1             2068  2152   2234   2236   2403   2434   2469
                              2484  2640
2069 line number2             2069 2x2120  2121 2x2126  2152   2318   2350
                              2469  2484   2494   2640
  27 max dimension in array declarations   27    351    353    399    403
                               631   646    661    682
2804 max nr of identifiers    2272  2804   2830
2803 max of inf list          2803  2830   2837   2841   2887
2807 min block depth          1273  1285   1286   1441   1859   2807   2834
   - NEW PAGE                 2980
   - NLCR                     2121  2759   2764   2772   2774   2775 2x2850
                              2851  2857   2866 2x2869
2809 normal compilation       2809  2829   2838   2977   2991
2372 no semicolon or begin end allowed     2372   2396   2447   2449   2453
                              2454
  26 nr of array segments       26 2x334    400    631    645    660    681
2070 nr of begins             2070  2137   2404 2x2408  2409   2410   2418
                            2x2461 2x2488  2641 2x2645 2x2983 2x2985
2095 NS                       ++++
 628 Open new block in information list     62    628
1526 other expression         ++++
2690 PR and RE                ++++
2707 PR and RE begin          1191  2707
2701 PR and RE end            1201  2701
2695 PR and RE semicolon       108   166    187    205    860    871    997
                              1016  1035   1062   1198   2695
1762 PR form expr             1762  1775   1778 2x1787  1916
1794 primary                  1741  1768   1794   1795   1833 2x1893  1906
2848 PR inf cells             2848  3004
2873 print catalogue          2873  2998
2809 print information list         2809   2848   2977
2745 PR int num               ++++
   - print pos                2759  2761   2762   2763   2772
   - PRINTTEXT                2715  2726   2759   2761   2774   2850   2957
                              2958
2718 PR nlcr                  ++++
 715 procedure block entry      89   715
```

24	procedure body	24	2x25	78	79	633	634	635
		653	718	733	734	739	1250	
785	procedure declaration	152	158	162	785			
1411	procedure statement	1209	1411	1588	1844			
30	proc id ass stat	30	96	110	732	2x1133	2x1136	1167
		1200	1248	1250	1284	1440	2x1445	1502
		1534	1536	1538	1539	1770	1801	1858
2752	PR operator	++++						
2779	PR sn	++++						
2713	PR string	++++						
2721	PR sym	++++						
-	PRSYM	2122	2674	2675	2686	2722	2728	2862
2654	PR synt unit	++++						
2803	ptr of inf list	611	2245	2252	2789	2x2790	2791	2798
		2800	2x2801	2803	2836	2840	2851	2884
		2886	2x2997					
2067	ptr of text	2067	2x2075	2076	2078	2079	2x2124	2125
		2x2130	2141	2x2149	2x2155	2x2160	2x2165	2x2185
		2x2197	2233	2235	2280	2294	2318	2343
		2350	2402	2416	2x2424	2426	2427	2468
		2483	2493	2494	2x2495	2638	2x3001	
2067	ptr of text1	1653	2067	2402	2434	2468	2482	2638
		2660						
2067	ptr of text2	1653	2067	2233	2280	2293	2x2402	2416
		2468	2483	2638	2660			
27	ptr to array segment	27	64	84	94	328	333	412
		440	442	475	630	644	659	680
		757						
1260	ptr to first formula procedure identifier				1260	1263	1298	1299
		1338	1341	1353				
2803	ptr to first ident	2254	2x2275	2803	2836	2852	2884	
2070	ptr to first letgit	175	184	196	229	331	342	811
		866	2070	2235	2294	2434	2467	2482
		2641						
26	ptr to formula	26	94	227	232	256	269	280
		388	389	395	411	630	635	643
		658	679					
787	ptr to formula param	787	819	1029	1030	1055		
26	ptr to integer	26	173	177	256	2x407	411	630
		634	642	657	678			
786	ptr to integer param	786	817	818	1054			
2804	ptr to name list	2231	2256	2274	2276	2277	2278	2798
		2804	2837	2841	2856	2885	2887	
-	PUSYM	2674	2675	2686	2722	2728		
-	PUTEXT	2715	2726					
2074	put in text array	2074	2110	2132	2133	2135	2136	2138
		2x2190	2198	2282	2283	2284	2285	2288
		2289	2344	2346	2352	3002		
2394	RE	++++						
2373	reading allowed	2096	2139	2156	2373	2952	2980	
2146	READ synt unit	2146	2157	2185	2196	2200	2205	2207
		2217	2218	2232	2293	2312	2347	2355
		2358	2360	2404	2416	2429		
2457	RE begin	75	2457	2710				
2452	RE end	730	756	2452	2457	2704		
2442	RE semicolon	179	236	356	879	2442	2x2644	2698
2478	RESET reading ptrs	69	93	752	893	1334	1335	1337
		1351	1384	1385	1487	1493	1915	2478
-	RESYM	2108	2109	2953				

-	RUNOUT	3003						
2459	SAVE reading ptrs	68	83	742	883	1273	1331	1340
		1381	1483	1913	2459			
607	Search for identifier	607	609	619	1152	1265	1419	1464
		1474	1488	1592	1847			
2550	SEEK	104	1194	2445	2550			
2492	SET reading ptrs on	258	285	293	295	415	443	477
		722	744	891	1343	2492		
2724	SHOW	++++						
1711	simple formula expression		1538	1541	1576	1632	1711	
1586	simple other expression		1529	1531	1586			
2516	SKIP text until	233	346	449	974	1326	1332	1382
		1415	1484	1798	1840	1889	1892	1905
		2516	2522	2526	2531			
28	snn necessary	28	272	311	2x406	411	667	
-	SPACE	2761	2762	2763	2764	2775		
2806	specified as name	++++						
2805	specified as value	++++						
2806	standard identifier	++++						
2607	standard symbol	++++						
1100	statement	97	1100	1135	1166	1191		
2927	store bits	++++						
2788	STORE into information list		++++					
-	STRING SYMBOL	1985	2609	2908				
2086	take from text array	1654	1657	2086	2092	2125	2661	2679
		2680						
2072	text array	++++						
1984	text symbol	++++						
1651	translate string	1651	1884					
169	type declaration	159	169					
2555	type of standard identifier		1436	1438	2555	2601		
1937	type symbol	1572	1895	1937	2020	2383		
1187	unconditional statement		1106	1132	1187			
1932	underlining symbol	1932	1968	1970	1998	2112	2153	2158
		2161	2166	2x2674	2740	2954		
1946	und symbol	1946	1971	1973	1974	1975	2x1976	1979
		2007	2012	2177	2181	2182	2184	2672
		2675	2738	2740				
2144	und symbol read	2144	2164	2167	2x2168	2181		
2806	with local	++++						
2807	without local	++++						