STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

AFDELING TOEGEPASTE WISKUNDE

Garbage-collection methods for ABC in ALGOL 60

by

R.P. van de Riet

TW report 110

January 1969

The Mathematical Centre at Amsterdam, founded the 11th of February, 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications, and is sponsored by the Netherlands Government through the Netherlands Organization for Pure Research (Z.W.O.) and the Central National Council for Applied Scientific Research in the Netherlands (T.N.O.), by the Municipality of Amsterdam and by several industries.

Acknowledgement

The author is grateful to W.P. de Roever for critically reading a preliminary manuscript; this revealed several obscurities concerning names and values.

Summary

The subject of this report is the implementation of automatic garbage-collection techniques in a formula-manipulation system written in ALGOL 60. Two garbage-collection methods, completely written in ALGOL 60, one using a relocation technique, the other using a free-list technique are compared with each other with respect to: ease of programming and memory space used.

Table of contents

1. Introduction

As a continuation of the investigations on formula manipulation in ALGOL 60, as described in [1,2], we study in this report formula-manipulation systems with automatic garbage collection and compare them with respect to ease of programming and memory space used.
On the basis of the results of this report a new formula-manipulation system will be made, in the near future, comparable with the system of [1], which should serve as a basis for a new programming language: ABC ALGOL (= ALGOL 60 + the new type formula), ABC standing for:

"Algebraïsche Bewerkingen met behulp van de Computer"

(Dutch for: "Algebraic operations by means of the computer").

Since the aim of this report is studying only the effect of garbage collection on the organization of the system, we have based ourselves on the simple formula-manipulation system of [1] chapter 1, which is copied and described in a condense form in section 2.
In section 3 we clarify the use of the terms "formula", "formula expression", a "stored formula" and "the name of a formula".
In section 4 the concepts name and value are introduced; while in sections 5,6 assignations, block entry and block exit are studied. Garbage collection with a relocation technique is treated in section 7-10 and with a free-list technique in sections 11-15. Finally, section 16 discusses the advantages and disadvantages of both techniques.

Complete and tested ALGOL 60 programs are reproduced from flexowriter tape: ALGOL 60 identifiers(or ALGOL 60 "text") occurring in the accompanying text are (or is) typed in *italics*.
Identifiers of names (section 4) will be typed in small letters; identifiers of values in capital letters.

2. Recapitulation of the simple formula-manipulation system

For the sake of convenience of the reader, the simple formula-manipulation system of [1] chapter 1 is copied:

```
begin integer one, zero, sum, product, algebraic variable, k;
  integer array F[1:1000,1:3];
  integer procedure STORE (lhs, type, rhs); value lhs, type, rhs;
    integer lhs, type, rhs;
  begin STORE:= k:= k+1; F[k,1]:= lhs;
        F[k,2]:= type; F[k,3]:= rhs
  end STORE;
  integer procedure TYPE (f, lhs, rhs); value f; integer f, lhs, rhs;
  begin lhs:= F[f,1]; TYPE:= F[f,2]; rhs:= F[f,3] end;
  integer procedure S(a,b); value a,b; integer a,b;
  S:= if a = zero then b else if b = zero then a
    else STORE (a, sum, b).
  integer procedure P(a,b); value a,b; integer a,b;
  P:= if a = zero ∨ b = zero then zero else
    if a = one then b else if b = one then a.
    else STORE (a, product, b);
  integer procedure DER(a,b); value f,x; integer f,x;
  begin integer a, type, b; type:= TYPE (f,a,b);
    DER:= if f = x then one else
      if type = sum then S(DER(a,x),DER(b,x)) else
      if type = product then S(P(a,DER(b,x)),P(DER(a,x),b))
      else zero
  end DER;
INITIALIZE: sum:= 1; product:= 2; algebraic variable:= 3; k:= 0;
  one:= STORE(0, algebraic variable, 0);
  zero:= STORE(0, algebraic variable, 0);
comment
```

Suppose one wishes to calculate:

f = (x*x+x) * dy/dx + (y*y+y) * dx/dx,

which is a trivial problem, but illustrates the need for automatic garbage collection.

The calculation is performed by the following actual program;

ACTUAL PROGRAM:

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

The result of the calculation is that f = ((y*y)+y); but during the calculation process the expression *S(P(x,x),x)* has been evaluated resulting in the storage of the useless formula ((x*x)+x) into the array *F*.

This formula is useless for two reasons:

a) it is not used for building up f;

b) it cannot be used later on since it is not known where it is stored in *F*.

Therefore, we may freely consider this formula as garbage. To get rid of it is not a simple matter since it occupies space in *F* which is surrounded by space in which still interesting formulae are stored (y and f).

In [1] section 2.14 we have studied the problem of making a procedure *COLLECT GARBAGE* which can be added to the above set of procedures and which can find out which formulae are garbage in order to create new storage space.

It has been shown that:

a) merely adding a procedure *COLLECT GARBAGE* is not possible since it is provided with insufficient information (it only knows the contents of *F*, but it should also know that the formula of which *f* is a name does not belong to the garbage)

b) a procedure *COLLECT GARBAGE* can be made if:

   1. The connection between the variables, being names of formulae, and the internal representations of these formulae, is made less direct;
   2. The form of the system procedures *S* and *P* is changed considerably.

3. Formulae, Formula expressions, internal representation of formulae and names of formulae

In this section we shall clarify the use of the term "formula".

A formula is a sequence of symbols satisfying the following syntactical rules:

<formula>::= <sum>|<product>|<derivative>|<algebraic variable>|
<formula identifier>

<sum>::= (<formula> + <formula>)

<product>::= (<formula> × <formula>)

<derivative>::= d<formula>/d<algebraic variable>

Where <algebraic variable> and <formula identifier> are just names defined in the same way as <identifier> in the ALGOL 60 report [3].

A formula as programmed in an ALGOL 60 program will have quite a different appearance; it is programmed as a Formula expression. A Formula expression is a sequence of symbols satisfying the following syntactical rules:

<Formula expression>::= <Sum>|<Product>|<DERivative>|<Algebraic variable>|
<Value of a formula variable>

<Sum>::= *S(*<Formula expression>,<Formula expression>*)*

<Product>::= *P(*<Formula expression>,<Formula expression>*)*

<Derivative>::= *DER(*<Formula expression>,<Algebraic variable>*)*

<Algebraic variable>::= *STORE(*<arithmetic expression>, *algebraic variable*,
<arithmetic expression>*)*

<Value of a formula variable>::= *V(*<formula variable>*)*

<formula variable>::= <variable>

(<variable> and <arithmatic expression> are defined in the ALGOL 60 report [3]; the rôle of *V* will be clarified in the following section). Due to its above definition a formula may appear in a mathematical textbook, a Formula expression may appear in an ALGOL 60 program.

Moreover, a Formula expression can appear only as a sequence of symbols, typed in *italics*.

By means of execution a Formula expression an image of a formula is stored in the array *F* (or in another array, see sections 11,12), occupying three places: *F*[k,*1*], *F*[k,*2*] and *F*[k,*3*], where k is some integer; this image is called the internal representation of the formula.

Some formulae have obtained names by means of an assignment statement; the left-hand side of which being a variable (or variables), the right-hand side of which being a Formula expression. These variables are called the names of the formulae. In the actual program of the preceding section we have, for instance,

one is the name of an algebraic variable;
x is the name of an algebraic variable;
f is the name of the formula f=(y×y)+y); the formula expression creating this formula being:

```
S(P(S(P(x,x),x),
    DER(y,x)),
  P(S(P(y,y),y),
    DER(x,x)
  ) )
```

and the array elements in *F*, where the internal representation of f is stored, being *F*[*8*,*1*], *F*[*8*,*2*] and *F*[*8*,*3*].
Note, that the above formula expression is not quite a Formula expression; but in the simple system of section 2 it is a legitimate formula expression. To make it a Formula expression, all *x*'s and *y*'s should be changed into *V(x)*'s and *V(y)*'s, respectively.

4. Values and names

In this section we shall study the connection between a name of a formula and the internal representation of a formula. As we have seen in the preceding section a name of a formula is an ALGOL 60 variable such as an integer *f* or an integer array element *g*[*10*].

Definition: A value is the whole of three array elements in F, for a certain number k: {F[k,*1*], F[k,*2*], F[k,*3*]}, in which the internal representation of a formula is stored.

Remark: If the particular method for storing the internal representation of a formula is chosen in another way, then the value is defined as the whole of storage cells in which the internal representation of a formula is stored.

In the simple system of section 2 the value of a variable f, being the name of a formula f, points to the value in which the internal representation of f is stored.
So, if we introduce the abbreviation: val(f) for value of f, then

val(f) → value v
with v = {F[val(f),*1*],F[val(f),*2*],F[val(f),*3*]}.

Introducing the notation val(f) for value of f, we have in the simple system of section 2 the following:

The variable *one* is the name of an algebraic variable,
val(*one*) = {F[*1*,*1*],F[*1*,*2*],F[*1*,*3*]},
while val(val(*one*)) = {*0*,*3*,*0*}.
The variable f is the name of the formula ((y×y)+y),
val(f) = {F[*8*,*1*],F[*8*,*2*],F[*8*,*3*]},
while val(val(f)) = {*7*,*1*,*4*};
the numbers *7* and *4* point to other values, namely
{F[*7*,*1*],F[*7*,*2*],F[*7*,*3*]} and {F[*4*,*1*],F[*4*,*2*],F[*4*,*3*]},
respectively.

In order to make the connection between the variable f, being a name of a formula f and the internal representation of f less direct, we introduce a name:

Definition: A name is a storage cell. The value of a name either is zero or points to a value. Each variable f, being the name of a formula f, shall point (by means of its value) to a unique name, called the name of f or name(f).

We now have the following situation:

$$\mathrm{val}(f) \rightarrow \underline{\mathrm{name}}(f).$$

If a variable g is not the name of a formula, but will possibly become the name of a formula, then for this g a name is created also, having the value zero. If a variable f is the name of a formula f, then the value of its name is defined as the value of f itself:

$$\mathrm{val}(\underline{\mathrm{name}}(f)) \rightarrow \underline{\mathrm{val}}(f) \ ;$$

hence in this case:

$$\mathrm{val}(f) \rightarrow \underline{\mathrm{name}}(f) \ ; \ \mathrm{val}(\underline{\mathrm{name}}(f)) \rightarrow \underline{\mathrm{val}}(f).$$

One might visualize the situation in the following manner, which is almost the manner described in the following sections:

Introduce the *integer array* $NAME[-1000:-1]$;

let $\mathrm{val}(f) = -5$,

then $\underline{\mathrm{name}}(f) = NAME[-5]$,

let $\mathrm{val}(NAME[-5]) = 8$, then

$\underline{\mathrm{val}}(f) = \underline{\mathrm{val}}(\underline{\mathrm{name}}(f)) = \{F[8,1],F[8,2],F[8,3]\}$.

Note, that chosing values of pointers to names to be non-positive and values of pointers to values to be positive gives the possibility of a run-time "type-check".

Let f be a formula having an internal representation stored in the value: val(f).

Definitions:

1. f is called a formula of the first kind if there exists a name n such that val(n) = val(f).
2. f is called a subformula of the formula g, if val(g) contains a pointer to val(f).
3. f is called a subformula of the formula h, if f is a subformula of the formula g and g is a subformula of h.
4. f is called a formula of the second kind if there exists no name with a value pointing at val(f), but there exists at least one formula g of the first kind of which f is a subformula.

It is clear now that giving a procedure *COLLECT GARBAGE* the list of names it can determine precisely which formulae belong to the garbage; namely, those formulae which are neither of the first kind nor of the second kind.

Assume that an integer array *NAME* were introduced in the program of section 2, then the final stage of the calculations might be visualized by means of the following diagram:

	n	*NAME*[*n*]
val(*one*)=	-1	1
val(*zero*)=	-2	2
val(*x*)=	-3	3
val(*y*)=	-4	4
val(*f*)=	-5	8

k	*F*[*k*,1]	*F*[*k*,2]	*F*[*k*,3]
1	0	3	0
2	0	3	0
3	0	3	0
4	0	3	0
5	3	2	3
6	5	1	3
7	4	2	4
8	7	1	4

fig. 1.

Abbreviation:

The value {*F*[*k*,1],*F*[*k*,2],*F*[*k*,3]} will be denoted by {*k*}.

From fig. 1, we see that the formulae with the values {1}, {2}, {3}, {4} and {8} are of the first kind, with the values {5} and {6} form the garbage and that the formula with the value {7} is of the second kind.

5. Assigning a formula to a variable by means of a Formula expression

In this section we shall investigate the ALGOL 60 analogue of an assignment statement, as e.g.

$$f:= ((x*x)+(y*y)). \qquad (1)$$

Although we have introduced the concept Formula expression already in section 3, we shall temporarily forget the concept in order to study how a formula expression should be built up. Of course, the declarations of the procedures for storing a sum and a product form the cornerstones.

There are principally two ways open:

1. We make procedures *s* and *p*, for storing a sum and a product, respectively, in such a way that they create a name, the pointer to which being delivered by the values of their procedure identifiers.
2. We make procedures *S* and *P*, for storing a sum and a product, respectively, in such a way that they create a value, the pointer to which being delivered by the values of their procedure identifiers.

Pursuing the first way, we observe that since we want to be able to write "*s(p(s(...* ", the values of the actual parameters of *s* and *p* should be pointers to names. The following statement is then legitimate:

$$f:= s(p(x,x),p(y,y)) \ , \qquad (2)$$

for, a) the values of x and y are pointers to names;

b) f becomes equal to a pointer to a name.

However, since *s* and *p* create unique names, it follows that, without precautions, execution of (2) leads to three unique names of which two are superfluous, namely the names created by $p(x,x)$ and $p(y,y)$.

Therefore, *s* and *p* should not only create names, they should also destroy the temporarily created names as the result of evaluation of their parameters. In order to save the names of x and y from this erasure it is necessary to insert a special procedure *save(f)* which delivers a pointer to a newly created name with the same value as the value of f.

Hence, (2) changes into:

$$f:= s(p(save(x),save(x)),p(save(y),save(y))). \qquad (3)$$

We shall now declare the procedure *s*, using the (undeclared) procedure *create name (VALUE)*, which delivers a pointer to a newly created name with as value: {*VALUE*}, and the (undeclared) procedure *REMOVE*, which destroys the lastly created name.

```
integer procedure s(a,b); value a,b; integer a,b;
begin integer S;
    S:= if V(a) = V(zero) then V(b) else
        if V(b) = V(zero) then V(a) else
        STORE (V(a), sum, V(b));
    REMOVE; REMOVE;
    s:= create name (S)
end s.
```

If the formula f produced by (3) becomes uninteresting, then a procedure call *ERASE(f)* is necessary to destroy the name of *f*.
It should be observed that merely assigning to *f* another value does not destroy the name of *f*; this name is only not be pointed at any more by the value of *f*. This means that the user should recognize that garbage is being formed, which means that the system is not an automatic garbage-collection system. Another disadvantage of the above approach is that a possible omission of *save* will lead to catastrophal results; moreover it is not possible for the system to check for such omissions since the values of *save(x)* and *x* are both pointers to names.

The second way, *S* and *P* deliver values pointing to values, will now be studied. Since the values of their actual parameters will now also be pointers to values (we want to write *S(P(S...)*, it is not possible to write.

f:= S(P(x,x),P(y,y));

since the values of *x*, *y* and *f* are pointers to names.
Instead, we now should have:

ASSIGN(f,S(P(V(x),V(x)),P(V(y),V(y)))), (4)

where *V(x)* becomes equal to the pointer, pointing at the value of *x*,

and where *ASSIGN* makes the value of the name of *f* equal to the pointer pointing at the value where the formula ((x*x)+(y*y)) is stored.
It has implicitly been assumed that the name of *f* does exist already, which suggests that all the names of the variables, being used as names for formulae, are created upon block entry through appropriate "formula declarations".
Let the procedure *SAVE(VALUE)* create a new name with as value: {*VALUE*}.

The procedure *S* may now be declared as follows:

```
integer procedure S(A,B); integer A,B;
begin integer A1,B1; A1:= A; SAVE(A1); B1:= B;
  comment The formula with a value pointed at by A1 is saved from erasure
    by a possible garbage collection during evaluation of the actual parameter
    B. After B has been evaluated there is no danger anymore from any state-
    ment in this procedure body, so that the temporarily created name for
    A1 can be removed by:
    REMOVE(A1);
  comment REMOVE(A1) has as effect that:
    1. The value of A1 becomes equal to the value of the lastly created name,
    which, after SAVE(A1), equalled the value of A, but which may have been
    changed after a possible garbage collection during "B1:=B",,
    2. the lastly created name is destroyed.;
    S:= if A1= V(zero) then B1 else
        if B1= V(zero) then A1 else
        STORE(A1, sum, B1)
    end S.
```

We remark that, for storing x+y+z+u, it is more easy to write *S(V(x),S(V(y),S(V(z),V(u))))* then *S(S(S(V(x),V(y)),V(z)),V(u))*; therefore, a change of the first four statements of the procedure declaration into:

```
B1:= B; SAVE(B1); A1:= A; REMOVE(B1)
```

will, in general, lead to the creation of less simultaneously existing temporary names.

An important observation is the following.
If the storage space for the names is chosen in the same array as for the values, then the space needed to save *B1*, in the above procedure *S* may be used afterwards for storing the triple: {*A1,sum,B1*} without a garbage collection; furthermore, the saving of *B1* will, in general, not cost extra storage space (provided *A1* and *B1* are not equal to *V(zero)*).

Erasure of the formula f, as introduced by (4) is now simply possible by assigning to its name another value; i.e. by assigning to the name of *f* another value. A possible garbage collection will then find out that f belongs to the garbage since there is no name pointing to its internal representation.
This system may thus be called an automatic garbage-collection system.
Moreover, syntactic control is possible on the appearance of *V*, since the value of *V* is a pointer to a value and the value of *x* is a pointer to a name (which may be chosen negative, e.g.).
It does not need saying that the latter, just described, method will be followed in the next sections.

6. Block entry and block exit

Upon block entry all the variables, to be used as names of formulae, should be declared as variables of integral type, and unique names should be created for them. To combine this creation and a possible initialization, we use a procedure *DE*, which heading runs as follows:

Boolean procedure DE (first time, f, F); value first time;
Boolean first time; integer f, F;

Creation of names and the initialization (to *F*) is then possible by means of a "declaration statement", syntatically defined as follows:

<declaration statement>::= *DE(true,*<variable>,<value>*)* |
DE(<declaration statement>,<variable>,<value>*)*
<value>::= *0* | <Formula expression>

Example: the variables x and y, to be used as names of algebraic variables, and the variable f to be used as name of the formula ((x*x)+(y*y)) are "declared" as follows:

```
integer x,y,f;
  DE(DE(DE(true,x,STORE(0,algebraic variable,0)),
        y, STORE(0,algebraic variable,0)),
     f, S(P(V(x),V(x)),P(V(y),V(y))));
```

It is necessary that, immediately after block entry, *DE* is called with *first time* = *true*; afterwards it has to be called with *first time* = *false*.

Example: The subscripted variables $g[i]$, $i = 1,\ldots,10$ will be used as names for, until now, unknown formulae; they may be "declared" by:

```
integer i; integer array g[1:10];
  for i:= 1 step 1 until 10 do DE(i=1, g[i], 0); .
```

Upon block exit all the lastly created names after the corresponding block entry, are erased by a call of *ERASE*. These lastly created names are created during and after a call of *DE* with *first time* = *true*; hence, each executed call: *ERASE* should correspond with one and only one executed call: "*DE(true,...)*".

7. Garbage collection with a relocation technique

In this section we describe a garbage-collection process which is based on a relocation technique; i.e. after a garbage collection the "non-garbage" formulae have been relocated in the array *F* such that there are no holes left in *F*.
In the following section the ALGOL 60 program will be reproduced.

The garbage-collection process is split into two subprocesses: GCPA and GCPB. GCPA stores the contents of the values of "non-garbage" formulae in a second array *Fdrum*, which serves as an image of the array *F*; it is filled as if it were the future array *F*.

GCPB stores the contents of *Fdrum* into *F*.
We remark that in an actual, more realistic, system we would not use an array *Fdrum* but secondary storage instead, such as e.g. a drum (See [1] chapter 2 (section 2.7)).

GCPA is described by means of the following algorithm:

Step 1: If there is a name, then choose the first name and take step 2;
otherwise, take step 8.

Step 2: Take as storage cell s the last chosen name. If the value of s is zero, then take step 7; otherwise, take step 3.

Step 3: Choose as value v the value pointed at by the value of s. Take step 4.

Step 4: If v is marked as being treated already, then the value of s becomes the pointer to the value in *Fdrum* where the contents of v has been stored and take step 7;
otherwise, take step 5.

Step 5: If v contains pointers to the values, v_i, i=1,..,n, n≥1, then for i=1,...,n do the following:
choose as storage cell s = v_i and
execute the GCPA beginning with step 3 and ending with step 6.
Take step 6.

Step 6: Store the contents of v (possibly being changed in step 5) into a value V of *Fdrum*. Mark v as being treated already and store the pointer to V into v and into s.
Take step 7.

Step 7: Choose the next name, if available, and take step 2.
If there is no name left, then take step 8.

Step 8: GCPA is finished.

The algorithm for GCPB is very simple: The contents of *Fdrum* is exactly copied into *F* and in the same locations; i.e.:

for i:= 1 step 1 until pointer of F do for j:= 1,2,3 do
F[i,j]:= Fdrum[i,j].

Array elements of *F*, occupying the "top position" in *F*, will be chosen for the names. The organisation of which is as a stack with the pointer: *pointer of name*, growing downstairs and shrinking upwards.

A garbage collection will take place if

pointer of F = pointer of name -1,

where *pointer of F* is the pointer of the values in *F*; i.e. the current value of *pointer of F* is pointing to the lastly created value in *F*; in the same way, the current value of *pointer of name* is pointing to the lastly created name in *F*.

The procedure *COLLECT GARBAGE* is provided also with a parameter *arr*, specified as *integer array*, in which it is possible to give to *COLLECT GARBAGE* some extra special names, not occuring in the name list (see the procedure declaration of *STORE*).

In order to organize the block entry (creation of names) block exit (erasure of names) mechanism, a second stack, the array *last name*, with pointer *pointer of last name*, is used. In this stack the current value of the pointer of the name list: *pointer of name*, is stored upon block entry by means of a call *DE(true,...)*. Upon block exit the pointer of the name list is reset to the value of the top of *last name* by means of a call of *ERASE*.

8. The relocation garbage-collection technique programmed in ALGOL 60 for a simple system.

In this section the ALGOL 60 program is reproduced describing the relocation garbage-collection technique. Some procedures have not been mentioned before: the procedure *AV* for storing an algebraic variable; the procedure *ERROR* which detects a possible error, prints the error message and discontinues the calculation by means of a call of the standard procedure *EXIT*;
the procedures *PR nlcr*, *PR string*, *PR*, *PR intnum* and *PR sym*, for printing and punching a new-line-carriage-return symbol, a string, a real number,

an integral number and a symbol;
the procedure *OUTPUT* which outputs a formula without superfluous brackets.
Finally, the program ends with an actual program which will be discussed, together with its also reproduced results, in the section 9.

```
begin comment A simple system of ABC - ALGOL 60 procedures for
  formula manipulation with garbage collection (relocation-technique).
  RPR 061168/02 - T8190. R.P. van de Riet;

  integer pointer of F,pointer of name,pointer of last name,
    max of F,max of last name,
    algebraic variable,sum,product,one,zero;

  max of F:= read; max of last name:= read;
begin integer array F[1:max of F,1:3],last name[1:max of last name],
    Fdrum[1:max of F,1:3],auxiliary array[1:5];

  procedure INITIALIZE;
  begin pointer of F:= pointer of last name:= 0; pointer of name:=
    max of F + 1; algebraic variable:= 1; sum:= 2; product:= 3;
    DE(DE(true,one,AV(0,1)),zero,AV(0,0));
  end INITIALIZE;

  procedure COLLECT GARBAGE(n,arr); value n; integer n;
    integer array arr;
  begin integer i,j;
    integer procedure SET ON DRUM(FF); value FF; integer FF;
    if FF = 0 then SET ON DRUM:= 0 else
    if F[FF,1] < 0 then SET ON DRUM:= -F[FF,1] else
    begin integer t,A,B; t:= TYPE(FF,A,B);
      if DYADIC OP(t) then
      begin A:= SET ON DRUM(A); B:= SET ON DRUM(B) end;
```

```
      SET ON DRUM:= pointer of F:= pointer of F + 1;
      Fdrum[pointer of F,1]:= A; Fdrum[pointer of F,2]:= t;
      Fdrum[pointer of F,3]:= B; F[FF,1]:= -pointer of F
    end SET ON DRUM;
    procedure DUMP;
    begin PR nlcr; for i:= 1 step 1 until max of F do
      begin PR nlcr; PR int num(i); for j:= 1,2,3 do
        begin PR string(« »);
          if ¬(j > 1 ∧ i ≥ pointer of name) then PR int num(F[i,j])
    end end end DUMP;
    if pointer of F = pointer of name - 1 then
    begin DUMP;
comment GCPA:; pointer of F:= 0; for i:= max of F step -1 until
        pointer of name do F[i,1]:= SET ON DRUM(F[i,1]);
      for i:= 1 step 1 until n do arr[i]:= SET ON DRUM(arr[i]);
comment GCPB:; for i:= 1 step 1 until pointer of F do
      for j:= 1,2,3 do F[i,j]:= Fdrum[i,j]; DUMP;
      ERROR(pointer of F = pointer of name - 1,«no space left»)
end end COLLECT GARBAGE;

integer procedure STORE(A,t,B); value A,t,B; integer A,t,B;
begin auxiliary array[1]:= pointer of F:= pointer of F + 1;
F[pointer of F,1]:= A; F[pointer of F,2]:= t; F[pointer of F,3]:= B;
  COLLECT GARBAGE(1,auxiliary array);
  STORE:= auxiliary array[1]
end STORE;

integer procedure AV(l,r); value l,r; integer l,r;
AV:= STORE(l,algebraic variable,r);
```

```
integer procedure SAVE(FF); value FF; integer FF;
begin ERROR(FF < 0,‹F negative in SAVE›);
  SAVE:= pointer of name:= pointer of name - 1;
  F[pointer of name,1]:= FF;
  COLLECT GARBAGE(0,auxiliary array)
end SAVE;

integer procedure V(f); value f; integer f;
begin ERROR(f ≥ 0,‹name not appropriate in V›);
  V:= F[-f,1]
end V;

integer procedure TYPE(FF,A,B); value FF; integer FF,A,B;
begin ERROR(FF ≤ 0,‹F negative in TYPE›);
  A:= F[FF,1]; TYPE:= F[FF,2]; B:= F[FF,3]
end TYPE;

Boolean procedure DYADIC OP(t); value t; integer t;
DYADIC OP:= t = sum ∨ t = product;

procedure REMOVE(FF); integer FF;
begin FF:= F[pointer of name,1]; pointer of name:= pointer of name + 1;
  ERROR(pointer of name > max of F - 1,‹REMOVE no appropriate›)
end REMOVE;

Boolean procedure DE(first time,f,F); value first time;
  Boolean first time; integer f,F;
begin if first time then
  begin pointer of last name:= pointer of last name + 1;
    ERROR(pointer of last name > max of last name,
    ‹pointer of last name too large›);
    last name[pointer of last name]:= pointer of name
  end; f:= - SAVE(F); DE:= false
end DE;
```

```
procedure ERASE;
begin pointer of name:= last name[pointer of last name];
   pointer of last name:= pointer of last name - 1;
   ERROR(pointer of last name < 1,‡ERASE not appropriate‡)
end ERASE;

integer procedure ASSIGN(f,FF); value f,FF; integer f,FF;
begin ERROR(f ≥ 0,‡name not appropriate in ASSIGN‡);
   ASSIGN:= F[-f,1]:= FF
end ASSIGN;

integer procedure ERROR(b,s); value b; Boolean b; string s;
if b then
begin PR nlcr; PR string(s); EXIT; ERROR:= 1 end ERROR;

procedure PR nlcr; PR string(‡
‡);
procedure PR string(s); string s;
begin PRINTTEXT(s); PUTEXT(s) end PR string;
procedure PR(r); value r; real r;
begin PUNCH(r); PRINT(r) end;
procedure PR int num(a); value a; integer a;
begin integer b; if a < 0 then begin PR string(‡-‡); a:= - a end;
   if a ≤ 9 then PR sym(a) else
   begin b:= a : 10; a:= a - b × 10; PR int num(b); PR sym(a) end
end PR int num;
procedure PR sym(s); value s; integer s;
begin PUSYM(s); PRSYM(s) end;
```

```
integer procedure S(A,B); integer A,B;
begin integer A1,B1; B1:= B; SAVE(B1); A1:= A;
  ERROR(A1 < 0 ∨ B1 < 0,‡A or B negative in S†); REMOVE(B1);
  S:= if A1 = V(zero) then B1 else if B1 = V(zero) then A1 else
      STORE(A1,sum,B1)
end S;

integer procedure P(A,B); integer A,B;
begin integer A1,B1; B1:= B; SAVE(B1); A1:= A;
  ERROR(A1 < 0 ∨ B1 < 0,‡A or B negative in P†); REMOVE(B1);
  P:= if A1 = V(zero) ∨ B1 = V(zero) then V(zero) else
      if A1 = V(one) then B1 else if B1 = V(one) then A1 else
      STORE(A1,product,B1)
end P;

procedure OUTPUT(name,OUTPUT VARIABLE); value name; integer name;
  procedure OUTPUT VARIABLE;
begin procedure OP(F,type); value F,type; integer F,type;
  begin integer t,A,B;
    procedure LBR; if t < type then PR string(‡(†);
    procedure RBR; if t < type then PR string(‡)†);
    t:= TYPE(F,A,B);
    if t = algebraic variable then OUTPUT VARIABLE(F) else
    if DYADIC OP(t) then
    begin LBR; OP(A,t);
      if t = sum then PR string(‡+†) else
      if t = product then PR string(‡×†);
      OP(B,t); RBR
    end else ERROR(true,‡f not appropriate in OUTPUT†)
  end OP;
  OP(V(name),0)
end OUTPUT;
```

ACTUAL PROGRAM:

```
begin integer x,y,f;
  procedure OV(f);
  begin integer A,t,B; t:= TYPE(f,A,B);
    if B ≤ 1 then PR int num(B) else
    if B = 2 then PR string(⟨x⟩) else
    if B = 3 then PR string(⟨y⟩) else
    ERROR(true,⟨error in output⟩)
  end;

  procedure PRINT(x,s); integer x; string s;
  begin PR nlcr; PR string(s); PR string(⟨ (name: ⟩); PR int num(x);
    PR string(⟨) (value: ⟩); PR int num(V(x)); PR string(⟨) formula: ⟩);
    OUTPUT(x,OV)
  end;

  PR nlcr; PR string(⟨results RPR 061168/02⟩);
  max of F:= 13; INITIALIZE;
  comment garbage:; AV(10,10);
  DE(DE(DE(true,x,AV(0,2)),y,AV(0,3)),f,0);
  PRINT(x,⟨x =⟩); PRINT(y,⟨y =⟩);
  ASSIGN(f,S(P(S(V(x),V(y)),V(x)),V(y)));
  PRINT(x,⟨x =⟩); PRINT(y,⟨y =⟩); PRINT(f,⟨f =⟩);
  ERASE;
  DE(DE(true,x,AV(0,2)),f,0);
  ASSIGN(f,P(V(x),S(V(x),S(V(x),V(one)))));
  PRINT(x,⟨x =⟩); PRINT(f,⟨f =⟩);
  ERASE;
```

```
    max of F:= 18; INITIALIZE;
    comment garbage:; AV(20,20); AV(30,30);
    DE(DE(DE(true,x,AV(0,2)),y,AV(0,3)),f,S(P(V(x),V(y)),V(one)));
    PRINT(x,‡x =‡); PRINT(y,‡y =‡); PRINT(f,‡f =‡);
    ASSIGN(f,S(P(V(f),P(V(x),V(y))),
                S(P(V(f),V(f)),
                  S(P(S(V(f),P(V(f),V(y))),V(zero)),
                    P(V(x),V(f))
            ) ) ) );
    PRINT(x,‡x =‡); PRINT(y,‡y =‡); PRINT(f,‡f =‡);
    ERASE
  end
end end   100      100

  results RPR 061168/02
  x = (name: -11) (value: 4) formula: x
  y = (name: -10) (value: 5) formula: y

  1       0       1       1
  2       0       1       0
  3       10      1       10
  4       0       1       2
  5       0       1       3
  6       5
  7       4
  8       5
  9       0
  10      5
  11      4
  12      2
  13      1
```

```
1       0       1       1
2       0       1       0
3       0       1       2
4       0       1       3
5       -4      1       3
6       4
7       3
8       4
9       0
10      4
11      3
12      2
13      1
x = (name: -11) (value: 3) formula: x
y = (name: -10) (value: 4) formula: y
f = (name: -9) (value: 7) formula: (x+y)×x+y

1       0       1       1
2       0       1       0
3       0       1       2
4       0       1       3
5       3       2       4
6       5       3       3
7       6       2       4
8       0       1       2
9       1
10      0
11      8
12      2
13      1
```

```
1       0       1       1
2       0       1       0
3       0       1       2
4       0       1       3
5       3       2       4
6       5       3       3
7       6       2       4
8       -3      1       2
9       1
10      0
11      3
12      2
13      1
x = (name: -11) (value: 3) formula: x
f = (name: -10) (value: 6) formula: x×(x+x+1)
x = (name: -16) (value: 5) formula: x
y = (name: -15) (value: 6) formula: y
f = (name: -14) (value: 8) formula: x×y+1

1       0       1       1
2       0       1       0
3       20      1       20
4       30      1       30
5       0       1       2
6       0       1       3
7       5       3       6
8       7       2       1
9       5       3       8
10      8       3       6
11      10
12      2
13      9
```

14	8		
15	6		
16	5		
17	2		
18	1		

1	0	1	1
2	0	1	0
3	0	1	2
4	0	1	3
5	3	3	4
6	5	2	1
7	3	3	6
8	6	3	4
9	-7	3	8
10	-8	3	6
11	8		
12	2		
13	7		
14	6		
15	4		
16	3		
17	2		
18	1		

1	0	1	1
2	0	1	0
3	0	1	2
4	0	1	3
5	3	3	4
6	5	2	1

7	3	3	6
8	6	3	4
9	6	2	8
10	6	3	6
11	10	2	7
12	4		
13	11		
14	6		
15	4		
16	3		
17	2		
18	1		

1	0	1	1
2	0	1	0
3	0	1	2
4	0	1	3
5	3	3	4
6	5	2	1
7	6	3	6
8	3	3	6
9	7	2	8
10	-7	3	6
11	-9	2	7
12	4		
13	9		
14	6		
15	4		
16	3		
17	2		
18	1		

x = (name: -16) (value: 3) formula: x

y = (name: -15) (value: 4) formula: y

f = (name: -14) (value: 12) formula: (x×y+1)×x×y+(x×y+1)×(x×y+1)+x×(x×y+1)

9. Discussion of the actual program and the results

In order to test the garbage-collection mechanism, some examples, in which garbage collections take place, have been treated in the actual program.
The garbage is introduced in two ways:
explicitly, by the calls *AV(10,10)*, *AV(20,20)* and *AV(30,30)*, introducing algebraic variables without a name;
implicitly, by means of *ERASE* and by means of the Formula expression:

P(S(V(f),P(V(f),V(y))),V(zero)) ,

producing the garbage: (f+(f×y)).

The output of a formula consists of the following:

1. the value of the variable being its name;
2. the value of the name of the variable being its name;
3. its ordinary representation as a formula.

Immediately preceding and immediatly after a garbage collection, the interesting part of the contents of the array *F* is output (see procedure *DUMP* in the procedure *COLLECT GARBAGE*); i.e. if *k* is the pointer to a value, the values of the array elements *F[k,1]*, *F[k,2]* and *F[k,3]* are printed; if *k* is the pointer to a name, then the value of *F[k,1]* is printed only. This output is preceded by a column in which the value of *k* is printed.

Remark: All garbage collections occurred after a call of *SAVE*, as should be expected.

10. The differentiation process

Calculating the derivative of a formula f with respect to an algebraic variable x means going recursively along the branches of the tree representing f and producing meanwhile partial results of the derivative.
However, producing partial results means storing formulae, which means possible garbage collections, which means changing the pointers to the values, which finally means that the branches of the tree, being pointers, are "slithery".

Therefore, in order to have fixed grip on the branches, names are introduced in the following declaration, for holding the branches.

```
integer procedure DER(F,x); value F,x; integer F,x;
begin integer t, a, A, b, B; t:= TYPE(F,A,B);
      if DYADIC OP(t) then DE(DE(true,a,A),b,B);
      DER:= if F = V(x) then V(one) else
        if t = sum then S(DER(V(a),x),DER(V(b),x)) else
        if t = product then S(P(DER(V(a),x),V(b)),P(V(a),DER(V(b),x)))
        else V(zero);
      if DYADIC OP(t) then ERASE
end DER.
```

Let the *depth* of a formula f being defined as follows:

```
integer procedure depth(f); value f; integer f;
begin integer procedure d(F); value F; integer F;
      begin integer t, A, B, dA, dB; t:= TYPE(F,A,B);
        if DYADIC OP(t) then
        begin dA:= d(A); dB:= d(B);
           d:= (if dA<dB then dB else dA) +1
        end else d:= 0
      end d; depth:= d(V(f))
end depth.
```

It can easily be seen that, if *depth*(f) = n, then, during the computation of *DER*(f,x), there will be a moment at which 2n new names have been created simultaneously and n places in the stack *last name* have been used. These 2n names are created in the procedure body of *DER*; besides these names, names may be created by calls of *S* and *P*. All names occupy temporarily the space which afterwards may be used for the storage of the derivative itself. A situation is, however, far from impossible that there is not enough space for storing these temporary names, while there would be space for storing the derivative.

Consider, for example, $f = \sum_{i=1}^{n+1} x = (\ldots((x+x)+x)\ldots+x)$, with *depth* (f) = n;
in calculating *DER*(f,x), *DER* introduces for each bracket pair two names and one place in *last name*; moreover *S* introduces one name for each bracket pair pointing to the value of *DER*(x,x) = *one*. Hence, at the moment the innermost sum (x+x) is being treated by *DER* and before the statement *ERASE* is called, there have been created 3 n new names; and n places in *last name* have been occupied.

However, the storage of df/dx needs only n places in the array *F*.

We conclude that the method should be refined in two ways:

1. the saving of values should not be performed by means of declaration statements; for, the declaration statements lead to the n places in *last name*.
2. the saving of values should be performed by a *SAVE-REMOVE* mechanism introducing the least as possible number of new names.

Let us face the problem once again:

Suppose f= (a*b), the values of *F*, *A* and *B* being the pointers to the values of f, a and b, respectively.

During the calculation of ((da/dx*b) + (a*db/dx)) a garbage collection is possible. We do not have to worry about f, a, and b being erased, since f will be a formula of the first or second kind. The problem we are involved with is that the pointers to the values of f, a and b may change, without a corresponding change of the values of *F*, *A* and *B*. We now observe that it is not necessary to know the values of *A* and *B*, if we know the value of *F*; for, the values of *A* and *B* can be calculated by means of *TYPE* from the value of *F*.

Let us now try the following procedure declaration:

```
integer procedure DER(F,x); value F,x; integer F,x;
begin integer procedure A of F;
      begin integer A,B; REMOVE (F); SAVE (F);
            TYPE(F,A,B); A of F:= A
      end;
      integer procedure B of F;
      begin integer A,B; REMOVE (F); SAVE (F);
            TYPE(F,A,B); B of F:= B
      end;
```

```
    integer t, A, B; t:= TYPE(F,A,B);
    if DYADIC OP(t) then SAVE(F);
    DER:= if F = V(x) then V(one) else
      if t = sum then S(DER(A of F,x), DER(B of F,x)) else
      if t = product then S(P(DER(A of F,x),B of F),
                            P(A of F, DER(B of F,x)))
      else V(zero);
    if DYADIC OP(t) then REMOVE(F)
end DER.
```

We have used in this declaration that *SAVE(F)* adds a new name with as value the value of *F*, on the top of the name stack and that *REMOVE(F)* makes the value of *F* equal to the value of the top name and removes this name. On first sight the declaration seems what we are looking for: no declaration statement and only one newly created name. However, we have to remember that *S* and *P* also create names at the top of the name stack, so that the name for *F* will not always be the top name. Hence the above declaration for the differentiation process may lead to errors. Fortunately, we can calculate, however, how far the name for *F* is sunk into the name stack by counting the number of *SAVE*'s executed by *S* and *P*.
In the following, and final, procedure declaration for the derivative process, the deepness of the name for *F* is being taken into account by means of the procedure *GET*, which digs up the name for *F*, combines the actions of the procedures *A of F* and *B of F*, declared above, and, finally, "buries" the name for *F* as deep as it originally lay.

```
integer procedure DER(F,x); value F,x; integer F,x;
begin integer t,A,B;
  integer procedure GET(i,lhs); value i,lhs; integer i; Boolean lhs;
  begin integer j; integer array REM[1:i];
    for j:= 1 step 1 until i do REMOVE(REM[j]);
    TYPE(REM[i],A,B); GET:= if lhs then A else B;
    for j:= i step -1 until 1 do SAVE(REM[j])
  end GET;
```

```
t:= TYPE(F,A,B);
if DYADIC OP(t) then SAVE(F);
DER:= if F = V(x) then V(one) else
      if t = sum then S(DER(GET(2,true),x),DER(GET(1,false),x)) else
      if t = product then
        S(P(DER(GET(3,true),x),GET(2,false)),
          P(GET(2,true),DER(GET(1,false),x))) else V(zero);
if DYADIC OP(t) then REMOVE(F)
end DER;
```

Note, that it were possible to construct the procedure *GET* in such a way that it does not use *SAVE* and *REMOVE*, but that it operates directly on the name list by means of *pointer of name* and *F*.

Having studied the derivative process and the value - name mechanism in such details we leave to the reader the Problem:

What may go wrong in the declaration statement:

DE(DE(true, a, A) b, B),

in the very first declaration of *DER*?

Concluding this section we observe that it turned out to be far from trivial to construct a "water-proof" differentiation process. This is due to the fact that the branches of the tree representing a formula are "slithery" in a relocation garbage-collection technique.
It is for this reason that we study in the next sections a garbage-collection technique without relocating the non-garbage formulae.

11. Garbage collection with a free-list technique

In a garbage-collection system relocating formulae, the free storage cells are characterized by lying in a certain area of the storage space; i.e. the pointer k of a free cell satisfies:

pointer of F < k < *pointer of name*.

This characterization of free cells is no longer possible if "non-garbage" formulae are not relocated such that there remain holes in the storage space.

There are two other ways to characterize a free cell:
1. the free cell is flagged;
2. the free cell is pointed at by the "outer world".

The implication of flagging a free cell is that the system has to go through the storage space in order to find a new free cell; this may be very time-consuming, in particular, if there are only a few free cells left.
Another disadvantage is the need for an extra bit: the flag.

The second way will be studied now.
We use a free-list technique as follows:
Suppose, that the first free cell has an address contained in the variable *free cell*.
Suppose, furthermore, that the (n+1)-st free cell as an address contained in the n-th free cell (assuming that there is at least one free cell).
Suppose, finally, that the address of the last free cell is contained in the variable *last free cell*.
The finding of a free cell is now obvious: it is pointed at by the value of *free cell* and the value of *free cell* is changed in order to point to the second free cell (if available). Adding a new free cell to the free-cell list is also a trivial matter: it is connected with the last free cell pointed at by the value of *last free cell* and the value of *last free cell* is changed in order to point to the new free cell.

Evidently, a garbage collection is necessary if

free cell = *last free cell*

and the last free cell has been used.

In section 4 we have introduced names, which, in the relocation garbage-collection technique, have two purposes:

1. as information for the system to determine which formulae are "non-garbage";
2. as storage cells in which the pointers to values can be stored (these storage cells have to be known by the system during a garbage collection).

Since we now have a situation tha the value of a "non-garbage" formula remains unchanged, it is not evident that we have to introduce names; for, a variable, being the name of a formula, may now contain itself the pointer to the value of that formula. So, let us temporarily discard the notion of names and concentrate on the problem of how "non-garbage" formulae can be identified without using names.

Assume that the only formulae present in the storage space are "non-garbage" formulae.
Let a certain formula f be condemned to be garbage dependent on whether:

1. it is not a subformula of a "non-garbage" formula and
2. it is not pointed at by a variable being a name of f.

The information: being not a subformula of a "non-garbage" formula can be found (in a non-trivial way) in the storage space itself.
The information that there is not a variable pointing at the value of f must be stored in the value of f itself; for, we do not use names. This information can not consist of a single flag since f may have more than one names; therefore, this information should be a number defining how many names f has. Let us call this number the reference counter of f. This number may then also be used to count how many times f is a subformula of another formula (how many times it is referenced). The treatment of f, condemned to be garbage, can now be easily performed by the following "erase" process:

Step 1: Choose as value the value of f. Take step 2.
Step 2: If the value of the last chosen value v points to the values v_i, i = 1,...,n, then
for i = 1,...,n do the following:
choose as value: v_i ; perform the "erase" process beginning with step 2 and ending with step 5.
Take step 3.

Step 3: Decrease the reference counter of v by one.
If the reference counter is zero then take step 4; otherwise, take step 5.

Step 4: Erase v by connecting it with the free-cell list.
Take step 5.

Step 5: The erase process is finished.

There are two serious drawbacks attached to this approach:

1. The reference counters will occupy much space; each value, whether it will be referenced often or seldom, will occupy a counter of a length dictated by the maximum number of possible references.
   This maximum number will, in general, be reached for the values of algebraic variables, being referenced often.
2. An erase call should explicitly be stated by the user; he should then constantly be aware of the appearance of garbage.
   An easy block-entry-block-exit device is not possible.

Having observed the consequences of dropping the names, we now proceed by introducing the names again.

From now on, we shall not use the array *F* of the preceding sections, but we shall use instead the arrays *C* and *C type* declared as:

integer array C[1:max of C,1:2], C type[1:max of C] .

The array elements of *C* will be used for names and "non-type"-parts of values. The array elements of *C type* will be used for storing the "type"-parts of values (i.e. *F[k,2]* in section 3). In a later, more realistic, system, we shall not use *C type* anymore, but we shall store the whole value in a compact way in the array elements of *C*.

The free cells are connected with each other as follows:
the value of *free cell* points to first free cell:
{*C[free cell,1], C[free cell,2]*} ;
consider a certain free cell: {*C[k,1],C[k,2]*} ,
then the next free cell is: {*C[C[k,1],1],C[C[k,1],2]*} ;
the last free cell is: {*C[last free cell,1],C[last free cell,2]*} .

The names are connected with each other as follows:
the last name is {*C[last name,1],C[last name,2]*} ;
consider a certain name: {*C[k,1],C[k,2]*} ,
then the preceding name is: {*C[C[k,2],1],C[C[k,2],2]*}, provided *C[k,2]≠0* ;
let the first name be: {*C[f,1],C[f,2]*}, then
the value of *C[f,2]* is zero.
Let a name be {*C[k,1],C[k,2]*}, then
the value of *C[k,1]* determines the possible value of the name.
If *C[k,1] = 0*, then there is no value, otherwise the value of
the name is {*C[C[k,1],1],C type[C[k,1]],C[C[k,1],2]*} .

Creation of a new name is performed by adding the first free cell (if available) to the end of the name list.
Erasure of the lastly created name (*REMOVE*), and thus condemning the value of this name possibly as being garbage, is performed by adding the last name to the free-cell list. It is thus necessary to keep track of the last name; it is not necessary to know the first name.
Erasure of an arbitrary name n is performed by:

1. the names preceding and following n are connected;
2. n is connected with the free-cell list.

As in section 7 there are two ways to create and erase names:

1. By means of a call: *SAVE(F)*; a new name is created having a value pointed at by the value of *F*.
   This creation is cancelled by a call of *REMOVE*, destroying the lastly created and still living name.
2. By means of a declaration statement:
   $DE(\ldots DE(DE(true,f_1,F_1),f_2,F_2)\ldots,f_n,F_n)$,
   which creates n new names whose pointers are assigned to f_i and whose values are, if $F_i \neq 0$, pointers to the values pointed at by the values of F_i $(i = 1,\ldots,n)$. The effect of this statement is cancelled by a call of *ERASE*.

The block-entry-block-exit mechanism uses a stack in which the current value of *last name* is stored upon block entry (*DE(true,...)*). This stack is also organized as a list with pointer: *pointer of stack*.
Let {*C[k,1],C[k,2]*} be a cell of this stack, then *C[k,1]* contains a value of *last name* and *C[k,2]* determines (if ≠0) a preceding cell of the stack.
The cell for which *k* = *pointer of stack* is the last cell of the stack. The cell for which *C[k,2]* = 0 is the first cell of the stack.
By means of a picture we shall now illustrate the organization of the arrays *C* and *C type*.
Since the cells of *C* may be thought of as beads on a string, which may be shuffled without breaking the string, we shall draw the cells of the list of free cells, the cells of the list of names, the cells of the list of values and the cells of the stack compactly.
By means of the arrows the pointers are made visible.

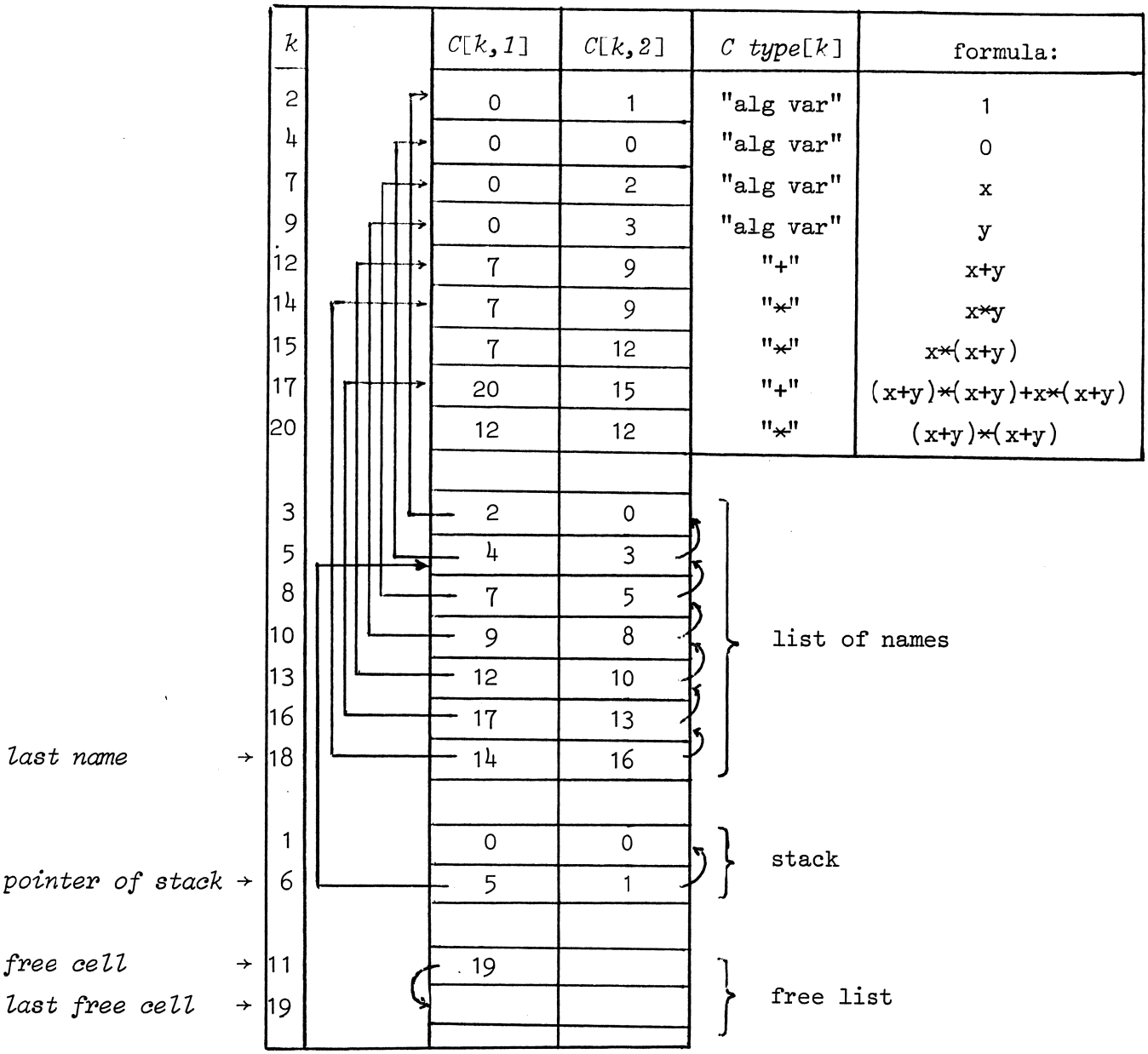

fig. 2. The storage organization.

The above example has been taken from the program output described in the next section; "alg var", "+", and "*" are symbolic representations for 1, 2, and 3, respectively.

The garbage-collection process may now be described by means of the following algorithm:

Step 1: If there is a name, then choose the last name and take step 2; otherwise, take step 7.

Step 2: Choose as value v the value pointed at by the last chosen name. Take step 3.

Step 3: If v is marked, as being treated already, then take step 6; otherwise, take step 4.

Step 4: If v contains pointers to the values, v_i, i = 1,...,n, n≥1, then for i = 1,...,n do the following:

Choose as value v the i-th value v_i and execute the garbage-collection process beginning with step 3 and ending with step 6, without executing it.

Take step 5.

Step 5: Mark v as being treated already.

Take step 6.

Step 6: If there is a preceding name, then choose this name and take step 2; otherwise, take step 7.

Step 7: Mark the storage cells constituting the list of names and the stack. Connect all the unmarked storage cells with the list of free cells. Remove all the marks introduced above.

Take step 8.

Step 8: The process is finished.

12. The free-list garbage-collection technique programmed in ALGOL 60 for a simple system

In this section the ALGOL 60 program is reproduced. The following procedures have not been mentioned already:

LHS, which becomes equal to the value of the left-hand side part of a storage cell {*C*[*k*,*1*],*C*[*k*,*2*]},

RHS, which becomes equal to the value of the right-hand side part of a storage cell,

STIL, stores a number in the left-hand side part of a storage cell,

STIR, stores a number in the right-hand side part of a storage cell,

ST, stores two numbers in a storage cell,

ST TYPE, stores the "type"-part of a value into *C type*,

(Remark, the above procedures have been introduced to make a future reorganization of the storage cells more easy.)

join to free space, connects a cell with the free-cell list.

Since the values of formulae are fixed now, we can take advantage of it by assigning the pointers of the values, not to names only, but also to variables, which are distinguished from variables, being names of formulae, by their identifiers in which capital letters are used.
In this way *ONE* and *ZERO* have values pointing to the values of the variables *one* and *zero*.

For a discussion of the actual program and its results we refer to the next section.

```
begin comment A simple system of ABC - ALGOL.
  Garbage collection with a free list.
  RPR 181168/02 - T 8190, R. P. van de Riet;

  integer free cell,last free cell,last name,pointer of stack,max of C,
  algebraic variable,sum,product,one,zero,ONE,ZERO;

  max of C:= read;
  begin integer array C[1:max of C,1:2],auxiliary array[1:5],Ctype[1:max of C];
    Boolean array traced[1:max of C];

    procedure INITIALIZE;
    begin integer i; for i:= 1 step 1 until max of C do C[i,1]:= i + 1;
      free cell:= 1; last free cell:= max of C;
      last name:= pointer of stack:= 0;
      algebraic variable:= 1; sum:= 2; product:= 3;
      DE(DE(true,one,AV(0,1)),zero,AV(0,0));
      ONE:= V(one); ZERO:= V(zero)
    end INITIALIZE;

    integer procedure LHS(k); value k; integer k;
    LHS:= C[k,1];

    integer procedure RHS(k); value k; integer k;
    RHS:= C[k,2];

    procedure STIL(k,v); value k,v; integer k,v;
    C[k,1]:= v;

    procedure STIR(k,v); value k,v; integer k,v;
    C[k,2]:= v;
```

```
procedure ST(k,vl,vr); value k,vl,vr; integer k,vl,vr;
begin C[k,1]:= vl; C[k,2]:= vr end;

integer procedure SAVE(F); value F; integer F;
begin integer k; ERROR(F < 0,↓ F < 0 in SAVE↑);
   k:= LHS(free cell); ST(free cell,F,last name);
   SAVE:= last name:= free cell;
   COLLECT GARBAGE(0,auxiliary array,k)
end SAVE;

procedure REMOVE;
begin join to free space(last name);
   last name:= RHS(last name);
   ERROR(last name = 2, ↓REMOVE not appropriate↑)
end REMOVE;

procedure join to free space(k); value k; integer k;
begin STIL(last free cell,k); last free cell:= k end;

integer procedure STORE(A,t,B); value A,t,B; integer A,t,B;
begin integer k; ERROR(A < 0 ∨ B < 0,
   ↓A or B not appropriate in STORE↑);
   STORE:= free cell; k:= LHS(free cell);
   ST(free cell,A,B); ST TYPE(free cell,t);
   auxiliary array[1]:= free cell;
   COLLECT GARBAGE(1,auxiliary array,k)
end STORE;

procedure ST TYPE(k,t); value k,t; integer k,t;
Ctype[k]:= t;

integer procedure AV(l,r); value l,r; integer l,r;
AV:= STORE(l,algebraic variable,r);
```

```
Boolean procedure DYADIC OP(t); value t; integer t;
DYADIC OP:= t = sum ∨ t = product;

procedure COLLECT GARBAGE(n,arr,fc); value n; integer n,fc;
integer array arr;
begin integer i;
  procedure TRACE(F); value F; integer F;
  if F > 0 then
  begin if ¬ traced[F] then
    begin integer t,A,B; t:= TYPE(F,A,B);
      if DYADIC OP(t) then begin TRACE(A); TRACE(B) end;
      traced[F]:= true
  end end TRACE;
  procedure DUMP;
  begin integer i,j; PR nlcr; PR string(⊧free cell = ⊧); PR int num(free cell);
    PR string(⊧ last free cell = ⊧); PR int num(last free cell);
    PR string(⊧ last name = ⊧); PR int num(last name);
    PR string(⊧ ptr of stack = ⊧); PR int num(pointer of stack);
    for i:= 1 step 1 until max of C do traced[i]:= false;
    i:= last name; for i:= i while i ≠ 0 do
    begin traced[i]:= true; i:= RHS(i) end;
    i:= pointer of stack; for i:= i while i ≠ 0 do
    begin traced[i]:= true; i:= RHS(i) end;
    i:= free cell; for i:= i while i ≠ last free cell do
    begin traced[i]:= true; i:= LHS(i) end;
    traced[last free cell]:= true;
    for i:= 1 step 1 until max of C do
    begin PR nlcr; PR int num(i);
      PR string(⊧        ⊧); PR int num(LHS(i));
      PR string(⊧        ⊧); PR int num(RHS(i));
      if ¬ traced[i] then begin PR string(⊧        ⊧); PR int num(Ctype[i]) end
    end
  end DUMP;
```

```
  if free cell ≠ last free cell then free cell:= fc else
  begin DUMP; free cell:= 0;
    for i:= 1 step 1 until max of C do traced[i]:= false;
    for i:= 1 step 1 until n do TRACE(arr[i]);
    i:= last name; for i:= i while i ≠ 0 do
    begin TRACE(LHS(i)); traced[i]:= true; i:= RHS(i) end;
    i:= pointer of stack; for i:= i while i ≠ 0 do
    begin traced[i]:= true; i:= RHS(i) end;
    for i:= 1 step 1 until max of C do
    if ¬ traced[i] then
    begin if free cell = 0 then free cell:= last free cell:= i else
      join to free space(i)
    end; ERROR(free cell = 0,⟨no space left⟩); DUMP
end end COLLECT GARBAGE;

integer procedure TYPE(F,A,B); value F; integer F,A,B;
begin ERROR(F ≤ 0 ∨ F > max of C,⟨F not appropriate in TYPE⟩);
  A:= LHS(F); B:= RHS(F); TYPE:= Ctype[F]
end TYPE;

Boolean procedure DE(first time,f,F); value first time;
  Boolean first time; integer f,F;
begin if first time then
  begin integer k; k:= LHS(free cell); ST(free cell,last name,pointer of stack);
    pointer of stack:= free cell; COLLECT GARBAGE(0,auxiliary array,k)
  end;
  f:= - SAVE(F); DE:= false
end DE;
```

```
procedure ERASE;
begin integer st; ERROR(pointer of stack ≤ 1,{ERASE not appropriate});
   join to free space(pointer of stack); st:= LHS(pointer of stack);
   pointer of stack:= RHS(pointer of stack);
   for st:= st while st ≠ last name do REMOVE;
end ERASE;

integer procedure ASSIGN(f,F); value f,F; integer f,F;
begin ERROR(f < - max of C ∨ f ≥ 0,
   {name not appropriate in ASSIGN});
   ASSIGN:= F; STIL(-f,F)
end ASSIGN;

integer procedure ERROR(b,s); Boolean b; string s;
if b then
begin PR nlcr; PR string(s); EXIT; ERROR:= 1 end;

integer procedure V(f); value f; integer f;
V:= if f ≥ 0 then ERROR(true,{name ≥ 0 in V}) else
   LHS(-f);

integer procedure S(A,B); integer A,B;
begin integer A1,B1; B1:= B; SAVE(B1); A1:= A; REMOVE;
   S:= if A1 = ZERO then B1 else if B1 = ZERO then A1 else
   STORE(A1,sum,B1)
end S;
```

```
integer procedure P(A,B); integer A,B;
begin integer A1,B1; B1:= B; SAVE(B1); A1:= A; REMOVE;
  P:= if A1= ZERO ∨ B1 = ZERO then ZERO else
  if A1 = ONE then B1 else if B1 = ONE then A1 else
  STORE(A1,product,B1)
end P;

procedure OUTPUT(f,OUTPUT VARIABLE); value f; integer f;
procedure OUTPUT VARIABLE;
begin procedure OP(F,type); value F,type; integer F,type;
  begin integer t,A,B;
    procedure LBR; if t < type then PR string(⟨(⟩);
    procedure RBR; if t < type then PR string(⟨)⟩);
    t:= TYPE(F,A,B);
    if t = algebraic variable then OUTPUT VARIABLE(F) else
    if DYADIC OP(t) then
    begin LBR; OP(A,t); if t = sum then PR string(⟨+⟩) else
      PR string(⟨×⟩); OP(B,t); RBR
    end else ERROR(true,⟨F not appropriate in OUTPUT⟩)
  end OP;
  OP(V(f),0)
end OUTPUT;

procedure PR string(s); string s;
begin PRINTTEXT(s); PUTEXT(s) end;
procedure PR nlcr; PR string(⟨
⟩);
procedure PR num(a); value a;real a;
begin PRINT(a); PUNCH(a) end;
```

```
procedure PR int num(a); value a; integer a;
begin integer b; if a < 0 then begin PR string(«-»); a:= -a end;
  if a ≤ 9 then PR sym(a) else
  begin b:= a : 10; a:= a - b × 10; PR int num(b); PR sym(a) end
end;
procedure PR sym(a); value a; integer a;
begin PRSYM(a); PUSYM(a) end;
```

ACTUAL PROGRAM:

```
begin integer x,y,f;
    procedure OV(F); value F; integer F;
  begin integer A,t,B; t:= TYPE(F,A,B);
    if B ≤ 1 then PR int num(B) else
    if B = 2 then PR string(«x») else
    if B = 3 then PR string(«y») else
    ERROR(true,«error in output»)
  end;

  procedure PRINT(x,s); integer x; string s;
  begin PR nlcr; PR string(s); PR string(« (name: »);
    PR int num(x); PR string(« ) (value: »); PR int num(V(x));
    PR string(« ) formula: »); OUTPUT(x,OV)
  end;

  max of C:= 15; INITIALIZE;
  DE(DE(DE(true,x,AV(0,2)),y,AV(0,3)),f,0);
  PR nlcr; PR string(«Results RPR 181168/02»);
  ASSIGN(f,S(P(V(x),V(x)),P(V(y),V(y)))); PRINT(f,«f =»); ASSIGN(f,ZERO);
  ASSIGN(f,S(P(V(x),V(x)),P(V(y),V(y)))); PRINT(f,«f =»);
  ERASE;
```

```
      max of C:= 20; INITIALIZE;
      DE(DE(DE(true,x,AV(0,2)),y,AV(0,3)),f,S(V(x),V(y)));
      PRINT(f,⊰f =⊱);
      ASSIGN(f,S(P(V(f),P(V(x),V(y))),
                  S(P(V(f),V(f)),
                    S(P(S(V(f),P(V(x),V(y))),V(zero)),
                      P(V(x),V(f))
               )    ) ) );
      PRINT(f,⊰f =⊱);
      ERASE;
    end
end end      100
```

```
Results RPR 181168/02
f = (name: -11 ) (value: 15 ) formula: xxx+yxy
free cell = 14 last free cell = 14 last name = 14 ptr of stack = 6
1       0       0
2       0       1       1
3       2       0
4       0       0       1
5       4       3
6       5       1
7       0       2       1
8       7       5
9       0       3       1
10      9       8
11      4       10
12      7       7       3
13      9       9       3
14      9       11
15      12      13      2
```

```
free cell = 12 last free cell = 15 last name = 14 ptr of stack = 6
1       0       0
2       0       1       1
3       2       0
4       0       0       1
5       4       3
6       5       1
7       0       2       1
8       7       5
9       0       3       1
10      9       8
11      4       10
12      13      7
13      15      9
14      9       11
15      12      13
f = (name: -11 ) (value: 15 ) formula: x×x+y×y
f = (name: -13 ) (value: 12 ) formula: x+y
free cell = 18 last free cell = 18 last name = 18 ptr of stack = 6
1       0       0
2       0       1       1
3       2       0
4       0       0       1
5       4       3
6       5       1
7       0       2       1
8       7       5
9       0       3       1
10      9       8
11      12      19      2
12      7       9       2
```

```
13      12      10
14      7       9       3
15      7       12      3
16      17      13
17      20      15      2
18      14      16
19      7       9       3
20      12      12      3
free cell = 11 last free cell = 19 last name = 18 ptr of stack = 6
1       0       0
2       0       1       1
3       2       0
4       0       0       1
5       4       3
6       5       1
7       0       2       1
8       7       5
9       0       3       1
10      9       8
11      19      19
12      7       9       2
13      12      10
14      7       9       3
15      7       12      3
16      17      13
17      20      15      2
18      14      16
19      7       9
20      12      12      3
f = (name: -13 ) (value: 19 ) formula: (x+y)×x×y+(x+y)×(x+y)+x×(x+y)
```

13. Discussion of the actual program and its results

The actual program chosen performs the same formula manipulations as the actual program discussed in section 3. The garbage is now formed by means of a reassignation of *f*, an *ERASE* call and by execution of the Formula expression:

P(S(V(f),P(V(x),V(y))),V(zero)),

which creates the "garbage" formula: (f+(x×y)).

The output consists again of two parts:

1. The pointer to the name, the pointer to the value and the ordinary appearance of a formula are printed.
2. The contents of the storage cells (*C* and *C type*) is printed immediately before and immediately after a garbage collection. These results are preceded by the values of *free cell*, *last free cell*, *last name*, and *ptr of stack*.

Since the values of the formulae are fixed in the garbage-collection system we can now almost return to the old situation where there was no need for surrounding formula names with "*V(*" and "*)*".
This can be accomplished by introducing besides *x*, *y* and *f*, the integer variables *X*, *Y*, *F*, the latter ones for holding the pointers to the values
The last example of the actual program might then read:

```
begin integer x,y,f,X,Y,F;
    DE(DE(DE(true,x,AV(0,2)),y,AV(0,3)),f,S(V(x),V(y)));
    X:= V(x); Y:= V(y); F:= V(f);
    ASSIGN(f,S(P(F,P(X,Y)),
              S(P(F,F),
               S(P(S(F,P(X,Y)),ZERO),
                 P(X,F)
          )   )));
    PRINT (f, ǂ f ǂ );
    ERASE
end
```

Note, that a statement: *F:= S(P(F,P(X,Y)),...,* would be erroneous; therefore, another assignment statement is necessary:

DOUBLE ASSIGN(f,F,S(P(F,P(X,Y)),...),

which not only changes the value of the name of *f* but changes the value of *F* also.

In order to refine the definition of Formula expression of section 3 to cope with this new situation we change the syntactical rule for <Value of a formula variable> into:

<Value of a formula variable>::= *V(*<formula variable>*)*|<Value variable>

and add the following syntactical rule:

<Value variable>::= <variable>.

14. The new derivative process

A procedure for calculating a derivative is easily written down. In order to be able to write:

DER(S(F,P(F,X)),x),

we save explicitly the value of the first actual parameter of *DER*. So, each call of *DER* involves one new name; note that each call of the *DER* of section 10 involves a number of new names equal to the *depth* of the formula to be differentiated.

```
integer procedure DER(F,x); value F,x; integer F,x;
begin integer X,A; integer procedure D(F); value F; integer F;
  begin integer t,A,B; t:= TYPE(F,A,B);
    D:= if F = X then ONE else
        if t = sum then S(D(A),D(B)) else
        if t = product then S(P(D(A),B),P(A,D(B))) else ZERO
  end D;
  X:= V(x); SAVE(F); DER:= D(F); REMOVE
end DER;
```

15. Testing the garbage-collection system

For test purposes we declare the following procedure:

```
integer procedure GARBAGE;
begin integer i,n; n:= 0; i:= free cell;
  for i:= i while i ≠ last free cell do
  begin n:= n + 1; i:= LHS(i) end; GARBAGE:= 1;
  for i:= 2 step 1 until n do AV(100,100)
end GARBAGE;
```

Next, we change in the procedurebody of *DER*, the statement *t:= TYPE(F,A,B)* into *t:= GARBAGE * TYPE(F,A,B)*; and we test the procedure *DER* by means of the following actual program:

```
ACTUAL PROGRAM: max of C:= 40; INITIALIZE;
begin integer d,f,x,y,F,X,Y;
  procedure PRINT(s,f); begin PR nlcr; PR string(s); OUTPUT(f,OV) end;
  procedure OV(F); value F; integer F;
  begin integer t,A,B; t:= TYPE(F,A,B);
    if B ≤ 1 then PR int num(B) else
    if B = 2 then PR string(⟨x⟩) else
    if B = 3 then PR string(⟨y⟩)
  end;
  integer procedure SG(A,B); SG:= GARBAGE × S(A,B);
  integer procedure PG(A,B); PG:= GARBAGE × P(A,B);

  DE(DE(DE(DE(true,x,AV(0,2)),y,AV(0,3)),f,SG(V(x),V(y))),d,0);
  X:= V(x); Y:= V(y); F:= V(f); PRINT(⟨f = ⟩,f);
  ASSIGN(d,SG(DER(PG(F,SG(F,PG(F,F))),x),
              DER(PG(F,SG(F,PG(F,F))),y)));
  PRINT(⟨derivative = ⟩,d);
  ERASE
 end
end end      100
```

which, after the dumps, produced by *COLLECT GARBAGE* have been removed, resulted in

f = x+y
derivative = x+y+(x+y)×(x+y)+(x+y)×(1+x+y+x+y)+x+y+(x+y)×(x+y)+(x+y)×(1+x+y+x+y)

It is remarked that the procedure *DER* of section 10 has been tested with a similar procedure *GARBAGE*.
It turned out that the above, free-list technique, procedure was about three times faster than the relocation-technique procedure.

16. Relocation-versus free-list technique

The apparent advantages of the relocation technique with respect to the free-list technique are:

1. direct access to free space;
2. formulae are stored compactly, thus making it easily possible to store more complicated structures as e.g. arrays (coefficients of a truncated power series or of a polynomial).

The apparent disadvantages are:

1. the intricate manner a procedure like *DER* should be constructed;
2. relocating formulae implies creation of more names due to the fact that the tree branches are "slithery";
3. secondary storage is needed for the garbage-collection process thus reducing the speed of this process considerably;
4. a Formula expression of the form:

   S(P(X,Y),U)

   is not possible, while it is with the free-list technique.

The main disadvantage of the free-list technique is that complicated structures as arrays cannot be stored compactly but should be stored in a "pointer-wise" way.

It is clear, however, that the free-list technique will be chosen to be expanded and to be used in the future.

References

[1] R.P. van de Riet, Formula Manipulation in ALGOL 60 part 1,
Mathematical Centre Tracts 17, 189 pp.
Mathematical Centre, Amsterdam 1968.

[2] R.P. van de Riet, Formula Manipulation in ALGOL 60 part 2,
Mathematical Centre Tracts 18, 196 pp.
Mathematical Centre, Amsterdam 1968.

[3] P. Naur (ed.) Revised report on the algorithmic language ALGOL 60,
Regnecentralen, Copenhagen 1962.