CERN - DATA HANDLING DIVISION
DD/73/16
H. Strubbe
June 1973

CALCULATIONS WITH SCHOONSCHIP

CALCULATIONS WITH SCHOONSCHIP*)

H. Strubbe

ABSTRACT

An algebraic manipulation system is a program that allows one to do calculations with formulae, in the same way that the FORTRAN compiler allows one to do calculations with numbers.

SCHOONSCHIP can handle any algebraic problem, at least if the programmer succeeds to present the method of solution in terms of elementary operations like: multiplying polynomials, substituting function, comparing terms, ...

It will be explained how SCHOONSCHIP works internally. In particular the use of memory, the representation of formulae and the overflow to disk will be sketched.

A few examples of different types of calculations will be discussed.

Part of this text will be presented at the "Third Colloquium on Advanced Computing Methods in Theoretical Physics." Marseille 25-29 June 1973.

---

*) SCHOONSCHIP: A CDC 6600 program for symbolic evaluation of algebraic expressions. CERN preprint by M. Veltman, 1967. CERN program library R 201.

1. DIFFERENCE BETWEEN NUMERICAL AND ALGEBRAIC CALCULATIONS

A comparison with a FORTRAN program. FORTRAN program:

```
A = 2
B = 3
C = 1
A1 = A + B
A2 = (A1 + C) * (A1 - C)
```

The intermediate results, line after line, are:

```
A = 2
B = 3
C = 1
A1 = 5
A2 = 24
```

Remarks:

* All variables at the right-hand side of a particular formula have a numerical value before that formula is evaluated.
* Each formula, whatever complicated it is, will reduce to one number in the end.
* Interchanging the order of the statements of the program leads to an error-message of the type: undefined quantity encountered.

The same calculation in SCHOONSCHIP goes exactly in the reversed order, SCHOONSCHIP program:

```
Z   A2 = (A1 + C) * (A1 - C)
ID, A1 = A + B
ID, C = 1
ID, B = 3
ID, A = 2
```

The intermediate results, line after line, are:

```
A2 = A1**2 - C**2
A2 = A**2 + 2 * A * B + B**2 - C**2
A2 = A**2 + 2 * A * B + B**2 - 1
A2 = A**2 + 6 * A + 8
A2 = 24
```

Remarks:

* The length of an evaluated expression is very hard to predict; this leads to special storage problems.
* The final result of the calculation is not necessarily one number.
* The mechanism of the calculation goes via a scanning and substitution procedure:

  Z in the first column indicates that the expression which follows has to be examined for substitutions

  ID in the first two columns defines the following expression as a substitution.

* Each term of the intermediate result is inspected for the next substitution. If its left-hand side matches, the replacement is made.

2. BASIC STATEMENTS IN SCHOONSCHIP

The elementary operations, out of which every SCHOONSCHIP calculation has to be built up, are:

* Multiplication and Addition of Polynomials.

```
Input                                   Output
Z   Z = (A + B)**2                      Z = A**2 + 2 * A * B + B**2
```

The quantities involved can be algebraic symbols, indices, functions or vectors.

* Substitutions.

```
Input                                   Output
Z   Z = F(A,B) - A**2 + F(X,Y)
ID, F(A,B) = A**2 + B**2                Z = B**2 + F(X,Y)
```

The replacement is made when the arguments match.

In contrast to this, arguments can be dummies. They are denoted with a + sign behind their names. U + means: for all values of U.

```
Input                                   Output
Z   Z = F((A + B), (A - B)) + F(X,Y)
ID, F(U+, V+) = U * V                   Z = A**2 - B**2 + X * Y
```

* Pattern Matching.

```
Input                                   Output
Z   Z = A**4 * C * B**2
ID, A**2 * C**2 * B = A2C2B
ID, A**4 * C * B = A4CB
ID, C * B**2 = CB2                      Z = A**4 * CB2
```

A more complicated example:

```
Input                                   Output
Z   Z = A**4 * F(2,4) * C**3            Z = A**4 * F(2,4) * C**3
        + A * F(1,3) * C                    + AFC(1,3)
        + A**3 * F(3,3) * C**4              + A**3 * F(3,3) * C**4
ID, A**N+ * F(N+, M+) * C = AFC(N,M)
```

* Special Commands in order to:

  Calculate traces;
  Compare coefficients of terms;
  Give a weight to terms;
  Rearrange function arguments, etc. .

* Finally, there exists a set of built in functions:

  Gamma matrices;
  Spin $^1/_2$ and spin $^3/_2$ spinors;
  Summation function ($= \Sigma_{j=k}^{\ell} f_j$);
  Kronecker delta, binomial function, etc. .

3. HOW CAN A COMPUTER DO SUCH CALCULATIONS?

3.1 The representation of formulae

Assume we want to represent the polynomial:

```
2.1 * A**3 * B * C**4 + 7.3 * A**4 * B**4 * C**4 * Y**2 * U * V
```

A possible convention could be to write in memory the following string

```
2.1, A, 3, B, 1, C, 4, 0, 7.3, A, 4, B, 4, C, 4, Y, 2, U, 1, V, 1, 0
```

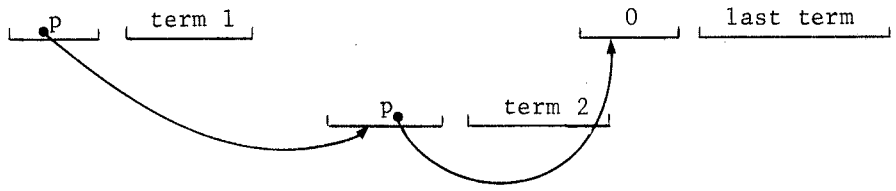
Remarks:

* The coefficients are double precision floating point numbers.
* Rather than using the actual name of each variable, it is sufficient to make a namelist and refer to a variable by its number in the list.
* In order to save space, several quantities can be packed together in one memory word.

* As the number of terms and their length are not known in advance, it is impossible to set up from the beginning an array with fixed locations for each term.

* The zero was put in as separator between terms. This is not very efficient. In order to find the next term, the entire previous term has to be fetched from storage and inspected for a possible zero.

A much more appropriate way is the use of a "POINTER" (symbolized by an arrow). A pointer is nothing but the address of the next term. In other words, we can go from one term to the next, simply by following pointers.



By convention, a pointer with value zero indicates the last term of the expression.

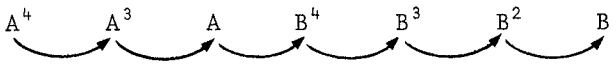
In order to avoid drawing those arrows all the time we will also use another notation: P(i,j).

P(1,j) is the first terms
P(i,j) points to P(j,k)
P(j,0) is the end.

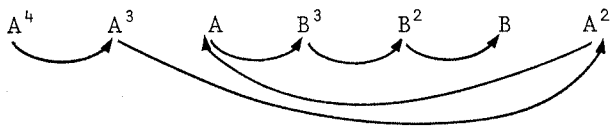
So the same structure as above is denoted by

P(1,3) Term 1, P(2,0) Last term, P(3,2) Term 2.

Another advantage of the use of pointers is that it makes the ordering of terms very easy. This is e.g. necessary for the terms in the output space. Assume we have the string:



Now the term A^2 comes in. Rather than moving all terms in order to allow A^2 to sit inbetween A^3 and A, it is sufficient to change one pointer:



Finally, we need a way to represent expressions in brackets. This is done by defining sub-expressions. The name of such sub-expression starts with a $.

Consider again the example:

```
Z   A2 = (A1 + C) * (A1 - C)
ID, A1 = A + B
ID, C = 3
```

This will be translated in the internal SCHOONSCHIP notation as follows:

```
INPUT ARRAY

A2 = P(1,0)(1.) * $1**1 * $2**1
$1 = P(2,3)(1.) * A1**1, P(3,0)(1.) * C**1
$2 = P(4,5)(1.) * A1**1, P(5,0)(-1.) * C**1
$3 = P(6,7)(1.) * A**1, P(7,0)(1.) * B**1
$4 = P(8,0)(3.)

SUBSTITUTION ARRAY

A1 = $3
C  = $4
```

When the translation of the input is finished, the calculation itself can start. This is the moment to replace the overlay IN, which did the work up till now, by the overlay EXEC.

The allocation of memory for the different arrays and overlays is given in appendix 1.

3.2 The calculation mechanism

The calculation mechanism can be formulated very easily:

a) the Z-expression is looked at.
b) $ expressions are immediately substituted.
c) for each term a scan through the substitution array is made. If one of the left-hand sides matches, that part of the term is replaced by the corresponding $ expression.
d) go back to b) till all substitutions are performed.

A major problem with algebra calculations is the fact that results tend to get extremely long and can no more fit into memory. Then they have to go on disk. This slows down the calculation considerably. Consequently, it is of crucial importance that an algebra system avoids intermediate expressions as much as possible.

Consider the following example:

```
Z   EXP = (A + B + .... + Y + Z)**4
ID, A = Z
ID, B = Z
---
ID, Y = Z
```

It would be very inefficient to calculate (A + B + ... + Y + Z)**4, then to do the first substitution on this result, then the second substitution on the new result, ...

The reason is that each of those intermediate results is so long that they will reside on disk.

The way SCHOONSCHIP performs such a calculation is the following:

```
EXP = P(1,0)(1.) * $1**4
$1  = P(2,3)(1.) * A**1, P(3.4)(1.) * B**1, P(4,5)(1.) * C**1, ...
```

EXP is generated term by term (by specifying pointers to the appropriate factors).

The first term is:

P(53,2) P(54,2) P(55,2) P(56,2)

what will lead to A^4.

This term is then looked at for substitutions. All substitutions are applied and worked out before the next term is generated:

P(..,2) P(..,2) P(..,2) P(..,3) or A^3B.

Now this term is inspected for substitutions, etc..

The order in which the terms are generated is:

P(..,2) P(..,2) P(..,2) P(..,4) or A^3C

etc.

P(..,2) P(..,2) P(..,2) P(..,27) or A^3Z
P(..,2) P(..,2) P(..,3) P(..,2) or A^2BA

till

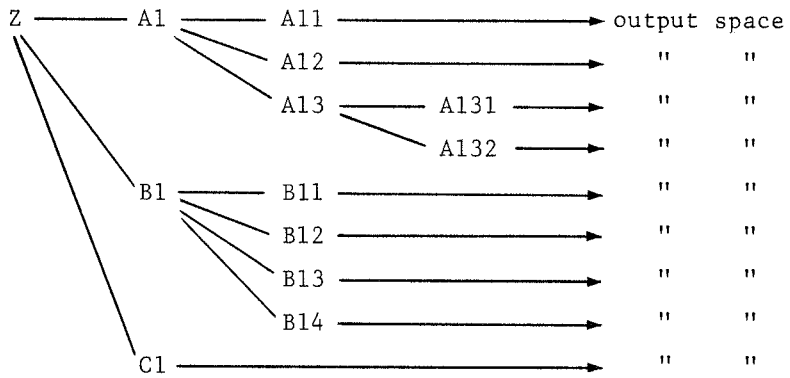
P(..,27) P(..,27) P(..,27) P(..,27) or Z^4

This generating mechanism is very easy because it simply has to follow the pointers of $1.

The important point is that we have only one term at the time in memory, rather than long intermediate expressions. The same philosophy applies to new expressions, arising from substitutions. They are also dealt with term by term. This leads roughly to a tree structure:

```
Z   Z = A1 + B1 + C1
ID, A1 = A11 + A12 + A13
ID, A13 = A131 + A132
ID, B1 = B11 + B12 + B13 + B14
```

will produce terms for the output in the following order:



All these terms overwrite each other as soon as they are no more needed.

3.3 Filling the output space

At the moment terms arrive in the output, it is checked whether they add up with a previous term. Three cases are possible.

Assume A**4 + 3 * A**2 + A is already in the output space:

P(1,2)(1.) * A**4, P(2,3)(3.) * A**2, P(3,0)(1.) * A**1

a) A**3 comes in:
P(1,4)(1.) * A**4, P(2,3)(3.) * A**2, P(3,0)(1.) * A**1, P(4,2)(1.) * A**3

b) -3 * A**2 comes in:
P(1,3)(1.) * A**4, P(2,3)(3.) * A**2, P(3,0)(1.) * A**1
N.B. the A^2 term is no more connected.

c) A**1 comes in:
P(1,2)(1.) * A**4, P(2,3)(3.) * A**2, P(3,0)(2.) * A**1

When the output is full, "Garbage Collection" takes place (non connected terms are removed). If this does not help, the disk is used as overflow array.

It is very unlikely that overflow occurs in the input array (i.e. the initial expression -- read in on cards -- is too long). One can easily get around this by reading in that expression in several pieces.

It is also very unlikely to have overflow in the execution space, as only one term is dealt with at the time.

Overflow in the output array happens very often. The following mechanism comes then into action.

Assume the output is nearly full and contains A**3 as only term depending on A. As the calculation proceeds, the following terms are produced:

A**4 : can still fit in memory, but occupied the last free words
A**2 : cannot be kept in memory. Goes to the overflow tape (tape 4, usually a disk file)
A**3 : adds up with the A**3 in core
A**5 : goes to tape 4
A**2 : goes to tape 4

Each term is compared with all terms in the output space. If it does not add up, it is sent to the overflow tape. However, no test is made whether that term adds up with a term, already stored on that tape.

In other words, tape 4 will contain in this example:

A**2, A**5, A**2.

All terms on the overflow tape are different from those in the output space.

When all terms are generated, the overlay EXEC is replaced by the overlay OUT.

This overlay then empties the output space (e.g. by printing).

Next terms are read in from tape 4 and put in the output space, while addition and cancellation can occur. New overflow is written on tape 5.

The output space is again emptied and by repeatedly interchanging the function of tape 4 and tape 5 this mechanism can go on till all terms are printed.

3.4 Linking of SCHOONSCHIP programs

Rather than printing the content of the output space, it is also possible to save it for the next part of the calculation or for a later run.

At the end of a calculation step one can transfer results from the output back to the input: then a new set of substitutions can be applied on it. Example:

```
Z  EXP(U,V) = U * V
*  NEXT
Z  PR = (EXP((A + B), (A - B)))**2
*  END
```

If the result to be transferred is too big for the input space, that result will be written on disk, and only one buffer of it will be in memory.

This explains how it is anyway possible to multiply three expressions -- each one too big to fit in memory -- with each other: each expression has only one buffer in memory.

Obviously this requires an additional amount of administration, specifying which expressions are in core or on tape, which part of them is in the buffer, etc..

Results to be used in a next run can also be written on disk (and then catalogued as permanent file). This allows one to split up any long calculation into little pieces what -- at least in the CERN computer center -- pays off in turn-around time. (N.B.: CERN X JOB = express job: 30 K memory and 64 sec 6600 CP time, turn-around ≈ 15 min).

An important point in this disk manipulation is that results are written in the internal SCHOONSCHIP notation, rather than in character code, what eliminates repeated conversion.

4. APPLICATIONS

Example A): The Y_{2n} functions.

The formulation of the problem was given by CAMPBELL[1]. The recursive definition of Y_{n+1} is of the following form:

$$Y_{n+1} = \sum_{\alpha+\beta=n} Y_\alpha Y_\beta + \sum_{\alpha+\beta+\gamma+\delta=n} Y_\alpha Y_\beta Y_\gamma Y_\delta$$

$$+ \sum YY' \;\&\; YY'' \;\&\; Y'Y' \;\ldots$$

Remarks (for any symbol manipulation system):

* $\alpha,\beta,\gamma,\delta \geq 0$; the prime indicates differentiation.
* It is necessary to keep a table of all previously calculated Y, Y′, Y″. This makes the given formula of Y_{n+1} a priori very inefficient. Therefore it is worthwhile to spend some effort in finding a better algorithm.
* Using the given formula anyway -- for benchmark purposes -- leads to the following structure of program:
  - There exists a table of
    Y_1, Y_1', Y_1''; Y_2, Y_2', Y_2''; ...; Y_n, Y_n', Y_n''
  - $Y_{n+1} = \Sigma U_\alpha U_\beta + \Sigma U_\alpha U_\beta U_\gamma U_\delta + \ldots.$
  - Order $\alpha \leq \beta \leq \gamma \leq \delta$ and add equal terms

- replace U_j by Y_j; U'_j by Y'_j; ... with the help of the table
- compute Y'_{n+1}
- compute Y''_{n+1}
- add these results to the table.

* It would be a waste of time of compute Y_3Y_7, Y_7Y_3, and to see only at the very end that those two expressions add up. Therefore, the U functions have to be introduced. In other words, it is the responsability of the programmer to carefully evaluate and minimize the amount of work he requires his program to do.

Remarks (specific for SCHOONSCHIP):

* Functions are considered to be non-commutative: U(3) * U(7) + U(7) * U(3) will not add up.

* An efficient way to get around this is to write:
  $Y_{n+1} = \Sigma U2(\alpha,\beta) + \Sigma U4(\alpha,\beta,\gamma,\delta) + \ldots$
  and then to use the instruction SYMXX. This command treats the functions as symmetrical with respect to the interchange of two arguments and places the arguments in a unique order.
  Then * YEP, the instruction to add equal terms, is given.
  Only now the substitution:

  ID, U4(AL +, BE +, GA +, DE +) = Y(AL) * Y(BE) * Y(GA) * Y(DE)
  (and similarly for the other terms)

  is applied.

* Differentiation is not a built in operation, but can easily be performed by means of the following substitutions:

```
Z   YP = Y * DIF
ID, A**N+ *DIF = N * A**N/A * AP + A**N * DIF
ID, B**N + *DIF = N * B**N/B * BP + B**N * DIF
ID, DIF = 0
```

  Here, YP stands for $\partial Y/\partial X$; AP for $\partial A/\partial X$, etc.. It is assumed that Y is a polynomial in A and B (both implicitly depending on X).

* To compute 10 Y functions, we used
  - 24 sec of CP time on a CDC 6600
  - 7000 words of Workspace. The program itself takes 17K when overlayed or 23K when not overlayed (words of 60 bits).

Example B): An eigenvalue problem.

Attention was drawn to this topic by B. SCHORR[2]. Mathematical formulation of the problem:

To determine the distribution of the statistic of a certain two-dimensional goodness-of-fit test of the von Mises type, the eigenvalues λ of the following integral equations are of great importance:

$$\phi(t_1,t_2) = \lambda \int_0^1 \int_0^1 K_1(s_1,s_2;\, t_1,t_2)\, \phi(s_1,s_2)\, ds_1 ds_2$$

where

$$(1) \qquad K_1(s_1,s_2;\, t_1,t_2) = \min(s_1,t_1)\, \min(s_2,t_2) - s_1 s_2 t_1 t_2$$

If

$$(2) \qquad K_{n+1}(s_1,s_2;\, t_1,t_2) = \int_0^1 \int_0^1 K_n(s_1,s_2;\, u_1,u_2).K_1(u_1,u_2;\, t_1,t_2)\, du_1 du_2$$

for $n = 1,2, \ldots$, it can be shown that

$$(3) \qquad L_n = \int_0^1 \int_0^1 K_n(s_1,s_2;\, s_1,s_2)\, ds_1 ds_2 = \sum_{j=1}^{\infty} \frac{1}{\lambda_j^n}, \qquad n = 1, 2, \ldots$$

Suppose L_n is known, then λ_j can be calculated.

N.B. $\min(a,b) = a\, H(b,a) + b\, H(a,b)$, with H being a theta function:

$H(a,b) = 1$ when $a > b$
$\frac{1}{2}$ when $a = b$
0 when $a < b$

Remarks:

* This calculation can be performed by repetively applying the following formulae:

$$\int_0^1 u_1^n\, H(s_1,u_1) * H(t_1,u_1)\, du_1$$

$$= \frac{s_1^{n+1}}{n+1} + \frac{t_1^{n+1} - s_1^{n+1}}{n+1} * H(s_1,t_1)$$

$$\int_0^1 u_1^n\, H(s_1,u_1)\, du_1 = \frac{s^{n+1}}{n+1}$$

The idea is to punch a card, e.g. I(2,2,3,1,2), and add to it the SINAC deck (300 cards) in order to obtain the value of the integral.

Remarks:

* The z-integration will depend on the value of r. Assume $0 < r \leq 4$. FORTRAN would use something like

  GO TO (1,2,3,4), R

  in order to select the needed substitution. The standard way to do this in SCHOONSCHIP is:

```
ID, Z**4 * B**N+*GAM**K+ = F4(N,K)
ID, Z**3 * B**N+*GAM**K+ = F3(N,K)
ID, Z**2 * B**N+*GAM**K+ = F2(N,K)
ID, Z**1 * B**N+*GAM**K+ = F1(N,K)
```

* Another criterion of selecting terms can be based on the numerical value of its coefficient

```
ID, IFGRE, 3.1416, AA
```

  All terms with coefficient > 3.1416 get AA appended to them. In order to treat those terms in a special way, the substitutions

```
ID, AA * F(N+, M+) = FGRE(N,M)
ID, F(N+, M+) = FSMA(N,M)
```

  could be given.

* Very complicated logic can be devised, based on the "weight" of terms

```
ID, COUNT, F, A, 3, B, -2, C, 4
```

  For each incoming term the weight is calculated, according to the given assignment:

  weight of A is 3
  weight of B is -2
  weight of C is 4

  In a product, weights of factors are added up. Then F (weight) is appended to each term.

```
A**3 becomes A**3 * F(9)
C * B becomes C * B * F(2)
```

  Next substitutions of the kind

```
ID, F(9) * A**N+ = FF9(A,N)
```

  can be given.

This feature can be very useful when examining the asymptotic behaviour of an expression, e.g. by setting to zero all terms with a negative weight.

* Another facility which is essential for Feynman diagram calculations, is the "rationalization". We mean the expansion of

$$\frac{1}{(x+a)^n(x+b)^m} = \sum_\alpha \frac{p_\alpha}{(x+a)^\alpha} + \sum_\beta \frac{q_\beta}{(x+b)^\beta}$$

where p_α and q_α depend only on (b-a).

As SCHOONSCHIP cannot divide polynomials, a special notation has to be used: Set XA = X + A; XB = X + B; BA = B - A; then XA**N * XB**M will be expanded by the command:

ID, RATIO, XA, XB, BA.

* Finally, the calculation leads to "subtracted logarithmns", which after integration are expressed in terms of

$$\psi_n^{(m)}(x) = \sum_{j=1}^{n} \frac{x^j}{j^m} \qquad \text{with } x = \text{number} .$$

Such expressions can be calculated by SCHOONSCHIP:

```
Z   PSI(N+, M+, X+) = DS(J, 1, N, (X**J/(J**M)))
```

but it is more efficient to exploit the fact that SCHOONSCHIP is linked to FORTRAN. The statement

```
PSI(N+, M+, X+) = DX(N, M, X)
```

generates a call to the FORTRAN subroutine EXTRA. This subroutine has to be written by the programmer and is added to SCHOONSCHIP. The result of EXTRA is then returned as the value of DX.

This feature allows one to do all formula manipulation in SCHOONSCHIP and all related numerical calculations in FORTRAN.

REFERENCES

1) J.A. Campbell, J. Comput. Phys. 10, 308 (1972).

2) B. Schorr, private communication.

3) A. Petermann, CERN preprint TH 1451, 1972.
D. Maison and A. Petermann, CERN preprint DD

APPENDIX 2: SCHOONSCHIP program for the eigenvalue problem

```
TAPE KEEP 1
TAPE COPY

C  THIS SET OF INSTRUCTIONS AND SUBSTITUTIONS IS APPLIED REPETITIVELY AND
C  GENERATES COPY TAPE 1 FOR NEXT ITERATION.

KEEP ITER,K,K1
* NEXT

PRINT 1,INPUT
Z ITER=ITER+1

C  INTEGRATION OF  K , SEE FORMULA 3.
Z L(S1,S2)=K(S1,S2,S1,S2)
ID, H(S1+,S2+)=
AL,          S1**K +=1/(K+1)
AL,          S2**K +=1/(K+1)

PRINT 2,OUTPUT
N 25
* YEP

PRINT 1,INPUT
N R
KEEP ITER,K,K1
* NEXT

PRINT 1,INPUT
C  CONSTRUCTION OF THE  FAT  K , SEE FORMULA 2.
Z K(S1,S2,T1,T2)=K(S1,S2,U1,U2)*K1(T1,T2,U1,U2)

ID, AINBE, U1** +**H(S1,U1)*K(T1,U1)=S1*S1**K/(K+1)+(T1*T1**K-S1*S1**K)*
           K(S1,T1)/(K+1)
AL, AINBE, U2** +**H(S2,U2)*K(T2,U2)=S2*S2**K/(K+1)+(T2*T2**K-S2*S2**K)*
           K(S2,T2)/(K+1)
AL, AINBE, H(S1,U1)*K(T1,U1)=S1+(T1-S1)*H(S1,T1)
AL, AINBE, H(S2,U2)*K(T2,U2)=S2+(T2-S2)*H(S2,T2)
AL,          U1** +**K(S1+,U1)=S1*S1**K/(K+1)
AL,          U2** +**K(S2+,U2)=S2*S2**K/(K+1)
AL,          U1**K +=1/(K+1)
AL,          U2** +=1/(K+1)
AL,K(S1+,S2+)=S1

TAPE END
T
```

```
INDEX K
SYMBOLS G1,G2,S1,S2,T1,T2
FUNCTIONS H,L

C  DEFINITION OF  K1  , SEL FORMULA 1.
Z K1(S1,S2,T1,T2)=S1*S2-S1*S2*T1*T2+S2*T1*H(S1,T1)-S1*S2*H(S1,T1)+
 S1*T2*H(S2,T2)-S1*S2*H(S2,T2)+T1*T2*H(S1,T1)*H(S2,T2)+S1*S2*H(S1,T1)*
 H(S2,T2)-S1*T2*H(S1,T1)*H(S2,T2)-S2*T1*H(S1,T1)*H(S2,T2)

KEEP K1
* NEXT
C  INITIALIZATION.

Z TIEM=0
Z K(S1,S2,T1,T2)=K1(S1,S2,T1,T2)
PRINT KLIST
TAPE REWIND

C  THE SET OF INSTRUCTIONS, COPIED ON TAPE, IS NOW READ IN AND APPLIED.
C  BUT ITS PRINTING IS SUPPRESSED.
TAPE READ
TAPE REWIND
TAPE READ
TAPE REWIND
TAPE READ
TAPE REWIND
TAPE READ
TAPE REWIND
* END
```