STANFORD ARTIFICIAL INTELLIGENCE PROJECT August 17, 1964
Memo No. 20

SOURCE LANGUAGE OPTIMIZATION OF FOR-LOOPS

by R. Reddy

Abstract: Program execution time can be reduced, by a considerable amount, by optimizing the 'For-loops' of Algol Programs. By judicious use of index-registers and by evaluating all the sub-expressions whose values are not altered within the 'For-loop', such optimization can be achieved.

In this project we develop an algorithm to optimize Algol Programs in List-structure form and generate a new source language program, which contains the "desired contents in the index registers" as a part of the For-clause of the For-statement and additional statements for evaluating the same expressions outside the 'For-loop'. This optimization is performed only for the innermost 'For-loops'.

The program is written entirely in LISP. Arrays may have any number of subscripts. Further array declarations may have variable dimensions. (Dynamic allocation of storage).

The program does not try to optimize arithmetic expressions (This has already been extensively investigated).

The research reported here was supported in part by the Advanced Research Project Agency of the Office of the Secretary of Defense. (SD-183)

TABLE OF CONTENTS

Page

DEFINITIONS i

INTRODUCTION 1

PROCEDURE 3

ALGOL IN LIST STRUCTURE FORM 5

FUNCTIONS 7

a. Description

b. Examples

c. M-expressions

PROGRAM 28

TEST RUN RESULTS 32

CONCLUSIONS 37

Definitions:

Definitions of those phrases which are considered unique to this report are given here. Please refer to Algol 60 report for any others, not explicitly defined.

Left list:

A list of all variables, simple or subscripted, that occur to the left of an assignment statement within a For-loop i.e. list of all variables whose value is altered within the For-loop explicitly.

Array - reference list:

List of all subscripted variables that occur within a For-loop.

I-coefficient:

Given any subscripted variable we can consider subscript as consisting of two parts, one that alters every time around the For-loop and the other that remains constant during the execution of the 'For-loop'. The first, which is assumed to be dependent upon the control variable of the 'For-loop' shall be called I-coefficient.

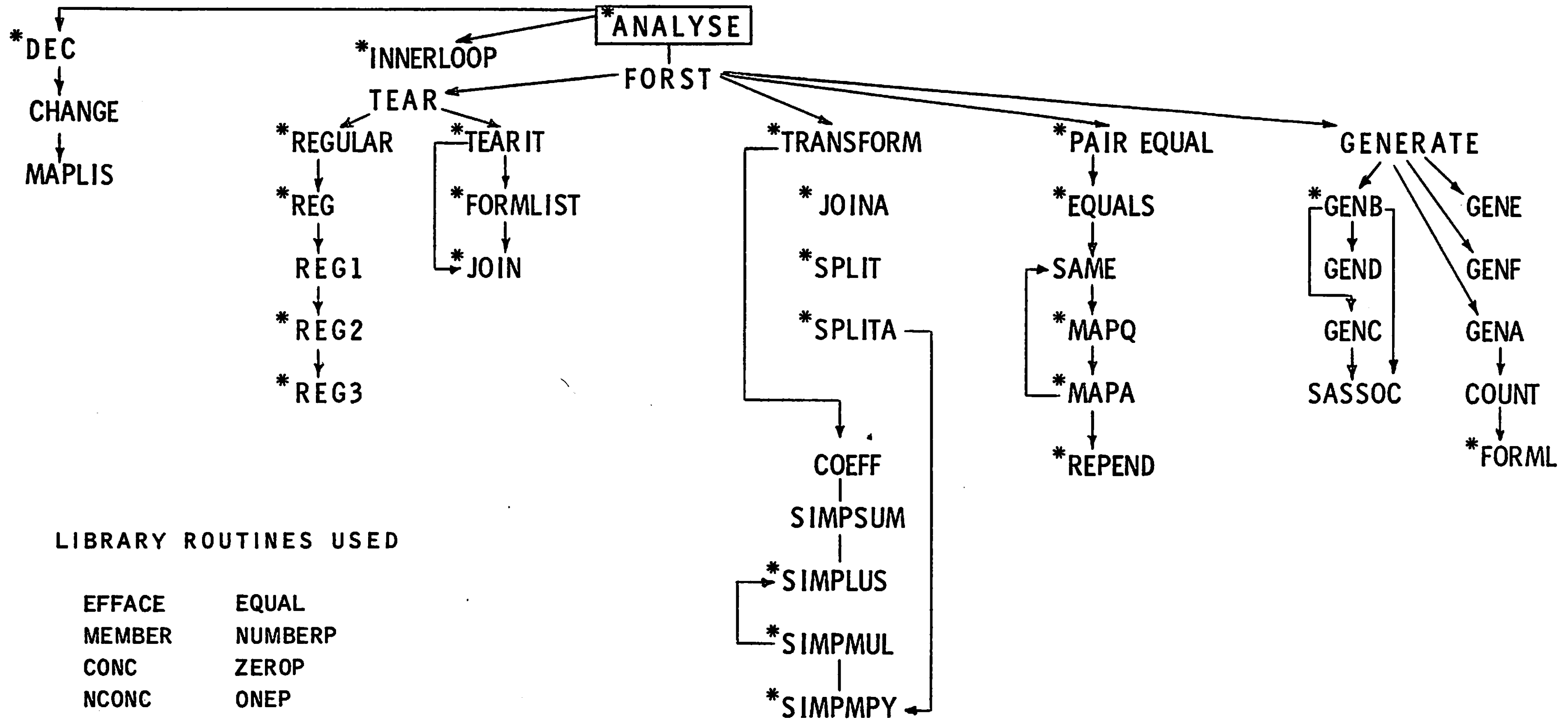
Constant Coefficient:

The second part above shall be called constant coefficient.

Control Variable:

Variable that controls the number of repetitions of the For-loop.

CONNECTION CHART AMONG FUNCTIONS



LIBRARY ROUTINES USED

EFFACE	EQUAL
MEMBER	NUMBERP
CONC	ZEROP
NCONC	ONEP
MAPLIST	ADD1
SUBLIS	SUB1
GENSYM	PLUS
PAIR	DIFFERENCE
	MINUS
	TIMES

*Indicates the functions that call themselves recursively.

SOURCE LANGUAGE OPTIMIZATION OF FOR LOOPS

by R. Reddy

Introduction:

For programs which are to be used over and over again it is desirable to have an efficient object program. However, to achieve this, when using Algol-type languages, one requires the use of highly sophisticated compilers. Such sophistication usually means relatively long compilation times for a given program. One approach suggested to remedy this is to have large core-memory so that both the program and the compiler are resident within the core memory. Then, one can recognize large numbers of special cases and generate efficient object code within a reasonable amount of time.

In the absence of such a large machine, one has to decide which parts of the program use up a major portion of the total execution time and try to optimize those parts. In a large number of programs 'For-loops', and arithmetic expressions, usually contribute most to the execution time.

Usually not much can be done with regard to arithmetic expressions. When one needs the same expression in two parts of the program, and when the value of the expression remains unchanged in between, one can save some time by storing the result in a temporary location and using it when required. However, when these occur explicitly within an arithmetic expression there is no reason to suppose that the programmer is not intelligent enough to recognize them! However, there are obvious cases, like exponentiation, where some compilers use log and antilog routines when it would have been (perhaps) optimal to use direct multiplication.

In programs using subscripted variables, determination of the effective addresses of the subscripted variables usually contributes a fair amount to the total execution time, if one attempts to evaluate them individually. Attempts have been made to use the index registers to speed-up the evaluation. However, one soon runs out of index registers and has to determine how to optimally allocate index registers so as to minimize the execution time. Many times, the subscripts of a variable are expression rather than simple variables. Although such subscript expressions may look radically different it might be possible to evaluate all of them by use of a single index register. For example, consider three arrays of dimensions A[0:9, 0:20], B[0:4, 0:15], C[0:500]. Within a For-loop, with I as a controlled variable, let there exist the following array references A[I + 3,J],B[2I + 7,K], C[10I-9] where J and K remain unchanged within the For-loop. Assuming that the arrays are stored row-wise (i.e., the last subscript varying most rapidly) their effective addresses with respect to zeroth elements

are given by A[10I + (J + 30)], B[10I + (K + 35)] and C[10I - 9]. Within the For-loop only the value of I is changed every time around the loop. Thus, by initializing addresses to A[J + 30], B[K + 35] and C[-9] before entering the For-loop and having the value (10I) within an index register we can treat the above variables the same way as the simple variables.

However, the whole problem of subscript evaluation is complicated by the following facts:

1. We may have no way of knowing whether or not the values of J and K(in the above example) are altered within the 'For-loop' eg. they may be altered while executing a procedure call 'from within' the 'For-loop'.

2. If we know that K and J are altered then the only way out is, of course, to evaluate the subscripted variable every time around the For-loop.

3. We may not have enough index registers to satisfy the requirements.

4. Since we can have any number of subscripts the problem becomes more complex.

5. Dynamic allocation of storage in Algol will mean that we don't know the array sizes at the time of compilation and would have to test for equality of the symbolic expressions that result during the subscript evaluation.

In the following section we outline a procedure that attempts to handle the above difficulties.

Procedure:

Consider an n-dimensional array x with the following upper and lower bounds

$$x[\ell_1:U_1,\ \ell_2:U_2,\ \ell_3:U_3,\ \ldots\ \ell_n:U_n]$$

In Algol we assume that arrays are stored row-wise i.e., the last subscript varies most rapidly. Anytime x is referred to within the program (with n subscripts), the subscript part may be replaced by an equivalent single subscript as shown below.

$$x[I_1,\ I_2,\ \ldots\ldots\ I_n]$$

$$= x[I_n - \ell_n + (U_{n-1} - \ell_{n-1} + 1)(I_{n-1} - \ell_{n-1} + (U_{n-2} - \ell_{n-2} + 1)$$

$$(I_{n-2} - \ell_{n-2} + \ldots(I_2 - \ell_2 + (U_1 - \ell_1 + 1)\ I_1)))\ldots)))$$

By writing $D_n = U_n - \ell_n + 1$

$$= x[I_n - \ell_n + D_{n-1}(I_{n-1} - \ell_{n-1} + D_{n-2}(I_{n-2} - \ell_{n-2} + \ldots +$$

$$(I_2 - \ell_2 + D_1 I_1)))\ldots)]$$

$$= x[I_n + D_{n-1}(I_{n-1} + D_{n-2}(I_{n-2} + \ldots(I_2 + D_1 I)))\ldots) -$$

$$[\ell_n + D_{n-1}\ (\ell_{n-1} + D_{n-2}\ (\ell_{n-2} + \ldots\ldots(\ell_2 + D_1\ell_1)))\ldots)] \qquad (1)$$

The second part of the subscript expression contains only the constants of the array and can be evaluated at the time of execution of the 'Array Declaration' statement (i.e., when the storage is allocated for the given array). Thus, instead of having the address of the first location of array x in its property list we can have the address of $x[\ell_n + D_{n-1}(\ell_{n-1} + D_{n-2}\ (\ell_{n-2} + \ldots + (\ell_2 + D_1\ell_1))\ldots)]$. (Note that this is the address of x[0,0,...0] which itself may not be a valid index).

Thus hereafter we shall concern ourselves with the evaluation of the first part of the subscript expression only.

$$x[I_n + D_{n-1}(I_{n-1} + D_{n-2}(I_{n-2} + \ldots + (I_2 + D_1 I_1))))\ldots))]$$

can be rewritten as

$$x[I_n + D_{n-1}I_{n-1} + D_{n-1}\ D_{n-2}\ I_{n-2} + \ldots + D_{n-1}\ D_{n-2}\ D_{n-3}\ldots D_1 I_1] \qquad (2)$$

which is a more convenient form to operate with since D's and I's may be numbers or symbolic expressions. (Note that the D_i represents the i-th dimension). Each I_i can now be subgrouped into two parts; the I-coefficient I' (the coëfficient of the control-variable of the For loop) and the constant coefficient C. Rewriting (2) we have

$$x[(I'_n + D_{n-1} I'_{n-1} + D_{n-1} D_{n-2} I'_{n-2} + \dots + D_{n-1} \dots D_1 I'_1) +$$

$$(C_n + D_{n-1} C_{n-1} + D_{n-1} D_{n-2} C_{n-2} + \dots + D_{n-1} \dots D_1 C_1)] \quad (3)$$

Now we consider the problem of 'For-loops' within 'For-loops', which may go to any level. Obviously the innermost 'For-loop' will be repeated the maximum number of times, which we shall try to optimize here.

It is possible to consider the other 'For-loops' also in the optimization process, but programs get larger and more complicated. For effective optimization in such cases, it becomes desirable to have the hardware capability of adding a number of index registers to a given address. Otherwise, only optimization that can be achieved for the outer 'For-loops' is the evaluation of those subscripted variables which do not involve the control-variables of lower levels.

For a given 'For-loop' we scan the program and form two lists. A left-parts list, the list of all the variables that occur to the left of an assignment statement and an Array-reference list, the list of all the subscripted variables that occur within the 'For-statement'. Any subscripted variable whose subscripts occur in the left-parts list (i.e., whose value is changed every time the For-loop is excuted) or which contains non-standard operators (like procedure operators and operators other than +, -, x) or which contain subscript expressions which are not amenable to simplification to single subscript form are removed from the Array-reference list. The remaining array references are transformed to single subscript form. All those with the same I-coefficient (coefficient of the control variable of the 'For-loop') are grouped together in the form given by equation 3. The I-coefficient of each group is replaced by a new generated variable and the generated variable is added to the For-clause for index register assignment. The constant coefficient part can be evaluated outside the For-loop, since by choice none of the subscripts occur in the left-parts list. However, this may not be necessary if the constant coefficient happens to be a number or a simple variable. When it is an expression, it is evaluated outside the For-loop and a new generated symbol replaces it. Identical expressions are evaluated only once outside the For-loop.

The whole program is now regrouped together as it was before except for changes in the subscripted variables. The new For-clause will contain the following information for use of the compiler.

1. Original for clause.

2. Functions of the I-coefficient that should be evaluated and placed in Index-registers.

3. Frequency of occurence of each of the above functions to facilitate optimal assignment of index registers.

In front of the 'For-loop' some more assignment statements are added, if required, to evaluate constant coefficient expressions whose value remains unchanged within the 'For-loop'.

Algol in List Structure Form:

To facilitate analysis of the program by using Lisp language, Algol programs are assumed to have been written in Lisp-Algol form. Some of the pertinent details are given in the following page.

The following example illustrates how a matrix multiplication routine coded in Algol 60, would appear when transcribed into Lisp-Algol code.

```
Integer J,K,I; Real Sum;

Real Array A[1:20, 1:20], B[1:20, 1:40], C[1:40, 1:20];

For I ← 1 step 1 until 20 do

   For J ← 1 step 1 until 20 do begin

      A[I,J] ← 0;

      For K ← 1 step 1 until 40 do

      A[I,J] ← A[I,J] + B[I,K] ⊗ C[K,J]; End;
```

```
((Declare (Integer J K I)
          (Real Sum)
          (Real Array (A ( 1 20)( 1 20)) (B ( 1 20) (1 40))
                (C (1 40) ( 1 20))))
(For (I (Step 1 1 20))
   (For (J (Step 1 1 20))
      (( ←(↓A I J ) 0)
        (For (K (Step 1 1 40))
           ( ←(↓A I J ) ( + (↓A I J) ( ⊗ (↓B I K)(↓C K J))))
        )
      )
   )
 ))
```

SALIENT FEATURES OF ALGOL IN LIST FORM

	ALGOL	LISP-ALGOL
Variable	A, SIGMA,	A, SIGMA.....
Subscripted Variable	A[I,J, K + 3]	(↓ A I J (+ K 3))
Expressions E	<Expression> operator <Expression>	(Operator operand1...operandn)
Statements S		
1) assignment	b←a←<Expression>	(←b(←a(Expression)))
2) conditional	If P then S If P then S else R	(IFD P S) (IFT P S R)
3) Go to	GO TO <Label>	(GOTO Label)
4) For	For I←a_1,a_2,...a_n DO S	(FOR (I a_1 a_2....a_n) S)
	For I←E_1 step E_2 until E_3 do S	(FOR (I (STEP E_1 E_2 E_3)) S)
5) Procedure P	P(a_1, a_2, ...a_n)	(P a_1 a_2...a_n)
Compound Statement	CS← Begin $S_1 S_2 ... S_n$ End	(S_1 S_2... S_n)
Block	<Block> ← (Declarations; CS)	((DECLARE (Integer..) (Real..) (Integer Array,...)...) S_1 S_2..S_n)
Label	X:S	(LABEL X) S
Operators	+, -, x, /, DIV, *, MOD	M-exp +, -, x, /, DIV, *, MOD S-exp PLUS, MINUS, DIFFERENCE, TIMES, DIV, POWER, MOD

Analyze [P, AD]:

P is the program to be optimized.

AD is the list of arrays encountered so far..

Analyze makes use of all the remaining routines to optimize the program. It scans through each statement and takes the following actions:

1. If the statement is a 'Declaration' then it modifies its AD list if necessary.

2. If it is a conditional statement it proceeds to analyze the statement, same as P.

3. If it is a For-statement and it is the innermost For-loop then it proceeds to optimize it by using FORST. Otherwise it proceeds to analyze it the same as P.

4. If it is a compound statement, analyze it the same as P.

5. Otherwise i.e., if the statement happens to be a basic statement like assignment, go to, procedure etc., analyze proceeds to examine the next statement.

Analyze [p; ad] : _If_ null [p] _then_ NIL _else if_ eq[caar[p]; DECLARE] then cons[car[p]; analyze [[cdr[p]; dec [cdar[p]; ad]]] _else if_ eq[caar[p]; IFT] ∨ eq[caar[p]; IFD] _then_ cons [cons[caar[p]; analyze [cdar[p], ad]]; analyze[cdr[p], ad] _else if_ eq[caar[p], FOR] then [_if_ innerloop [caddar[p]] _then_ append [Forst [cdar [p]; ad]; analyze [cdr[p], ad]] else cons [cons [caar [p]; cons [cadar [p], analyze [cddar [p]; ad]]]; analyze [cdr [p], ad]]] _else_ cons [car [p]; analyze [cdr [p], ad]].

Innerloop [p]:

Innerloop is a predicate which is true if P is a program which does not contain any 'for' statements within it. It is used to determine whether a given 'for' statement is the innermost 'for' statement.

Examples:

```
Inner loop    ((Declare (Integer I   J))
              (For (I 1 2 3) ← A(I) 0))
              ( ← SUM  A(J))) = F
```

```
Innerloop     (( ← K   I)
              (IFD ( = I  J) (GO TO K))
              ( ← (A I J)  0)) = *T*
```

M-exp

Innerloop [p] = If null [p] then TRUE else if ¬[atom [car[p]]] then innerloop [car[p]] ∧ innerloop [cdr [p]] else if eq[car[p], FOR] then FALSE else if eq [car[p], IFD] ∧ eq[car[p], IFT] then innerloop [cddr[p]] else TRUE

DEC [L, AD]:

Given a list of declarations L, Dec scans the list and picks up all the array declarations, modifies the dimensions so that we have sizes of each dimension of the array. In the calculation of the address of an array the last dimension of the array plays no part. Thus the result only has the remaining array dimensions.

```
DEC [((Integer N)
      (Real Element Sum)
      (Real Array (x 1 10)) (A (1 20) (1 20))
        (R (0 10) ( 0 N)))
      (Integer Array (I (-5 10) (-5 10) (10 20) (0 N) (6 24)))) NIL]
=     [(I 16 16 11(-N(-1))) (x) (A 20) (R 11)]
```

M-exp

Dec[ℓ,ad] = If null [ℓ] then ad else if eq[cadar[ℓ]; ARRAY] then Dec[cdr[ℓ]; nconc[change[cddar[ℓ]; ad] else Dec[cdr[ℓ], ad]

CHANGE [P]

Given a list of upper and lower bounds of the dimensions of the array, change looks at each dimension, subtracts the lower dimension from the upper of all but the last dimension and forms a new list.

Example

```
Change [((x (1 20)) (A (1 20) (1 20)) (R (0 10) ( 0 N)))] =

      ((x) (A 20) (R 11))
```

M-exp

Change [p] = maplist [p; λ[[c]; cons [caar[c]; maplis [cdar[c]; λ[[d];

If onep[caar[d]] then cadar[d] else numberp[caar[d]] then [if numberp

[cadar[d]] then [cadar[d] - [caar[d] -1]] else list [DIFFERENCE;

cadar[D]; [caar[d] -1]]] else if numberp[cadar[d]] then list [DIFFERENCE;

[cadar[d] + 1]; caar[d]] else

list [DIFFERENCE; cadar[d]; list[DIFFERENCE; caar[d]; 1]]

MAPLIS[X, FN]:

Same as Maplist except it ignores the last element of the list X.

M-exp

Maplis [x;fn] = If null [cdr [x]] then NIL else cons [fn[x]; maplis

[cdr[x]; fn]]

FORST [L, AD]:

L is a For Statement to be optimized.

AD is list of Arrays with their dimensions.

FORST performs the following functions:

1. If statement to be repeated is a Block then it modifies its AD as required.

2. Selects all the array references in L that it can modify by using Tear.

3. Converts them to singly subscripted variables by using Transform.

4. Collects together array references that have the same I-coefficient by using pairequal.

5. Assigns a new symbol for each different I-coefficients occuring in the program.

If the constant coefficients are expressions that can be evaluated outside the for loop profitably, it does so. At that stage it looks for equivalent constant-coefficient expressions and represents them by the same symbol.

6. Produces a new program substituting these symbols created in place of the expression.

M-exp

```
Forst [ℓ, ad] = λ[[c]; generate [cadr [ℓ]; car [ℓ]; [λ[[a];

pairequal [pair[a; transform[caar[ℓ]; a;c]]]] [Tear[cadr [ℓ]]]]]

[If eq[caaadr[ℓ]; DECLARE] then  dec[caadr[ℓ],ad] else ad]
```

TEAR [P]:

Given any list P of Algol statements Tear scans through the statements and produces a list of all array references in P that have Regular subscripts (see under function Regular). The array references are numbered depending on which statement they occur in. This is mainly for future use in case one does have sufficient number of index registers and wishes to determine optimal allocation of index register as computation proceeds through the For loop.

M-exp

```
Tear [p] = λ[[c]; regular [car[c]; cadr[c]]] [tearit  [p,0]]
```

TEARIT [P, NO]:

Each statement of P is numbered and by the use of Formlist all the array references (numbered) and all the elements whose value is altered within P (left-list) are formed.

Example

```
Tearit [(( ←(↓A I J) ( + (↓A I J) (x (↓B I K) (↓C K J))))

          ( ←K ( + I 3))

          (IFT ( = (↓A I J) 0) (←(↓A I J) 1) (←( A I J) 0)) 0]
```

results in

```
[((↓A I J)  K (↓A I J) (↓A I J))

 ((↓A I J).0)  ((↓ A I J).0) ((↓ BIK).0) ((↓C K J).0)

 ((↓ A I J).2) ((↓ A I J).2))]
```

M-exp

```
Tearit  [p, no] = If null [p] then list [NIL;NIL] else join [formlist

[car[p]; NIL;NIL; no];  tearit [cdr[p]; no + 1]
```

FORMLIST [P L A NO]

P: Any statment or expression
L: Left-list elements encountered so far
A: Array references encountered so far
NO: Number of this statement

Formlist recursively scans compound statements, assignment statements etc. to pick up all the array references and Left-list elements.

```
    Formlist ((← (↓ A I J) ( + (↓ A I J) (x (↓ B I K) (↓ C K J)))):
              NIL; NIL; 0)
 =  Formlist ((↓ A I J) ( + (↓ A I J) (x (↓ B I K) (↓ C K J))));
              ((↓ A I J)); NIL;0)
 =  Formlist  (( + (↓ A I J) (x(↓B I K) (↓C  K J)))) ((↓ A I J));
              (((↓ A I J). 0));0 )
```

and so on

```
 = (((↓ A I J)) (((↓ A I J).0)  ((↓ A I J).0) ((↓ B I K).0) ((↓C K J).0))
```

M-exp

Formlist [p, ℓ, u, no] =

If atom [p] then list [ℓ,a] else if [atom[car[p]]] then join [formlist [car[p], ℓ u, no]; formlist [cdr[p], ℓ, a, no]] else if eq [car [p], ASSIGN] then formlist [cdr[p], cons[cadr[p],ℓ], a, no] else if eq[car[p], DARROW] then list [ℓ, cons [cons[p, no],a]] else if eq[car[p], DECLARE] then list [ℓ,a] else formlist [cdr [p],ℓ,a,no]

JOIN (L,M):

L is a list of lists

M is a list of lists

Join appends the elements of L to the corresponding elements of M

Example

Join (((A) (A B) (A B C)) ((X Y Z) (X Y Z) (Y Z X)))

= ((A X Y Z) (A B X Y Z) (A B C Y Z X))

M-exp

join [ℓ,m] = if null [ℓ] then NIL else cons [append [car[ℓ]; car [m]];

join [cdr[ℓ]; cdr [m]]]

REGULAR [L,A]

L is the left-list of the For-statement

A is the list of all array references in the For-statement

Any array reference is regular if it satisfies the following conditions:

1) No subscript is an element in the left list i.e., the value of it is not modified with the For-loop.

2) It is an arithmetic expression of one of the following categories.

   a. Constant or a Variable not contained in the left-list

   b. '(x aaaaa)' i.e., Product of Variables

   c. '(± (a or b) (a or b))' i.e. Sum or difference of Products or Variables

   d. '(-(a or b))' i.e. negation of Products or Variables.

Example

REGULAR [(K J (↓A I J) S U M); (((↓X(+ M 3)).2) ((↓B I L (x 3 N)).3) ((↓A I J).3))]

= (((↓X (+ M 3)) . 2) ((↓B I L (x 3 N)).3))

M-exp

Regular [ℓ;a] = If null [ℓ] then NIL else [if reg[cddaar [a]; ℓ]

then cons [caar [a]; regular [ℓ; cdr [a]] else regular [ℓ,cdr[a]]

REG [E L]:

E list of all subscript elements of one array-reference

L is the left list

REG is a predicate which is true if each element of E satisfies conditions (1) and (2) listed under 'REGULAR'.

Examples

Reg [(I L (× 3N)) (K J (↓ A I J) SUM)] = T

Reg [(I J) (K J (↓ A I J) SUM)] = F

M-exp

Reg [a,ℓ] = null[a]∨[regl[car[a],ℓ]∧reg [cdr[a],ℓ]]

REGl [E L]

E is one of the subscripts of an array-reference

L is the left-list

Regl is a predicate which is true if element E satisfies conditions 1 and 2 of Regular

Examples

Regl [(+ M 3) (K J (↓ A I J) SUM)] = T

Regl[J (K J (↓ A I J) SUM)] = F

M-exp

Regl[e,ℓ] = If atom [ℓ] then ¬ number [e,ℓ] else if member [car[ℓ];

(DIFFERENCE PLUS)] then reg2[cadr[ℓ];ℓ] reg 2[caddr[E];ℓ] else if

member [car[ℓ]; (TIMES MINUS)] then reg2 [e,ℓ] else FALSE

REG2[E L]

L is the left list

E must be of the form (2a) or (2b) given under Regular

Reg 2 is a predicate which is true if E of the form above (i.e. variable or product of variables)

Examples

Reg 2 ((+ M 3) (K J (↓ A I J) SUM)) = F

Reg 2((× M 3) (K J (↓ A I J) SUM)) = T

M-exp

Reg2[e,ℓ] .

If atom [e] then ¬[member [e,ℓ]] else if eq [car [e], MINUS] then reg2 [cadr[e]; ℓ] else if eq [car[e], TIMES] then reg 3 [cdr[e],ℓ] else reg 2[car[e],ℓ]∧reg 2 [cdr[e],ℓ]

REG 3 [E L]

L is the left list

E must be list elements of the form (2a) given under Regular

Reg 3 is true if E is a list containing elements which are either constants or variables not contained in L.

Example

Reg 3 (M (K J (↓ A I J) SUM)) = T

Reg 3 (J (K J (↓ A I J) SUM)) = F.

M-exp

Reg 3[e,ℓ]

If null [e] then TRUE else atom [car[e]]∧ ¬[member [car[e],ℓ]]∧ reg3[cdr[e],ℓ]

TRANSFORM [I L A]:

A major routine in the program. L contains list of all array references within the FOR statement. Using the dimensions of each of the arrays L is transformed to a singly subscripted array whose subscript is split into two parts: the I-coefficient and the constant coefficient. The result is a list ((IC CC)...(IC CC)) which contains one element for each Array reference.

Example

Transform [I; ((↓ X I) (↓ X(+ I 3)) (↓ A I J) (↓ A J I) (↓ X(+ (x 20 I) 3)) (↓ A (+ I 4) J)); ((A 20) (X))]

results in ((1 NIL) (1 3) (20 J) (1 (x 20J)) (20 3) (20 (+J 80)))

M-exp

Transform [i,ℓ,a] = If null [ℓ] then NIL else cons [coeff [split [i; cddar [ℓ]];NIL; NIL; sassoc [cadar[ℓ], a, NIL]]; transform [i,cdr [ℓ], a]]

JOINA[L M]:

When a given subscript is an expression one may have to analyze subexpressions separately, so that the I-coefficients and constant coefficient may have to be combined together to form one I-coefficient and one constant coefficient. This is done by joina.

Example

((NIL (+ J 3)) ((+ 3 K) NIL)) = ((+ 3K) (+ J 3)

M-exp

joina [ℓ,m]:

If null [ℓ] *then* NIL *else* cons [(*If* null [car[ℓ]] *then* car [m]

else if null [car[m] *then* car [ℓ] *else* list [car [ℓ]; car[m]]);

joina [cdr[ℓ]; cdr[m]]]

SPLIT (I L):

Given an array reference with a number of subscripts split converts each subscript to I-coefficient and constant coefficient form by the use of splita...

Example

Split ((+I 3) J) = ((I 3) (NIL J))

M-exp

Split [i,ℓ]:

If null [ℓ] then NIL *else* cons [splita[i;car[ℓ]]; split[i; cdr[ℓ]]]

SPLITA(I E):

Given an expression E (one of the subscript elements), splita examines E and separates it into 2 expressions the I-coefficient part and the constant coefficient part.

Examples :

Splita [I; (+ (x 3 I)6)] =(3 6); Splita [I;(+ (x 3 J I) (-(x I 3)))]

= (((x 3 J)-3) NIL)

M-exp

Splita[i,e] =

If atom [e] then [if eq[i;e] then list [1; NIL] else list [NIL;e]] else if eq[car[e]; TIMES] then [if member [i, cdr[e]] then list [simpmpy [efface [i,cdr[e]]; NIL; 1]; NIL] else list [NIL; simpmpy [cdr[e]; NIL, 1]] else if eq[car[e]; DIFFERENCE] then joina [splita[i, cadr[e]]; then joina [splita[i, cadr[e]]; splita [i, list [MINUS; caddr [e]]]] else if eq [car [e]; MINUS] then λ[[c]; maplist [c; λ[[a]; If null [car [a]] then NIL else if numberp [car [a]] then minus [car[a]] else list [MINUS; car [a]]]] [splita [i; cadr e]]else joina [splita [i, cadr [e]]; splita [i; caddr [e]]]

COEFF [S, I, C, L]:

This function takes subscripts s(in modified form) and computes I-coefficient I and constant coefficient C using the dimensions of the array as given by L.

Example:

A[1:20, 1:20] and A[(I + 1); (I + 1)] is referred to coeff (((1 1) (1 1)) NIL NIL (20)) = (21 21))

Coeff (((1 NIL) (NIL J) (NIL K)) NIL NIL (20 20)) = (400 (PLUS (TIMES 20 j) k)) where B[I,J,K] is 20 x 20 x 10 matrix

M-exp

coeff[s, i, c, ℓ] = If null [cdr[s]] then list [simpsum [caar[s]; i]; simpsum [cadar [s];c]] else coeff [cdr[s]; cons[simpmul[caar[s],ℓ], i]; cons [simpmul [cadar [s], ℓ]; c]; cdr [ℓ]]

SIMPSUM (A I):

We wish to add the terms in A and I to give a result of the form 'number + (sumbolic exp)'. However A can have different list structures requiring different action. Eg. '3' '(-J)' '(x 5 J)' '(+ 3 K)' '((X 3 K)J)'

(The last term is equivalent to '(+ (x 3 K) J)')

Example

Simpsum ((+ J 3) (20 K 4)) = Simplus ((J 3 20 K 4) NIL 0)

= (Plus 27 J K)

M-exp

Simpsum [a,i] = If atom [a]∨ member [car[a]; (TIMES MINUS)] then simplus [cons [a,i], NIL, 0] else if eq [car[a]; PLUS] then simplus [append [cdr[a],i]; NIL; 0] else simplus [append [a;i], NIL, 0]

SIMPLUS [S,V, C]:

Same as simpmpy except we test if C = 0 instead of 1 and even if some elements are null we continue processing.

Examples:

Simplus ((20 5 17) NIL 0) = 42

Simplus ((20 (x J 4)4 -3 (-K)) NIL 0)

= (Plus 21 (x J 4) (-K))

M-exp

Simplus [s,v,c] = If null [s] then [if zerop[c] then [if null [v] then NIL else if null [cdr [v]] then car [v] else cons [PLUS; v]] else [if null [v] then c else cons [PLUS; cons [c,v]]]] else if numberp [car[s]] then simplus [cdr[s]; v; [car[s] + c]] else if null [car[s]] then simplus [cdr[s]; v; c] else if ¬ [atom[car[s]] ∧eq[caar[s]; PLUS] then simplus [append [cdar [s]; cdr [s]]; v; c] else simplus [cdr[s]; cons[car[s];v]; c]

SIMPMUL [A,I]:

We wish to multiply elements of A and I to give a single resulting product list simplified. Terms 'I' are either numbers or S-expressions that have already been simplified. However A can be an atom or

'(x 3 J)' or '(+ 5 K)' or '((x 5 J) K)' = '(+ (x 5 J) K)'. All the terms are multiplied together (using distributive law if necessary) to arrive at the simplified product.

Example

Simpmul (((x 5 J) K) (20 20)) = (Simpmul (x 5 J) (20 20)) + (Simpmul

K(20 20) = (+(x2000 J) (x 400 K))

M-exp

Simpmul [a,ℓ] = If atom [a] then simpmpy [cons [a,ℓ]; NIL; 1] else if eq [car[a]; TIMES] then simpmpy [append [cdr[a],ℓ]; NIL; 1] else if eq [car[a]; PLUS] then simpmul [cdr[a]; ℓ] else if eq[car[a], MINUS] then list [MINUS; simpmul [cadr[a];ℓ] else simplus [list [simpmul [car [a];ℓ]; simpmul [cadr [a]; ℓ]]; NIL; 0]

SIMPMPY [S, V, C]

Gives simplified form of result of multiplication of the elements in list S. V is used to store the variables (S-exp) in S and C is a number containing the successive multiplier as the routine scans S. At the end the following cases are handled.

1. No Constant
   - No Variable list
   - There is Variable list
     - 1 element
     - more then 1

2. There is a constant
   - No Variable list
   - There is a Variable list

Examples:

Simpmpy (5 J 20 20) = (x 2000 J)

Simpmpy (5 20 20) = 2000

M-exp

Simpmpy [s,v,c] = If null [s] then [if onep [c] then [if null [v] then

NIL elseif null[cdr[v] then car [v] else cons [TIMES, v]] else

[if null [v] then c else cons [TIMES; cons [c; v]]]] else if

numberp [car[s]] then simpmpy [cdr [s]; v; times [car[s],c]] else if

null [car[s]] then NIL else simpmpy [cdr[s]; cons [car[s];v];c]

PAIREQUAL [L]:

List L contains each array reference with its I-coefficient and constant coefficient. We use Pairequal to scan through L and group together all those array references which have the same I-coefficient.

Example:

Pairequal (((↓X I) 1 NIL) ((↓X (+ (x 20 I) 3)) 20 3)

((↓ A I J) 20 J) ((↓ X (+ I 3) 1 3)

((↓A (+ I 4) J) 20 (+ J 80)) ((↓A J I) 1 (x 20 J)))

results in

[((((↓ A J I) 1 (x 20 J)) ((↓X(+ I 3)) 1 3) ((↓ X I) 1 NIL));

(((↓A (+ I 4) J) 20(+ J 80)) ((↓A I J) 20 J)

((↓X (+ 20 I) 3)) 20 3))]

M-exp

Pairequal [ℓ] = If null [ℓ] then NIL else λ [[c]; cons [car[c];

pairequal [cadr [c]]]] [equals [cadar [ℓ]; cdr [ℓ]; list [car [ℓ] ;

NIL]]

EQUALS[A, B, X, Y]:

If the array reference (car b) has the same I-coefficient as the array references in X(given by a) then add it to list X else to list Y. Repeat for all the array references in list b.

Thus given any I-coefficient 'a' equals separates the array reference list into two lists, those that have 'a' as their I-coefficient and those that do not.

```
Equals [20;  (((↓ A I J) 20 J)
              ((↓X I)  1 NIL)
              ((↓X (+ I 3) 1 3)
      (((↓ X ( x 20 I)) 20 NIL)); NIL]
```

results in

```
      [(((↓A( + I 4) J) 20( + J 80))
        ((↓A I J) 20 J)
        ((↓X (x 20 I)) 20 NIL ));
       (((↓X ( + I 3) 1 3) ((↓X I) 1 NIL))]
```

<u>M-exp</u>

Equals [a; b; x; y] = If null [b] then list [x;y] <u>else if</u> same

[a; cadar [ℓ]] then equals [a; cdr [b]; cons [car [b]; x]; y]

<u>else</u> equals [a; cdr [b]; x; cons [car [ ]; y]]

SAME [a; b]:

Given any two arithmetic expressions Same checks to see if they are equal. However, the terms in expressions need not be in the same order. The operator must be the same and the operands must have a one- to-one correspondence between them (commutativity of arithmetic expressions is used. However, no effort is made to check for equality under distributivity).

<u>Example</u>:

Same [(+ (x 20 I) 30 (-J) (+ A 20)) (+ (-J) (x 20 I) (+ A 20) 30)]

= *T*

<u>M-exp</u>

Same [a,b] = If atom [a] <u>then</u> equal [a,b] <u>else if</u> atom [b] then

FALSE <u>else</u> eq [car [a]; car [b]]$\wedge$mapq [cdr [a]; cdr [b]]

MAPQ [A, B]:

Given lists of operands A and B, Mapq attemtps to find a one-to-one correspondence betweem them.It is true if there exists a 1-1 relation between A and B else false.

Examples

Mapq [(30 (+ A 20)) ((+ A 20)30)] = T

Mapq [(30 (+ A 20)) ((x A 20) (+ A 20) 30)] = F

M-exp

Mapq [a,b] = If null [a] $\wedge$ null [b] then TRUE else if null [a] $\vee$ null [b] then FALSE else mapq [cdr [a]; mapa [car [a]; b; NIL]]

MAPA [A, B, X]:

Given an expression 'A' Mapa scans B to see if there exists an equivalent expression in B. If so, the result is a list of all the elements of B except the one equivalent to A. If there is no element equal to A then the result is 'NIL'.

Examples

Mapa (20 (J 20 K 40) NIL) = (J K 40)

Mapa (20 (J K 40)) = NIL

M-exp

mapa [a,b,x] = If null [b] then NIL else if same [a, car [b]] then repend [x; cdr [b]] else mapa [a; cdr [b]; cons [car [b]; x]]

REPEND [A, B]:

Adds the elements of list 'A" to list 'B'. Similar to append but the elements are added in the reverse order.

Example

Repend [(J (+ 20 K) L); (30 (x 3 J))]
= (L (+ 20 K) J 30 (x 3 J))

M-exp

Repend [a,b] =

If null [a] then b else repend [cdr [a], cons [car [a], b]]

GENERATE [FS, FC, L]

Generate is a major routine to the program. Its main purpose is to create a modified program of the innermost 'FOR' loop (generating a 'FOR CLAUSE' which can be used direclty in the Index register assignment.) Given the old FOR Statement FS, For Clause FC and List of Pairs L it generates a modified For Statement. L is a list of all the subscripted variables referenced, grouped so that all the subscripts that have the same coefficient of I (control variable of the For Statement) are together. Each element in L is a dotted pair of Array Reference. Equivalent Single Subscript (which in turn is a dotted pair of I-coefficient and constant coefficient).

M-exp

Generate [fs; fc; ℓ] =

λ[[b]; conc [genf [cadr [b]]; list [conc [gene [cddr [b]; fc; car

[fc]]; list [sublis [car [b]; fs]]]]]

[λ[[a]; conc [genb [car [a] ; NIL; NIL]; cdr [a]]]

[gena [ℓ; NIL; NIL; NIL]]]

GENB [L N IL]

Given a list of all the array references, GENB scans through their constant coefficients and replaces where desirable (when there exists an S-exp which can be evaluated outside the FOR loop) by an atomic symbol in N and the corresponding atomic symbols and S-exp are dotted together to form list IL (Index list).

Example:

```
Genb  [((((↓A I ( x 20 J)) G00012 ( x 20 J))
         ((↓A I J) G00012 J)
         ((↓ X ( + ( x 20 I) 3)) G00012 3)
         ((↓ A J I) G00011 ( x 20 J)
         ((↓ X ( + I 3) G00011 3)
         ((↓ X I) G00011 NIL)); Nil; Nil]
```

results in

```
[(((↓X I) . (↓ X G00011))
  (↓ X ( + I 3)) . (↓ X ( + G00011 3))
  (↓ A J I). (↓ A ( + G00011 G00013))
  ( ↓X ( + ( x 20 I) 3)). (↓X  ( + G00012 3))
  (↓ A I J) . (↓A  ( + G00012  J))
  (↓ A I ( x 20 J)).  (↓ A  (+ G00012 G00013))),
  (( x 20 J). G00013)]
```

<u>M-exp</u>

Genb [ℓ, n, iℓ] =

If null [ℓ] then list [n; iℓ] else λ[[c]; genb [cdr [ℓ]; cons [gend [car [ℓ]; car [c]]; n]; cadr [c]]] [genc [caddar [ℓ]; iℓ]]

Sassoc [x, y, u]:

Same as the one in the Lisp 1.5 Manual except it uses equal instead of eq for greater flexibility.

<u>M-exp</u>

sassoc [x, y, u] = <u>If</u> null [y] <u>then</u> u <u>else if</u> equal [caar [y]; n] <u>then</u> cdar [y] <u>else</u> sassoc [n, cdr [y], u]

GENC [L, IL]:

Given the constant coefficient L of the array element being processed GENC determines whether there exists an equivalent expression for which a symbol has been created or whether it is necessary to create a new symbol (of course if the coefficient is atom there is no need for any more simplication). Result is (symbol. Index list). Symbol may be the generated symbol or same as L (if Number or NIL).

<u>Examples</u>:

```
genc [( x 20 J) ((( x 20 J) . G00013))] = (G00013,(((x 20 J). G00013)))
genc[ 3 ((( x 20 J). G00013))] = ( 3 ((( x 30 J). G00013)))
genc [ ( + J 3) ((( x 20 J). G00013))] =
 (G00014 ((( x 20 J). G00013) (( + J 3). G00014)))
```

M-exp

genc [ℓ; iℓ] =

If atom [ℓ] then list [ℓ; iℓ] else

λ[[a]; if null [a] then λ[[b]; list [b; cons [cons [ℓ; b]; iℓ]]

[gensym] else list [a, iℓ]]

[sassoc [ℓ; iℓ; NIL]]

GEND [L, N]

Given L, the (array reference. I-coefficient . Constant coefficient) this juction forms a new array reference dotted with the old one all ready for substitution into the program.

Example:

Gend [((↓ A I (x 20 J)) G00012 (x 20 J)); G00013)

= ((↓ A I (x 20 J)). (↓ A (+ G00012 G00013)))

M-exp

Gend [ℓ, n] =

If null [n] then cons [car [ℓ]; list [caar [ℓ]; cadar [ℓ]; cadr [ℓ]]]

else if null [cadr [ℓ]] then cons [car [ℓ]; list [caar [ℓ]; cadar [ℓ], n]]

else cons [car [ℓ]; list [caar[ℓ]; cadar [ℓ]; list [PLUS; cadr [ℓ]; n]]]

GENA [L, N, FL, FR]:

Given a list L of array references grouped so that all those with the same I-coefficient are together GENA recursively replaces equivalent I-coefficients (Via FORML) by a single atomic symbol in List N and the new symbols created and their equivalent S-expressions are formed into lists of dotted pairs FL. Result is (N FL FR) N is a single list (() for equal groups are now removed). FR gives the frequency of occurrence of each I-coefficient which may be used in the optimal assignment of storage.

Example:

Gena [(((↓ X I) 1 NIL)

((↓ X (+ I 3)) 1 3) } I-coefficient = 1

((↓ A J I) 1 (x 20 J)))

```
        ((↓x ( + ( x 20 I) 3)) 20 3)
        ((↓A I J) 20 J)
        ((↓A ( + I 4) J) 20 (+ J 80)))); NIL; NIL]
results in
((((↓A( + I 4) J) G00012 ( + J 80))
  ((↓A I J G00012 JO
  ((↓ X 20 I) 3)) G0012 3)
  ((↓A J I G00011 ( x 20 JOO
  ((↓ X ( + I 3)) G00011 3)
  ((↓ X I) G00011 NIL)) ((G00012.20) (G00011.1))
   (FREQUENCY (G00012 3) (G00011 3)))
```

<u>M-exp</u>

gena [ℓ; n; fℓ; fr] =

If null [ℓ] <u>then</u> list [n, fℓ; cons [FREQUENCY; fr]] <u>else if</u> null [cadaar [ℓ]] <u>then</u> gena [cdr [ℓ]; append [car[ℓ]; n]; fℓ; fr] <u>else</u> λ [[c]; gena [cdr [ℓ], formℓ [car [ℓ]; c; n]; cons [cons [c; cadar [ℓ]]; fℓ]; cons [list [c; count [car [ℓ]]; fr]] [gensym]

COUNT [L]:

Counts the number of elements (or S-expressions) in the list L

<u>Example</u>

Count (A B (DARROW X Y I)) = 3

<u>M-exp</u>

Count [ℓ] =

If null [ℓ] then 0 else 1 + count [cdr [ℓ]]

FORML [L, C, N]

Given a list L of array references whose I-coefficients are the same Formℓ replaces the I-coefficients by the atomic symbol given by C and forms the new list in N

```
Formℓ [(((↓X I) 1 NIL)

        ((↓X ( + I 3)) 1 3)

        ((↓A J I) 1 ( x 20 J))); G00011; NIL]

gives [(((↓X I) G00011 NIL)

        ((↓X ( + I 3)) G00011 3)

        ((↓A J I G00011 ( x 20 J))]
```

<u>M-exp</u>

Formℓ [ℓ; c; n] =

<u>If</u> null [ℓ] <u>then</u> n <u>else</u>

formℓ [cdr [ℓ]; c; cons [list [caar [ℓ]; c; caddar [ℓ]]; n]]

GENE [L, FC, I]:

Given a list L of all the functions of I, the control variable that would be desirable to have in index registers, gene forms an extend For-clause of the original For-clause FC and the control variable L is a list of dotted pairs.
<u>Example</u>:

```
Gene [(((G0001.60)(G0002.J)(G0003 Plus  J 20)) (FREQUENCY (G0001 4)
       (G0002 1)(G0003 5))) ( I step 1, 1, 20) I]
  → (FOR ((I step 1  1  20)

           ( ←G0001 ( x 60 I))

           ( ←G0002 ( x J I))

           ( ←G0003 ( x ( + J 20) I )))
           (FREQUENCY (G0001 4)(G0002 1)(G0003 5)))
```

<u>M-exp</u>

gene [ℓ, fc, i] =

list [FOR; conc [fc; maplist [car [ℓ] ; λ[[a]; list [ASSIGN;

caar [a]; list [TIMES; cdar [a]; i]]]; cdr [ℓ]]

GENF [L]:

Given a list L containing all the constant-coefficient expressions required in the evaluation of subscripts that could be more effectively performed outside (because (1) they donot change within the FOR loop (2) they are evaluated more than once within the For-loop), GENF generates a number of assignment statements assigning the value of the expression to a given symbol.

Example:

genf (((G0010 TIMES J K) (G0012 PLUS (TIMES 3 J) 5)))

→ ((← G0010 (x J K))

(← G0012 (+ (x 3 J) 5)))

M-exp

genf [ℓ] =

Maplist [ℓ; λ[[a]; list [ASSIGN; cdar [a]; caar [a]]]

```
 A018315LISP        1MIN    400RAJ REDDY CS 239
EXECUTION. LIBRARY PROGRAM---LISP

            TEST

EVALQUOTE OPERATOR AS OF 1 MARCH 1961.    INPUT LISTS NOW BEING READ.



 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
DEFINE
(((ANALYSE (LAMBDA (P AD) (COND ((NULL P) NIL) ((EQ (CAAR P) (QUOTE DECLARE)) (CONS (CAR P) (ANALYSE (CDR P)
(DEC (CDAR P) AD)))) ((OR (EQ (CAAR P) (QUOTE IFT)) (EQ (CAAR P) (QUOTE IFD))) (CONS (CONS (CAAR P) (ANALYSE
(CDAR P) AD)) (ANALYSE (CDR P) AD))) ((EQ (CAAR P) (QUOTE FOR)) (COND ((INNERLOOP (CADDAR P)) (APPEND (FORST
(CDAR P) AD) (ANALYSE (CDR P) AD))) (T (CONS (CONS (CAAR P) (CONS (CADAR P) (ANALYSE (CDDAR P) AD))) (ANALYSE
(CDR P) AD))))) ((NOT (ATOM (CAAR P))) (CONS (ANALYSE (CAR P) AD) (ANALYSE (CDR P) AD))) (T (CONS (CAR P) (
ANALYSE (CDR P) AD)))))) (INNERLOOP (LAMBDA (P) (COND ((NULL P) T) ((NOT (ATOM (CAR P))) (AND (INNERLOOP (CAR
P)) (INNERLOOP (CDR P)))) ((EQ (CAR P) (QUOTE FOR)) F) ((OR (EQ (CAR P) (QUOTE IFT)) (EQ (CAR P) (QUOTE IFD)))
(INNERLOOP (CDDR P))) (T T)))) (DEC (LAMBDA (L AD) (COND ((NULL L) AD) ((EQ (CADAR L) (QUOTE ARRAY)) (DEC (
CDR L) (NCONC (CHANGE (CDDAR L)) AD))) (T (DEC (CDR L) AD))))) (CHANGE (LAMBDA (P) (MAPLIST P (FUNCTION (LAMBDA
(C) (CONS (CAAR C) (MAPLIS (CDAR C) (FUNCTION (LAMBDA (D) (COND ((ONEP (CAAR D)) (CADAR D)) ((NUMBERP (CAAR
D)) (COND ((NUMBERP (CADAR D)) (DIFFERENCE (CADAR D) (SUB1 (CAAR D)))) (T (LIST (QUOTE DIFFERENCE) (CADAR D)
(SUB1 (CAAR D)))))) ((NUMBERP (CADAR D)) (LIST (QUOTE DIFFERENCE) (ADD1 (CADAR D)) (CAAR D))) (T (LIST (QUOTE
DIFFERENCE) (CADAR D) (LIST (QUOTE DIFFERENCE) (CAAR D) 1))))))))))))) (MAPLIS (LAMBDA (X FN) (COND ((NULL
(CDR X)) NIL) (T (CONS (FN X) (MAPLIS (CDR X) FN))))))))


END OF EVALQUOTE, VALUE IS ..
(ANALYSE INNERLOOP DEC CHANGE MAPLIS)



 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
DEFINE
(((FORST (LAMBDA (L AD) ((LAMBDA (C) (GENERATE (CADR L) (CAR L) ((LAMBDA (A) (PAIREQUAL (PAIR A (TRANSFORM
(CAAR L) A C)))) (TEAR (CADR L) (MAPLIST C (FUNCTION (LAMBDA (J) (CAAR J)))))))) (COND ((EQ (QUOTE DECLARE)
(CAAADR L)) (DEC (CAADR L) AD)) (T AD))))) (TEAR (LAMBDA (P A) ((LAMBDA (C) (OK (REGULAR (CAR C) (CADR C)) NIL
A)) (TEARIT P 0)))) (OK (LAMBDA (L N A) (COND ((NULL L) N) ((MEMBER (CADAR L) A) (OK (CDR L) (CONS (CAR L)
N) A)) (T (OK (CDR L) N A))))) (TEARIT (LAMBDA (P NO) (COND ((NULL P) (LIST NIL NIL)) (T (JOIN (FORMLIST (CAR
P) NIL NIL NO) (TEARIT (CDR P) (ADD1 NO))))))) (FORMLIST (LAMBDA (P L A NO) (COND ((ATOM P) (LIST L A)) ((NOT
(ATOM (CAR P))) (JOIN (FORMLIST (CAR P) NIL NIL NO) (FORMLIST (CDR P) L A NO))) ((EQ (CAR P) (QUOTE ASSIGN))
(FORMLIST (CDR P) (CONS (CADR P) L) A NO)) ((EQ (CAR P) (QUOTE DARROW)) (LIST L (CONS (CONS P NO) A))) ((EQ
(CAR P) (QUOTE DECLARE)) (LIST L A)) (T (FORMLIST (CDR P) L A NO))))) (JOIN (LAMBDA (L M) (COND ((NULL L) NIL)
(T (CONS (APPEND (CAR L) (CAR M)) (JOIN (CDR L) (CDR M))))))) (REGULAR (LAMBDA (L A) (COND ((NULL A) NIL) (
T (COND ((REG (CDDAAR A) L) (CONS (CAAR A) (REGULAR L (CDR A)))) (T (REGULAR L (CDR A)))))))) (REG (LAMBDA
(A L) (OR (NULL A) (AND (REG1 (CAR A) L) (REG (CDR A) L))))) (REG1 (LAMBDA (E L) (COND ((ATOM E) (NOT (MEMBER
E L))) ((MEMBER (CAR E) (QUOTE (DIFFERENCE PLUS))) (AND (REG2 (CADR E) L) (REG2 (CADDR E) L))) ((MEMBER (CAR
E) (QUOTE (TIMES MINUS))) (REG2 E L)) (T F)))) (REG2 (LAMBDA (E L) (COND ((ATOM E) (NOT (MEMBER E L))) ((EQ
(CAR E) (QUOTE MINUS)) (REG2 (CADR E) L)) ((EQ (CAR E) (QUOTE TIMES)) (REG3 (CDR E) L)) (T (AND (REG2 (CAR
E) L) (REG2 (CDR E) L)))))) (REG3 (LAMBDA (E L) (COND ((NULL E) T) (T (AND (ATOM (CAR E)) (NOT (MEMBER (CDR
E) L)) (REG3 (CDR E) L))))))))
```

28

```
END OF EVALQUOTE, VALUE IS ..
(FORST TEAR OK TEARIT FORMLIST JOIN REGULAR REG REG1 REG2 REG3)



 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
DEFINE
(((TRANSFORM (LAMBDA (I L A) (COND ((NULL L) NIL) (T (CONS (COEFF (SPLIT I (CDDAR L)) NIL NIL (SASSOC (CADAR
L) A NIL)) (TRANSFORM I (CDR L) A)))))) (SPLIT (LAMBDA (I L) (COND ((NULL L) NIL) (T (CONS (SPLITA I (CAR L))
(SPLIT I (CDR L))))))) (SPLITA (LAMBDA (I E) (COND ((ATOM E) (COND ((EQ I E) (LIST (QUOTE 1) NIL)) (T (LIST NIL
E)))) ((EQ (CAR E) (QUOTE TIMES)) (COND ((MEMBER I (CDR E)) (LIST (SIMPMPY (EFFACE I (CDR E)) NIL 1) NIL))
(T (LIST NIL (SIMPMPY (CDR E) NIL 1))))) ((EQ (CAR E) (QUOTE DIFFERENCE)) (JOINA (SPLITA I (CADR E)) (SPLITA
I (LIST (QUOTE MINUS) (CADDR E))))) ((EQ (CAR E) (QUOTE MINUS)) ((LAMBDA (C) (MAPLIST C (FUNCTION (LAMBDA (
A) (COND ((NULL (CAR A)) NIL) ((NUMBERP (CAR A)) (MINUS (CAR A))) (T (LIST (QUOTE MINUS) (CAR A)))))))) (SPLITA
I (CADR E)))) (T (JOINA (SPLITA I (CADR E)) (SPLITA I (CADDR E))))))) (JOINA (LAMBDA (L M) (COND ((NULL L) NIL)
(T (CONS (COND ((NULL (CAR L)) (CAR M)) ((NULL (CAR M)) (CAR L)) (T (LIST (CAR L) (CAR M)))) (JOINA (CDR L)
(CDR M))))))) (COEFF (LAMBDA (S I C L) (COND ((NULL (CDR S)) (LIST (SIMPSUM (CAAR S) I) (SIMPSUM (CADAR S)
C))) (T (COEFF (CDR S) (CONS (SIMPMUL (CAAR S) L) I) (CONS (SIMPMUL (CADAR S) L) C) (CDR L)))))) (SIMPSUM (
LAMBDA (A I) (COND ((OR (ATOM A) (MEMBER (CAR A) (QUOTE (TIMES MINUS)))) (SIMPLUS (CONS A I) NIL 0)) ((EQ (
CAR A) (QUOTE PLUS)) (SIMPLUS (APPEND (CDR A) I) NIL 0)) (T (SIMPLUS (APPEND A I) NIL 0))))) (SIMPMUL (LAMBDA
(A L) (COND ((ATOM A) (SIMPMPY (CONS A L) NIL 1)) ((EQ (CAR A) (QUOTE TIMES)) (SIMPMPY (APPEND (CDR A) L) NIL
1)) ((EQ (CAR A) (QUOTE PLUS)) (SIMPMUL (CDR A) L)) ((EQ (CAR A) (QUOTE MINUS)) (LIST (QUOTE MINUS) (SIMPMUL
(CADR A) L))) (T (SIMPLUS (LIST (SIMPMUL (CAR A) L) (SIMPMUL (CADR A) L)) NIL 0))))) (SIMPLUS (LAMBDA (S V
C) (COND ((NULL S) (COND ((ZEROP C) (COND ((NULL V) NIL) ((NULL (CDR V)) (CAR V)) (T (CONS (QUOTE PLUS) V))))
(T (COND ((NULL V) C) (T (CONS (QUOTE PLUS) (CONS C V))))))) ((NUMBERP (CAR S)) (SIMPLUS (CDR S) V (PLUS (CAR
S) C))) ((NULL (CAR S)) (SIMPLUS (CDR S) V C)) ((AND (NOT (ATOM (CAR S))) (EQ (CAAR S) (QUOTE PLUS))) (SIMPLUS
(APPEND (CDAR S) (CDR S)) V C)) (T (SIMPLUS (CDR S) (CONS (CAR S) V) C))))) (SIMPMPY (LAMBDA (S V C) (COND
((NULL S) (COND ((ONEP C) (COND ((NULL V) NIL) ((NULL (CDR V)) (CAR V)) (T (CONS (QUOTE TIMES) V)))) (T (COND
((NULL V) C) (T (CONS (QUOTE TIMES) (CONS C V))))))) ((NUMBERP (CAR S)) (SIMPMPY (CDR S) V (TIMES (CAR S) C)))
((NULL (CAR S)) NIL) (T (SIMPMPY (CDR S) (CONS (CAR S) V) C)))))))


END OF EVALQUOTE, VALUE IS ..
(TRANSFORM SPLIT SPLITA JOINA COEFF SIMPSUM SIMPMUL SIMPLUS SIMPMPY)



 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
DEFINE
(((PAIREQUAL (LAMBDA (L) (COND ((NULL L) NIL) (T ((LAMBDA (C) (CONS (CAR C) (PAIREQUAL (CADR C)))) (EQUALS
(CADAR L) (CDR L) (LIST (CAR L)) NIL)))))) (EQUALS (LAMBDA (A B X Y) (COND ((NULL B) (LIST X Y)) ((SAME A (
CADAR B)) (EQUALS A (CDR B) (CONS (CAR B) X) Y)) (T (EQUALS A (CDR B) X (CONS (CAR B) Y)))))) (SAME (LAMBDA
(A B) (COND ((ATOM A) (EQUAL A B)) ((ATOM B) F) (T (AND (EQ (CAR A) (CAR B)) (MAPQ (CDR A) (CDR B))))))) (MAPQ
(LAMBDA (A B) (COND ((AND (NULL A) (NULL B)) T) ((OR (NULL A) (NULL B)) F) (T (MAPQ (CDR A) (MAPA (CAR A) B NIL))))))
(MAPA (LAMBDA (A B X) (COND ((NULL B) NIL) ((SAME A (CAR B)) (REPEND X (CDR B))) (T (MAPA A (CDR B) (CONS (
CAR B) X)))))) (REPEND (LAMBDA (A B) (COND ((NULL A) B) (T (REPEND (CDR A) (CONS (CAR A) B))))))))

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.                 FULL WORDS  1408 FREE   2910  PUSH DOWN DEPTH    115

END OF EVALQUOTE, VALUE IS ..
(PAIREQUAL EQUALS SAME MAPQ MAPA REPEND)
```

```
 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
DEFINE
(((GENERATE (LAMBDA (FS FC L) ((LAMBDA (B) (CONC (GENF (CADR B)) (LIST (CONC (GENE (CDDR B) FC (CAR FC)) (LIST
(SUBLIS (CAR B) FS)))))) ((LAMBDA (A) (CONC (GENB (CAR A) NIL NIL) (CDR A))) (GENA L NIL NIL NIL))))) (GENB
(LAMBDA (L N IL) (COND ((NULL L) (LIST N IL)) (T ((LAMBDA (C) (GENB (CDR L) (CONS (GEND (CAR L) (CAR C)) N)
(CADR C))) (GENC (CADDAR L) IL))))))) (GENC (LAMBDA (L IL) (COND ((ATOM L) (LIST L IL)) (T ((LAMBDA (A) (COND
((NULL A) ((LAMBDA (B) (LIST B (CONS (CONS L B) IL))) (GENSYM))) (T (LIST A IL)))) (SASSOC L IL NIL)))))) (
GEND (LAMBDA (L N) (COND ((NULL N) (CONS (CAR L) (LIST (CAAR L) (CADAR L) (CADR L)))) ((NULL (CADR L)) (CONS
(CAR L) (LIST (CAAR L) (CADAR L) N))) (T (CONS (CAR L) (LIST (CAAR L) (CADAR L) (LIST (QUOTE PLUS) (CADR L)
N))))))) (SASSOC (LAMBDA (X Y U) (COND ((NULL Y) NIL) ((EQUAL (CAAR Y) X) (CDAR Y)) (T (SASSOC X (CDR Y) U)))))
(GENA (LAMBDA (L N FL FR) (COND ((NULL L) (LIST N FL (CONS (QUOTE FREQUENCY) FR))) ((NULL (CADAAR L)) (GENA
(CDR L) (APPEND (CAR L) N) FL FR)) (T ((LAMBDA (C) (GENA (CDR L) (FORML (CAR L) C N) (CONS (CONS C (CADAAR
L)) FL) (CONS (LIST C (COUNT (CAR L))) FR))) (GENSYM)))))) (COUNT (LAMBDA (L) (COND ((NULL L) 0) (T (ADD1 (
COUNT (CDR L))))))) (FORML (LAMBDA (L C N) (COND ((NULL L) N) (T (FORML (CDR L) C (CONS (LIST (CAAR L) C (CADDAR
L)) N)))))) (GENE (LAMBDA (L FC I) (LIST (QUOTE FOR) (CONC FC (MAPLIST (CAR L) (FUNCTION (LAMBDA (A) (LIST
(QUOTE ASSIGN) (CAAR A) (LIST (QUOTE TIMES) (CDAR A) I)))))) (CDR L))))) (GENF (LAMBDA (L) (MAPLIST L (FUNCTION
(LAMBDA (A) (LIST (QUOTE ASSIGN) (CDAR A) (CAAR A)))))))))

END OF EVALQUOTE, VALUE IS ..
(GENERATE GENB GENC GEND SASSOC GENA COUNT FORML GENE GENF)
```



```
 FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
ANALYSE
(((DECLARE (INTEGER J K I) (REAL SUM) (REAL ARRAY (A (1 20) (1 20)) (B (1 20) (1 40)) (C (1 40) (1 20)))) (
FOR (I (STEP 1 1 20)) (FOR (J (STEP 1 1 20)) ((ASSIGN (DARROW A I J) 0) (FOR (K (STEP 1 1 40)) ((ASSIGN (DARROW
A I J) (PLUS (DARROW A I J) (TIMES (DARROW B I K) (DARROW C K J)))) (ASSIGN P (PLUS I 3)) (IFT (EQUAL (DARROW
A I J) 0) (ASSIGN (DARROW A I J) 1) (ASSIGN (DARROW A I J) 0))))))) (WRITE SUM C)) NIL)

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.                     FULL WORDS  1406 FREE   2851  PUSH DOWN DEPTH    266

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.                     FULL WORDS  1400 FREE   2785  PUSH DOWN DEPTH    183

END OF EVALQUOTE, VALUE IS ..
((DECLARE (INTEGER J K I) (REAL SUM) (REAL ARRAY (A (1 20) (1 20)) (B (1 20) (1 40)) (C (1 40) (1 20)))) (FOR
(I (STEP 1 1 20)) (FOR (J (STEP 1 1 20)) ((ASSIGN (DARROW A I J) 0) (ASSIGN G00013 (PLUS (TIMES 20 I) J)) (
ASSIGN G00012 (TIMES 20 I)) (FOR (K (STEP 1 1 40) (ASSIGN G00011 (TIMES 40 K)) (ASSIGN G00010 (TIMES 1 K))
(FREQUENCY (G00011 1) (G00010 1))) ((ASSIGN (DARROW A G00013) (PLUS (DARROW A G00013) (TIMES (DARROW B (PLUS
G00010 G00012)) (DARROW C (PLUS G00011 J))))) (ASSIGN P (PLUS I 3)) (IFT (EQUAL (DARROW A G00013) 0) (ASSIGN
(DARROW A G00013) 1) (ASSIGN (DARROW A G00013) 0))))))) (WRITE SUM C))
```

```
FUNCTION   EVALQUOTE   HAS BEEN ENTERED, ARGUMENTS..
ANALYSE
(((DECLARE (REAL ARRAY (R (1 6) (1 6)) (A (1 6) (1 6)) (C (1 12) (1 N))) (INTEGER ARRAY (X (1 10) (1 10) (1
N) (1 15)) (Y (1 10) (0 9) (1 N))) (INTEGER I J K L) (REAL TEMP)) (FOR (I (STEP 1 1 3)) (FOR (J (STEP 1 1 I))
((ASSIGN K (PLUS (ASSIGN L (TIMES 3 I)) J)) (ASSIGN (DARROW R I J) (DARROW C K (PLUS (TIMES 3 I) J))) (ASSIGN
(DARROW R (PLUS I 3) (PLUS J 3)) (DARROW R I J)) (ASSIGN (DARROW R (PLUS I 3) J) 0)))) (FOR (I (STEP 1 1 3))
(FOR (J (STEP 1 1 3)) ((ASSIGN (DARROW A (PLUS I 3) J) (DARROW R (PLUS J 3) I)) (ASSIGN (DARROW A (PLUS J 3)
(PLUS I 3)) (DARROW R I J))))) (FOR (I (STEP 1 1 6)) (FOR (J (STEP 1 1 6)) ((ASSIGN (DARROW A I J) (DARROW
R J I)) (ASSIGN (DARROW A I J) (DARROW R (DIFFERENCE 7 I) (DIFFERENCE 7 J))) (ASSIGN (DARROW A I J) (DARROW
R (DIFFERENCE 7 I) J)) (ASSIGN (DARROW A I J) (DARROW R I (DIFFERENCE 7 J)))))) (FOR (I (STEP 1 1 10)) ((ASSIGN
(DARROW X I J K 1) (DARROW Y I J K)) (ASSIGN (DARROW X (PLUS I 3) (TIMES I J) (PLUS K 1) I) (TIMES (DARROW
Y I (PLUS (TIMES 10 I) J) (PLUS K 1)) (DARROW X (DIFFERENCE (TIMES I 3) 4) (MINUS I) K 1))))))) NIL)

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1409 FREE   2899  PUSH DOWN DEPTH    195

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1400 FREE   2883  PUSH DOWN DEPTH    164

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1400 FREE   2863  PUSH DOWN DEPTH    209

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1388 FREE   2740  PUSH DOWN DEPTH    202

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1381 FREE   2684  PUSH DOWN DEPTH    216

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1371 FREE   2652  PUSH DOWN DEPTH    177

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1368 FREE   2549  PUSH DOWN DEPTH    237

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1364 FREE   2516  PUSH DOWN DEPTH    246

GARBAGE COLLECTOR ENTERED AT  04316 OCTAL.          FULL WORDS  1354 FREE   2429  PUSH DOWN DEPTH    208

END OF EVALQUOTE, VALUE IS ..
((DECLARE (REAL ARRAY (R (1 6) (1 6)) (A (1 6) (1 6)) (C (1 12) (1 N))) (INTEGER ARRAY (X (1 10) (1 10) (1
N) (1 15)) (Y (1 10) (0 9) (1 N))) (INTEGER I J K L) (REAL TEMP)) (FOR (I (STEP 1 1 3)) (ASSIGN G00017 (PLUS
21 (TIMES 6 I))) (ASSIGN G00016 (TIMES 6 I)) (ASSIGN G00015 (PLUS 18 (TIMES 6 I))) (FOR (J (STEP 1 1 I) (ASSIGN
G00014 (TIMES 1 J)) (FREQUENCY (G00014 4))) ((ASSIGN K (PLUS (ASSIGN L (TIMES 3 I)) J)) (ASSIGN (DARROW R (
PLUS G00014 G00016)) (DARROW C K (PLUS (TIMES 3 I) J))) (ASSIGN (DARROW R (PLUS G00014 G00017)) (DARROW R (
PLUS G00014 G00016))) (ASSIGN (DARROW R (PLUS G00014 G00015)) 0)))) (FOR (I (STEP 1 1 3)) (ASSIGN G00023 (PLUS
18 (TIMES 6 I))) (ASSIGN G00022 (TIMES 6 I)) (ASSIGN G00021 (PLUS 21 I)) (ASSIGN G00020 (PLUS 18 I)) (FOR (
J (STEP 1 1 3) (ASSIGN G00019 (TIMES 6 J)) (ASSIGN G00018 (TIMES 1 J)) (FREQUENCY (G00019 2) (G00018 2))) (
(ASSIGN (DARROW A (PLUS G00018 G00023)) (DARROW R (PLUS G00019 G00020))) (ASSIGN (DARROW A (PLUS G00019 G00021))
(DARROW R (PLUS G00018 G00022)))))) (FOR (I (STEP 1 1 6)) (ASSIGN G00030 (PLUS 49 (MINUS (TIMES 6 I)))) (ASSIGN
G00029 (PLUS 7 (TIMES 6 I))) (ASSIGN G00028 (PLUS 42 (MINUS (TIMES 6 I)))) (ASSIGN G00027 (TIMES 6 I)) (FOR
(J (STEP 1 1 6) (ASSIGN G00026 (TIMES 6 J)) (ASSIGN G00025 (TIMES 1 J)) (ASSIGN G00024 (TIMES -1 J)) (FREQUENCY
(G00026 1) (G00025 5) (G00024 2))) ((ASSIGN (DARROW A (PLUS G00025 G00027)) (DARROW R (PLUS G00026 I))) (ASSIGN
(DARROW A (PLUS G00025 G00027)) (DARROW R (PLUS G00024 G00030))) (ASSIGN (DARROW A (PLUS G00025 G00027)) (DARROW
R (PLUS G00025 G00028))) (ASSIGN (DARROW A (PLUS G00025 G00027)) (DARROW R (PLUS G00024 G00029)))))) (ASSIGN
G00040 (PLUS 1 (TIMES -400 N) (TIMES N K))) (ASSIGN G00039 (PLUS 1 (TIMES 10 N J) (TIMES N K))) (ASSIGN G00038
(PLUS 1 (TIMES 10 J) K)) (ASSIGN G00037 (PLUS (TIMES 10 J) K)) (ASSIGN G00036 (PLUS (TIMES 300 N) (TIMES N
K) N)) (FOR (I (STEP 1 1 10) (ASSIGN G00035 (TIMES (PLUS 1 (TIMES 100 N) (TIMES 10 N J)) I)) (ASSIGN G00034
(TIMES 100 I)) (ASSIGN G00033 (TIMES 200 I)) (ASSIGN G00032 (TIMES (TIMES 100 N) I)) (ASSIGN G00031 (TIMES
(PLUS (TIMES 300 N) (TIMES -10 N)) I)) (FREQUENCY (G00035 1) (G00034 1) (G00033 1) (G00032 1) (G00031 1)))
((ASSIGN (DARROW X (PLUS G00032 G00039)) (DARROW Y (PLUS G00034 G00037))) (ASSIGN (DARROW X (PLUS G00035 G00036))
(TIMES (DARROW Y (PLUS G00033 G00038)) (DARROW X (PLUS G00031 G00040)))))))
```

TEST RESULTS:

The following two program were run to illustrate the use of the program. The input and output are given below in M-expression form.

Matrix multiplication routine:

```
Anaylze [

  ((Declare (Interger J K I)

            (Real Sum)

            (Real Array (A ( 1 20) ( 1 20))(B ( 1 20) (1 40))

                 (C (1 40) ( 1 20))))

  (For (I (Step 1 1 20))

     (For (J (Step 1 1  20))

        (( ← (↓A I J) 0)

          (For (K (Step 1 1 40))

             ( ← (↓A I J ) ( + (↓A I J) ( ⊗ (↓B I K) (↓C K J))))

          )
        )
     )
  )) ; NIL]  results in

  (( Declare (Integer J K I)

             (Real Sum)

             (Real Array (A (1 20) ( 1 20)) (B(1 20)(1 40))(C(1 40)
                   (1 20))))

    (For (I (Step 1 1 20))

       (For (J (Step 1 1 20))

          (( ←( ↓A I J ) 0)

             ( ←G00012 ( + ( x 20 I)J))

             (For (K (step 1 1 40) ( ←  G00011 ( ⊗ 40 K)) ( ←G0010 (⊗ 1 K))
                     (Frequency (G00011 1) (G0010 1)))
```

```
( ←( ↓A G00012)( + (↓A G00012 ) ( ⊗ (↓B(+ G00010 G0012))
   (↓C (+ G00011 J))  )))  )))))
```

Example 2:

```
Analyze [
(( Declare (Real Array (R (1 6) (1 6)) (A (1 6) (1 6)) (C ( 1 12)(1 N)))
        (Integer Array ( X ( 1 10) ( 1 10)(1 N) (1 15)) (Y(1 10)(0 9)
                    (1 N)))
        (Integer I J K L)
        ( Real Temp))
 (For (I (step 1 1 3))
    (For (J (Step 1 1 I))
     (( ←K ( + ( ←L ( ⊗3 I)) 3))
      ( ←( ↓R I J) (↓C K L))
      ( ←(↓R (+ I 3) ( + J 3)) (↓R I J))
      ( ←(↓R (+ I 3) J) 0)
    )))
 (For (I (Step 1 1 3))
    (For (J (Step 1 1 3))
    (( ←(↓A (+ I 3) J) (↓R ( + J 3) I ))
     ( ←(↓A (+ J 3) (+ I 3)) (↓R I J))
    )))
 (For (I (Step 1 1 6))
    (For ( J (Step 1 1 6))
    (( ←(↓A I J) (↓R J I))
     ( ←(↓A I J) (↓R ( - 7 I) ( - 7 I)))
    ( ←(↓A I J) (↓R ( - 7 I) J))
    ( ←(↓A I J) (↓R I ( - 7 J)))
  )))
```

```
( For (I (Step 1 1 10))
  (( ← (↓X I J K 1) (↓Y I J K ))
   ( ← (↓X ( + I 3) (⊗ I J ) ( + K 1) I) (⊗ (↓ Y I (+(x 10 I) J)(+ K 1))
        (↓ X ( - ( ⊗ I 3) 4) (- I) K 1)))
  ))); NIL]
```

Result of Example 2

```
(( Declare (Real Array (R (1 6)(1 6))(A (1 6)(1 6))(C (1 12)(1 N)))
           (Integer Array (x (1 10)(1 10)(1 N)(1 15))(Y(1 10)(0 9)(1 N)))
           (Integer I J K L)
           (Real Temp))
  (For (I (Step 1 1 3))
      ( ←G00017 (+ 21 (x 6 I)))
      ( ←G00016 (x 6 I))
      ( ←G00015 ( + 18 ( x 6 I)))
    (For (J (Step 1 1 I)(G00014 ( ⊗ 1 J))(Frequency (G00014)))
      (( ←K (+ ( ←L (x 3 I) 3))
       ( ←( ↓R(+G00014 G00016)) (↓C K L))
       ( ←( ↓R ( + G00014 G0017) (↓R ( + G00014 G0016)))
       ( ←( ↓R( + G00014 G00015)) 0)
    )))
  (For (I (Step 1 1 3))
       ( ←G00023 ( + 18 ( ⊗ 6 I )))
       ( ←G00022 ( ⊗ 6 I))
       ( ←G00021 ( + 21 I))
```

```
    ( ←G00020 ( + 18 I))
    (For (J (Step 1 1 3) ( ←G00019 ( ⊗ 6 J)) (←G00018 ( ⊗ 1 J))
        (Frequency (G00019 2) (G00018 2)))
    (( ←(↓A ( + G00018 G00023 (↓R ( + G00019 G00020)))
     ( ←(↓A ( + G00019 G00021)) (↓R (+ G00018 G00022)))
)))
    (For (I (Step 1 1 6))
       ( ←G00030 (+ 49 ( - ( ⊗ 6 I))))
       ( ←G00029 ( + 7( ⊗ 6 I)))
       ( ←G00028 (+ 42 ( - ( ⊗ 6 I))))
       ( ←G00027 ( ⊗ 6 I))
       (For (J (Step 1 1 6)( ←G00026 (⊗ 6 I)) (←G00025(⊗ 1 J))
         ( ←G00024 ( x -1 J))(Frequency(G00026 1)(G00025 5)(G00024 2)))
        (( ←( ↓A (+ G00025 G00027))( ↓R (+G00026 I)))
         ( ←( ↓A (+ G00025 G00027)) (↓R(+G00024 G00030)))
         ( ←( ↓A (+ G00025 G00027)) (↓R (+ G00025 G00028)))
         ( ←( ↓A (+G00025 G00027)) (↓R ( + G00024 G00029))) )))
    (G00040 (+ 1 ( ⊗ - 400 N) ( ⊗N K)))
    (G00039 ( + 1 ( ⊗10 N J) ( ⊗N K)))
    ( ←G00038 (+ 1 ( ⊗10 J) K))
    ( ←G00037 (+ ( ⊗10 J) K)
    ( ←G00036 ( + ( ⊗300 N) ( ⊗N K ) N))
(For (I (Step 1 1 10) ( ←G00035 ( ⊗ (+ 1 (⊗ 100 N)( ⊗10 N J)) I))
         ( ←G00034 (⊗ 100 I)) ( ←G00033 ( ⊗ 200 I))( ←G00032
                         ( ⊗ ( ⊗100 N) I))
         ( ←G00031 ( ⊗ ( + ( ⊗ 300 N) ( ⊗ - 10 N)) I))
```

```
(Frequency (G00035 1)(G00034 1)(G00033 1) (G00032 1)(G00031 1)))
(( ←(↓ X ( + G00032 G00039)) (↓ Y ( + G00034 G00047 )))
 ( ←(↓ X ( + G00034 G00036 )) ( ⊗ (↓ Y ( + G00033 G00038 ))
      (↓ X ( + G00031 G00040))  )) )) )
```

CONCLUSION:

We have seen how program execution time can be minimized by modifying the subscripted variables to single subscript form and recognizing equivalent expressions so that they need to be evaluated only once.

One of our objectives was to modify the source language program to reflect the above changes and yet keep it machine independent. Thus the additions to the program are also in Algol-statement form. Thus a computer which does not have index registers can still make use of the information provided to generate a better object code.

It is not anticipated that one will use this whole program as a sub-program in a compiler. That would unnecessarily involve many more scans of the program than required. However, many of the routines can be used to perform their appropriate functions within a more generalized set-up, which would generate the object code at the same time.

The following interesting aspects could not be investigated because they are outside the scope of the present investigation:

1. Optimal allocation of index registers: It is obvious that there will be usually more index expressions that have to be stored in index registers than index registers themselves. One way to allocate them is by using the Frequency statement of the modified program. Another would be to scan through the For-loop and see if some index expressions are required only in the first half and others in the latter half of the For-loop; i.e., determine the bounds of usage of index-expressions.

2. Relax the restriction that we do not try to optimize those variables whose subscripts contain variables that occur in the Left-list of the For-loop. This in turn raises numerous other problems which in their generalized form seem to involve more bookkeeping.

3. In the present set-up we assume that 'procedure calls' do not alter the value of the I-coefficients and constant coefficients of the subscripted variables within the For-loop. In the general case it is not obvious whether this assumption is a valid one. One would have to require the programmer to include another declaration statement to indicate that a particular part is not to be optimized.