SCOPE 1.5
A Source-Code Optimization PackagE
for
REDUCE 3.6

=====

User's Manual

J.A. van Hulzen
University of Twente, Department of Computer Science
P.O. Box 217, 7500 AE Enschede, The Netherlands
Email: infhvh@cs.utwente.nl

Abstract

The facilities, offered by SCOPE 1.5, a Source-Code Optimization PackagE for REDUCE 3.6, are presented. We discuss the user aspects of the package. The algorithmic backgrounds are shortly summarized. Examples of straightforward and more advanced usage are shown, both in algebraic and symbolic mode. Possibilities for a combined use of GENTRAN and SCOPE

are presented as well.

Contents

1 Introduction

An important application of computer algebra systems is the generation of code for numerical purposes via automatic or semi-automatic program generation or synthesis. GENTRAN [?, ?, ?, ?], a flexible general-purpose package, was especially developed to assist in such a task, when using MACSYMA or REDUCE. Attendant to **automatic program generation** is the problem of **automatic source-code optimization**. This is a crucial aspect because code generated from symbolic computations often tends to be made up of lengthy arithmetic expressions. Such lengthy codes are grouped together in blocks of straight-line code in a program for numerical purposes. The main objective of SCOPE, our source-code optimization package, has been minimization of the number of (elementary) arithmetic operations in such blocks. This can be accomplished by replacing repeatedly occuring subexpressions, called common subexpressions or cse's for short, by placeholders. We further assume that new statements of the form "placeholder := cse" are inserted correctly in the code. This form of optimization is often helpful in reducing redundancy in (sets of) expressions. A recent application, code generation for an incompressible Navier-Stokes problem [?], showed reduction from 45.000 lines of FORTRAN code to 13.000 lines.

Optimizing compilers ought to deal effectively and efficiently with the average, hand coded program. The enormous, arithmetic intensive expressions, easily producable by a computer algebra system, fall outside the range of the FORTRAN programs, once analyzed and discussed by Knuth [?]. He suggested that optimization of the arithmetic in such a program is slightly overdone. The usual compiler optimization strategy is based on easy detection of redundancy, without assuming the validity of (some) algebric laws (see for instance [?]) Our optimization strategy however, requires the validity of some elementary algebraic laws. We employ heuristic techniques to reduce the arithmetic complexity of the given representation of a set E_{in} of input statements, thus producing a set E_{out} of output assignment statements. E_{in} and E_{out} define blocks of code, which would compute the same exact values for the same exact inputs, thus implicitly proving the correctness of the underlying software. Obviously the use of E_{out} ought to save a considerable amount of execution time in comparison with the application of E_{in}. Johnson et al [?] suggest that such transformations do not destabilize the computations. However this is only apparent after a careful error analysis of both E_{in} and E_{out}. In view of the size of both E_{in} and E_{out} such an analysis has to be automatized as well. Work in this direction is in progress [?, ?, ?].

Although the use of SCOPE can considaribly reduce the arithmetic complexity of a given piece of code, we have to be aware of possible numerical side effects. In addition we have to realize that a mapping is performed from one source language to another source language, seemingly without taking into account the platform the resulting numerical code has to be executed on. Seemingly, because we implemented some facilities for regulating minimal expression length and for producing vector code.

This manual is organized as follows. We begin with some preliminary remarks in section 2. The history and the present status, the optimization strategy and the interplay between GENTRAN and SCOPE are shortly summarized. The basic algebraic mode facilities are presented in section 3. Special SCOPE features are discussed in section 4. Besides facilities for Horner-rules and an extended version of the REDUCE function `structr`, we introduce some tools for extending SCOPE with user defined specialties. File management follows in section 5. In section 6 declaration handling and related issues are discussed, before illustrating our strategy concerning data dependencies and generation of vector code in section 7. In section 8 is shown how a combined used of GENTRAN and SCOPE can be profitable for program-generation. The use of SCOPE in symbolic mode is presented in section 9. A SCOPE syntax summary is given in section 10. For completeness we present guidelines for installing the package in the last section.

Requests

- Comment and complaints about possible bugs can be send to the author using the e-mail address infhvh@cs.utwente.nl. A compact piece with REDUCE code, illustrating the bug, is prefered.
- When SCOPE 1.5 is used in prepairing results, presented in some publication, reference to its use is highly appreciated. A copy of the published document as well.

2 Preliminaries

For completeness we present a historical survey, a birds-eye view of the overall optimization strategy and the interplay between GENTRAN and SCOPE.

2.1 History and Present Status

The first version of the package was designed to optimize the description of REDUCE-statements, generated by NETFORM [?, ?]. This version was tailored to a restrictive class of problems, mainly occurring in electrical network theory, thus implying that the right-hand sides (rhs's) in the input were limited to elements of $\mathbf{Z_2}$[V], where V is a set of identifiers. The second version [?] allowed rhs's from $\mathbf{Z}$[V]. For both versions the validity of the commutative and the associative law was assumed. A third version evolved from the latter package by allowing to apply the distributive law, i.e. by replacing (sub)expressions like $a.b + a.c$ by $a.(b + c)$ whenever possible. But the range of possible applications of this version was really enlarged by redefining V as a set of kernels, implying that almost any proper REDUCE expression could function as a rhs. The mathematical capabilities of this version are shortly summarized in [?], in the context of code generation and optimization for finite-element analysis. This version was further extended [?] with a declaration-module, in accordance with the strategy outlined in [?], chapter 6. It is used in combination with GENTRAN, for the construction of Jacobians and Hessians [?, ?] and also in experiments with strategies for code vectorization [?]. In the meantime the Jacobian-Hessian production package, at present called GENJAC, is further extended with possibilities for global optimization and with some form of loop-differentiation. So in stead of optimizing separate blocks of arithmetic we are now able to optimize complete programs, albeit of a rather specific syntactical structure [?]. The present 1.5 version of SCOPE, is an intermediate between the distributed first version and the future, second version. Version 2 is currently in development and will contain, besides the already existing common sub expression (cse) searches, facilities for structure and pattern recognition. The 1.5 version permits -using the present REDUCE terminology- rounded coefficients, based on the domain features, described in [?], discovery and adequate treatement of a variety of data dependencies, and quotient-optimization, besides a collection of other improvements and refinements for using the facilities in the algebraic mode. Furthermore, an increased flexibility in the interplay between GENTRAN and SCOPE is accomplished. It is used for experiments concerning automatic differentiation [**?**], symbolic-numeric approaches to stability analysis [**?**, **?**] and for code generation for numerically solving the Navier-Stokes equations for incompressible flows [**?**]. An interesting example of its use elsewhere can be found in [**?**].

2.2 Acknowledgements

Many discussions with Victor V. Goldman, Jaap Smit and Paul S. Wang have contributed to the present status of SCOPE. I express my gratitude to the many students, who have also contributed to SCOPE, either by assisting in designing and implementing new facilities, or by applying the package in automated program generation projects in and outside university, thus showing shortcomings and suggesting improvements and extensions. I mention explicitly Frits Berger, Johan de Boer, John Boers, Pim Borst, Barbara Gates, Marcel van Heerwaarden, Pim van den Heuvel, Ben Hulshof, Emiel Jongerius, Steffen Posthuma, Anco Smit, Bernard van Veelen and Jan Verheul.

2.3 The Optimization Strategy in a Birds-eye View

In [**?**, **?**] we described the overall optimization strategy used for SCOPE as a composite function $\mathrm{R}^{-1} \circ \mathrm{T} \circ \mathrm{R}$. The function R defines how to store the input E_0 in an expression database D_0. The inverse function R^{-1} defines the output production. The function T defines the optimization process itself. It essentially consists of a heuristic remodeling of the (extendable and modifiable) expression database in combination with storing information required for a fast retrieval and correct insertion of the detected cse's in the output. This is accomplished by an iteratively applied search, resulting in a stepwise reduction of the arithmetic complexity of the input set, using an extended version of Breuer's grow factor algorithm [**?**, **?**, **?**]. It is applied until no further profit is gained, i.e. until the reduction in arithmetic complexity stops. Before producing output, a finishing touch can be performed to further reduce the arithmetic complexity with some locally applied techniques. Hence T is also a composite function. The overall process can be summarized as follows:

$$\begin{array}{rcrcl} \mathrm{R} & : & \mathrm{E_{in}} = \mathrm{E_0} & \rightarrow & (\mathrm{D_0}, \mathrm{profit_0}) \\ \mathrm{T}_\beta & : & (\mathrm{D_i}, \mathrm{profit_i}) & \rightarrow & (\mathrm{D_{i+1}}, \mathrm{profit_{i+1}})\ ,\ \mathrm{i} = 0, ..., \ \lambda - 1. \\ \mathrm{F} & : & (\mathrm{D}_\lambda, \mathrm{profit}_\lambda) & \rightarrow & D_\lambda \\ \mathrm{R}^{-1} & : & D_\lambda & \rightarrow & \mathrm{E}_\lambda = \mathrm{E_{out}} \end{array}$$

$\mathrm{D_0}$ is created as a result of an R-application performed on input $\mathrm{E_0}$. The termination condition depends on some profit criterion related to the arithmetic complexity of the latest version of the input, D_i. Hence we assume $\mathrm{profit}_i = true$ for $i = 0, \ \cdots, \lambda - 1$ and $\mathrm{profit}_\lambda = false$. The function T is defined by $\mathrm{T} = \mathrm{F} \circ \mathrm{T}_\beta^\lambda$, where T_β defines one iteration step, i.e. one application of the extended version of Breuer's algorithm, and where F defines a finishing touch, resulting in the final version D_λ of $\mathrm{D_0}$, used to produce the output E_λ. It is stated in [?] that the computing time for T_β^λ is O(n.m), where n is the size of $\mathrm{E_{in}}$ and m the number of cse's found during this process. Practical experience showed that the finishing touch can take about 10 % of the actual cpu-time and that its real profit is limited. Therefore its use is made optional.

The wish to optimize source code, defining arithmetic, usually leads an attempt to minimize the arithmetic complexity. This can be accomplished by replacing cse's by placeholders, assuming a new assignment statement "placeholder := cse" is correctly inserted in the code. So most of the cse-searches are done in right hand sides of arithmetic assignment statements.

The search strategy depends on the permissible structure of the arithmetic expressions. We assume these expressions to be multivariate polynomials or rational functions in a finite set of kernels, and presented in some normal form. Let us further assume that scalar placeholders are substituted for the non-scalar kernels, such that back-substitution remains possible, using an adequate information storage mechanism. Then we are left with the interesting question how to define a minimal set of constituents of multivariate polynomials in some normal form norm. Let us take as an example of such a polynomial or rational function $p = 3a + 2b + 3b^2c(3a + 2b)(c + d)^2$. We easily recognize linear forms, i.e. $3a + 2b$ (twice) and $c + d$, possibly raised to some power ($(c + d)^2$), power products, such as b^2c, or monomial parts of products, i.e. $3b^2c$. Hence with some imagination, one realizes that every polynomial can be decomposed in a set of linear forms and a set of power products. When assuming the validity of the commutative and the associative law, one can also realize that we can associate a coefficient matrix with the linear forms and an exponent matrix with the power products. The rows can correspondent with (sub)expressions and the columns with scalar identifiers. The entries are either coefficients or exponents. It is therefore conceivable to make a parser, mapping a set of REDUCE expressions in a database, consisting of two incidence matrices and a function table, such that the original expressions can be retrieved. Taking a group of assignmemnt statements or a list of equations, where in both cases the lhs's function as right hand side recognizers, does not confuse this idea. This rather informal indication merely serves as a suggestion how R and its inverse operation are designed.

So we suggest that we can consider any set of rhs's as being built with linear forms and power products only. An additional remark is worth being made: Non-scalar kernels will in general have non-commutable arguments. These arguments can in turn be arbitrary REDUCE-expressions, which also have to be incorporated in the database. Hence the function table is created recursively.

What is a cse and how do we locate its occurrences? A (sub)expression is common when it occurs repeatedly in the input. The occurrences are, as part of the input, distributed over the matrices, and shown as equivalent (sub)patterns. In fact, we repeatedly search for completely dense (sub)matrices of rank 1. The expression $2a+3c$ is a cse of $e_1 = 2a+4b+3c$ and $e_2 = 4a+6c+5d$, representable by (2,4,3,0) and (4,0,6,5), respectively. We indeed have to assume commutativity, so as to be able to produce new patterns (2,0,3,0,0), (0,4,0,0,1) and (0,0,0,5,2), representing $s = 2a+3c$, $e_1 = 4b+s$ and $e_2 = 5d+2s$, respectivily, and thus saving one addition and one multiplication. Such an additive cse can be a factor in a (sub)product, which in turn can extend its monomial part, when replacing the cse by a new symbol. Therefore an essential part of an optimization step is regrouping of information. This migration of information between the matrices is performed if the Breuer-searches are temporarily completed. After this regrouping the distributive law is applied, possibly also leading to a further regrouping. If at least one of these actions leads to a rearrangement of information the function T_β is again applied. In view of the iterative character of the optimization process we always accept minimal profits.

A similar search is performed to detect multiplicative cse's, for instance occuring in $e_1 = a^2b^4c^3$ and $e_2 = a^4c^6d^5$. However, given a power product $\prod_{i=1}^{m} {x_i}^{a_i}$, any product $\prod_{i=1}^{m} {x_i}^{b_i}$, such that some $b_i \leq a_i$, for i = 1(1)m, can function as a cse. We therefore extend the search for multiplicative cse's by employing this property, and as indicated in [**?**].

The finishing touch F is made to perform one-row and/or one-column searches. Once the extended Breuer-searches do not lead to further reduction in the arithmetic complexity we try -applying it- to improve what is left. The coefficients in (sub)sums can have, possibly locally, a gcd, which can be factored out. One-column operations serve to discover and to replace properly constant multiples of identifiers. As part of the output-process we subject all exponentiations left - at most one for each identifier - to an addition chain algorithm.

2.4 The Interplay between GENTRAN and SCOPE 1.5

The current version of SCOPE is written in RLISP. Like GENTRAN, it can be used as an extension of REDUCE. When SCOPE is loaded GENTRAN is also activated.

If we start a REDUCE session, we create an initial algebraic mode programming environment. All switches get their initial value, such as `ON EXP,PERIOD` and `OFF FORT`. Certain REDUCE commands serve to modify or to enrich the current environment. Others are used to perform calculations, producing formulae. Such a calculation follows a standard pattern, although parts of this repertoire can be influenced by the user, for instance by changing the value of certain switches. Usually execution is a three-step process. First the infix text is parsed into a prefix form. Then the internal algebra is applied on this form, leading to a so-called standard quotient. This quotient is stored on the property list of the identifier functioning as assigned variable for this value. The last step defines the inverse route from internal existence to external presentation in infix form. Occurrences of identifiers are recursively replaced by their standard quotient representation when the internal algebra is applied. Hence the REDUCE simplification strategy follows the imperative programming paradigm.

When loading SCOPE, and thus GENTRAN, the environment is enriched with features for program generation and program optimization. Evaluation of GENTRAN and SCOPE commands differs from the standard REDUCE approach to evaluation. Both packages employ their own storage mechanism. The output is normally produced as a side-effect of the command evaluation. The output is directed to some medium, a file or a screen. Command evaluation is similar in GENTRAN and in SCOPE.

The code generation process of GENTRAN can be viewed as the application of a composite function to an argument, which is almost equivalent with a piece of REDUCE code. Almost, because some GENTRAN specific facilities can be used. We can distinguish between the preprocessing phase, the translation phase and the postprocessing phase. During preprocessing relevant parts of the input are evaluated prior to translation into prefix form. Such a locally performed evaluation can be accomplished through recognition of certain "evaluation markers", i.e. modifications of the traditional assignment symbol `:=`, such as `::=`, `:=:` and `::=:`. The `:=` operator "orders" GENTRAN to translate the statement literally. Addition of an extra colon to the left hand side orders subscript expression evaluation before translation. An extra colon to the right hand side leads to right hand side evaluation, but without application of the storage mechanism of REDUCE. Hence evaluations remain anonymous and are only incorporated in the translatable "text". Another aspect of preprocessing is initialization of the symbol table, using information provided by a `DECLARE` statement. GENTRAN also allows to further rewrite (sets of) arithmetic assignment statements, using the switches `GENTRANOPT` and `GENTRANSEG`, introduced for code optimization (using SCOPE) and segmentation, respectively. It possibly leads to storage of additional information in the symbol table. During the translation phase the final internal form of the code is produced, in combination with formatting specifications and instructions to produce declarations. Postprocessing finally does produce formatted code strings. So essentially, each GENTRAN command has its own seperate translation process, implying that the symbol table, required for the production of declarations, is fresh at the beginning of a GENTRAN command evaluation.

As stated before, a SCOPE command evaluation is also a composite operation. The role of the assignment operators in both GENTRAN and SCOPE is similar. In SCOPE, the locally performed evaluation provides information to be entered in the database D_0. If the declaration feature is activated, the symbol table generation and maintenance mechanism is borrowed from GENTRAN. For output production, we can make a choice from GENTRAN's target language repertoire. When declarations are required, we simply obey the GENTRAN regime as well. D_λ is used to update the symbol table. All cse-names, generated during the optimization process, are typed in accordance with the strategy for dynamic typing, which is discussed in [?], chapter 6. We assume all relevant identifiers of E_{in} to be adequately typed, using SCOPE's `DECLARE` facility, an equivalent of GENTRAN's `DECLARE` statement. The production of D_λ is completely decoupled from the normal REDUCE simplification strategy, because we employ our own expression database.

In principle, the status of REDUCE before and after a GENTRAN or SCOPE command execution is unaltered. In principle, because some minor modifications, although user controlable, may be necessary. The special assignment symbols -also usable in SCOPE- were only introduced as a syntactical instrument to allow internal algebraic actions, decoupled from the standard REDUCE expression processing.

This short excursion into the different evaluation strategies is added to assist in understanding the functioning of the different SCOPE commands and facilities, to be introduced in the next sections.

3 The Basic SCOPE 1.5 Facilities in the Algebraic Mode

REDUCE allows, roughly speaking, two modes of operation in algebraic mode: `ON EXP` or `OFF EXP`. The first is the default setting, leading to expanded forms. The latter gives unexpanded forms, as discussed by Hearn in some detail [**?**, **?**]. It is obvious that the `OFF EXP` setting is in general preferable over the `ON EXP` setting when attempting to optimize the description of a set of assignment statements.

The result of an application of SCOPE can be influenced by the use of certain REDUCE- or SCOPE-switches. The influence of `EXP` is obvious: unexpanded input is more compact than expanded. `ON ACINFO` serves to produce tables with the numbers of arithmetic operations, occuring in E_0 and E_λ, respectively. `ON INPUTC` serves to echo the input, processed by SCOPE. The actual form of the input can be the consequence of locally performed evaluations, before the actual parsing into the database takes place. `ON PRIMAT` can be used to visualize both D_0 and D_λ. `ON PRIALL` finally, can be used instead of `ON ACINFO,INPUTC,PRIMAT`. These SCOPE-switches are initially all turned `OFF`. SCOPE has a facility to visualize the status of all SCOPE-switches and some relevant REDUCE-switches. The current status of all relevant switches can be presented with the command

```
SCOPE_SWITCHES$
```

Example 1

The start of a REDUCE session shows the initial state of REDUCE, directly after loading the SCOPE package. The set of relevant switches is made visible. Besides the REDUCE switches `EVALLHSEQP`, `EXP`, `FORT`, `NAT`, `PERIOD`, $\widehat{\mathrm{R}}$OUNDBF and `ROUNDED` six additional SCOPE switches, i.e. `AGAIN`, `FTCH`, `INTERN`, `PREFIX`, `SIDREL` and `VECTORC`, and the GENTRAN switches `DOUBLE` and `GENTRANOPT` are thus presented. They all wil be discussed in more detail below.

```
REDUCE 3.6, 15-Jul-95 ...

1: load_package nscope$

2: SCOPE_SWITCHES$

 ON  :  exp  ftch  nat  period
 OFF :  acinfo  again  double  evallhseqp  fort  gentranopt  inputc
        intern  prefix  priall  primat  roundbf  rounded  sidrel
        vectorc

3: % etc. ...
```

□

Output is by default given in REDUCE syntax, but FORTRAN syntax is possible in the usual way, e.g. `ON FORT` and `OFF PERIOD`, for instance. The use of other target languages from the GENTRAN repertoire is discussed in section 6.

3.1 The `OPTIMIZE` command: Straightforward use

A SCOPE application is easily performed and based on the use of the following syntax:

<SCOPE_application>	::=	`OPTIMIZE` <object_seq> [`INAME` <cse_prefix>]
<object_seq>	::=	<object>[,<object_seq>]
<object>	::=	<stat> | <alglist> | <alglist_production>
<stat>	::=	<name> <assignment operator> <expression>
<assignment operator>	::=	:= | ::= | ::=: | :=:
<alglist>	::=	{<eq_seq>}
<eq_seq>	::=	<name> = <expression>[,<eq_seq>]
<alglist_production>	::=	<name> | <function_application>
<name>	::=	<id> | <id> (<a_subscript_seq>)
<a_subscript_seq>	::=	<a_subscript>[,<a_subscript_seq>]
<a_subscript>	::=	<integer> | <integer infix_expression>
<cse_prefix>	::=	<id>

A SCOPE action can be applied on one assignment statement. The assigned variable is either a scalar identifier, like `z` in example 2, or a name with subscripts, such as `a(1,1)` in example 3. In stead of one statement a sequence of such statements, separated by comma's, is possible. An alternative is provided by the use of an algebraic mode list, consisting of REDUCE equations. An assigned variable, identifying such a list, is allowed as well. Examples are presented in section 3.2. The function_application is introduced in section 4. Such an application ought to produce an alglist.

The expressions, i.e. rhs's in assignments or equations are legal REDUCE expressions or ought to evaluate to such expressions. Statements inside expressions are allowed, but only useful if these expressions are evaluated, before being optimized. Only integer or rounded coefficients are supported by SCOPE. So we either suppose the default integer setting or allow the switch `ROUNDED` to be turned `ON`.

The optional use of the `INAME` extension in an `OPTIMIZE` command is introduced to allow the user to influence the generation of cse-names. The cse_prefix is an identifier, used to generate cse-names, by extending it with an integer part. If the cse_prefix consists of letters only, the initially selected integer part is 0. All following integer parts are obtained by incrementing the previous integer part by 1. If the user-supplied cse_prefix ends with an integer its value functions as initial integer part. The `gensym`-function is applied when the `INAME`-extension is omitted. The three alternatives are illustrated in example 2.

As stated before minimal profits are accepted during all stages of the optimization process: many small steps may lead to impressive final results. But it can also lead to unwanted details. Therefore, it can be desirable to rerun an optimization request with a restriction on the minimal size of the rhs's. The command

```
SETLENGTH <integer>$
```

can be used to produce rhs's with a minimal arithmetic complexity, dictated by the value of its integer argument. Statements, used to rename function applications, are not affected by the `SETLENGTH` command. The default setting is restored with the command

```
RESETLENGTH$
```

We now illustrate the use of the `OPTIMIZE` command through a number of small examples, being parts of REDUCE sessions. We show in example 2 the effect of the different visualization switches, the use of `SETLENGTH` and `RESETLENGTH` and of the three `INAME` alternatives. In example 3 the effect of some of the GENTRAN and SCOPE input processing features is presented. Some finishing touch activities are illustrated in the examples 4 and 5. The approach towards rational exponents is presented in example 6, while some form of quotient optimization is illustrated in example 7. Finally, we present the differences in `ON/OFF EXP` processing in example 8.

Example 2

The multivariate polynomial `z` is a sum of 6 terms. These terms, monomials, are constant multiples of power products. A picture of D_0 is shown after the input echo. The sums-matrix consists of only one row, identifiable by its Fa(the)r `z`, the lhs. Its exponent is given in the EC (Exponent or Coefficient) field. The 6 monomials are stored in the products-matrix. The coefficients are stored in the EC-fields and the predecessor row index, 0, is given in the Far-field. Before the D_λ picture is given the effect of the optimization process, the output and the operator counts are shown. The optimized form of `z` is obtained by applying the distributive law. The output also shows applications of an addition chain algorithm ([**?**] 441-466) as part of R^{-1}, although its use in example 4 is more apparent.

Observe that the output illustrates the heuristic character of the optimization process: In this particular case the rhs can be written as a polynomial in `g4`, thus saving one extra multiplication.

The `SETLENGTH` command is illustrated too. See also example 12. Application of a Horner-rule may be profitable as well. See, for instance example 16.

```
ON PRIALL$

z:=a^2*b^2+10*a^2*m^6+a^2*m^2+2*a*b*m^4+2*b^2*m^6+b^2*m^2;

      2  2       2  6    2  2         4     2  6    2  2
z := a *b  + 10*a *m  + a *m  + 2*a*b*m  + 2*b *m  + b *m

OPTIMIZE z:=:z$

      2  2       2  6    2  2         4     2  6    2  2
z := a *b  + 10*a *m  + a *m  + 2*a*b*m  + 2*b *m  + b *m

Sumscheme :

   || EC|Far
------------
  0||  1| z
------------
```

```
Productscheme :

   |  0  1  2| EC|Far
---------------------
  1|     2  2|  1| 0
  2|  6     2| 10| 0
  3|  2     2|  1| 0
  4|  4  1  1|  2| 0
  5|  6  2   |  2| 0
  6|  2  2   |  1| 0
---------------------
0  : m
1  : b
2  : a

Number of operations in the input is:

Number of (+/-) operations      : 5
Number of unary - operations    : 0
Number of * operations          : 10
Number of integer ^ operations  : 11
Number of / operations          : 0
Number of function applications : 0

g1 := b*a
g5 := m*m
g2 := g5*b*b
g3 := g5*a*a
g4 := g5*g5
z := g2 + g3 + g1*(2*g4 + g1) + g4*(2*g2 + 10*g3)

Number of operations after optimization is:

Number of (+/-) operations      : 5
Number of unary - operations    : 0
Number of * operations          : 12
Number of integer ^ operations  : 0
Number of / operations          : 0
Number of function applications : 0

Sumscheme :

   |  0  3  4  5| EC|Far
------------------------
```

```
  0|         1  1|  1| z
 15|         2 10|  1| 14
 17|  2  1       |  1| 16
------------------------
0  : g4
3  : g1
4  : g2
5  : g3
```

```
Productscheme :

   |  8  9 10 11 17 18 19 20| EC|Far
-----------------------------------
  7|                  1  1|  1| g1
  8|  1               2   |  1| g2
  9|  1                  2|  1| g3
 10|  2                   |  1| g4
 11|               2      |  1| g5
 14|     1                |  1| 0
 16|            1         |  1| 0
-----------------------------------
8  : g5
9  : g4
10 : g3
11 : g2
17 : g1
18 : m
19 : b
20 : a

OFF PRIALL$
SETLENGTH 2$

OPTIMIZE z:=:z INAME s$

      2  2
s1 := b *m
      2  2
s2 := a *m
              4                        4
z := (a*b + 2*m )*a*b + 2*(s1 + 5*s2)*m  + s1 + s2

RESETLENGTH$

OPTIMIZE z:=:z INAME s1$

s1 := b*a
s5 := m*m
s2 := s5*b*b
s3 := s5*a*a
s4 := s5*s5
z := s2 + s3 + s1*(2*s4 + s1) + s4*(2*s2 + 10*s3)
```

□

Example 3

The input echo below shows the literal copy of the first assignment. This is in accordance with role the GENTRAN assignment operator `:=` ought to play. The second assignment, this time using the operator `::=:`, leads to rhs evaluation (expansion) and lhs subscript-value substitution. Application of the distributive law is refected by the rhs of `a(1,1)` in the presented result.

```
OPERATOR a$ k:=j:=1$ u:=c*x+d$ v:=sin(u)$
ON INPUTC$

OPTIMIZE a(k,j):=v*(v^2*cos(u)^2+u), a(k,j)::=:v*(v^2*cos(u)^2+u) INAME s;

              2       2
a(k,j) := v*(v *cos(u)  + u)

                      2             3
a(1,1) := cos(c*x + d) *sin(c*x + d)  + sin(c*x + d)*c*x + sin(c*x + d)*d

s9 := cos(u)*v
a(k,j) := v*(u + s9*s9)
s6 := x*c + d
s5 := sin(s6)
s10 := s5*cos(s6)
a(1,1) := s5*(s6 + s10*s10)
```

□

Example 4

The effect is shown of a finishing touch application, in combination with FORTRAN output. The value of `S0` is rewritten during output preparation, and the earlier mentioned addition chain algorithm is applied. When turning `OFF` the switch `FTCH` the latter activity is skipped.

```
ON FORT$ OFF PERIOD$

OPTIMIZE z:=(6*a+18*b+9*c+3*d+6*f+18*g+6*h+5*j+5*k+3)^13 INAME s;

      S0=5*(J+K)+3*(3*C+D+1+6*(B+G)+2*(A+F+H))
      S3=S0*S0*S0
      S2=S3*S3
      Z=S0*S2*S2
```

```
OFF FTCH$

OPTIMIZE z:=(6*a+18*b+9*c+3*d+6*f+18*g+6*h+5*j+5*k+3)^13 INAME s;

      Z=(5*(J+K)+3*(3*C+D+1+6*(B+G)+2*(A+F+H)))**13
```

□

Example 5

Recovery of repeatedly occurring integer multiples of identifiers, as part of the finishing touch, is illustrated. This facility is part of the finishing touch and will seemingly be neglected when using `SETLENGTH 2$` instruction in stead of `OFF FTCH`.

```
OPTIMIZE x:=3*a*p, y:=3*a*q, z:=6*a*r+2*b*p,
         u:=6*a*d+2*b*q, v:=9*a*c+4*b*d, w:=4*b INAME s;

s2 := 3*a
x := s2*p
y := s2*q
s0 := 2*b
s3 := 6*a
z := s0*p + s3*r
u := s0*q + s3*d
w := 4*b
v := w*d + 9*c*a

OFF FTCH$

OPTIMIZE x:=3*a*p, y:=3*a*q, z:=6*a*r+2*b*p,
         u:=6*a*d+2*b*q, v:=9*a*c+4*b*d, w:=4*b INAME t;

x := 3*p*a
y := 3*q*a
z := 2*b*p + 6*r*a
u := 2*b*q + 6*d*a
v := 4*d*b + 9*c*a
w := 4*b

ON FTCH$
SETLENGTH 2$

OPTIMIZE x:=3*a*p, y:=3*a*q, z:=6*a*r+2*b*p,
         u:=6*a*d+2*b*q, v:=9*a*c+4*b*d, w:=4*b INAME t;
```

```
x := 3*p*a
y := 3*q*a
z := 2*b*p + 6*r*a
u := 2*b*q + 6*d*a
v := 4*d*b + 9*c*a
w := 4*b
```

□

Example 6

This example serves to show how SCOPE deals with rational exponents. All rational exponents of an identifier are collected. The least common multiple lcm of the denominators of these rationals is computed and the variable is replaced by a possibly newly selected variable name, denoting the variable raised to the power 1/lcm. This facility is only efficient for what we believe to be problems occurring in computational practice. That is easily verified by extending the sum we are elaborating here with some extra terms.

```
ON INPUTC,FORT$

OPTIMIZE z:=:FOR j:=2:6 SUM q^(1/j) INAME s;

      1/6    1/5    1/4    1/3
z := q    + q    + q    + q    + sqrt(q)

      S0=Q**(1.0/60.0)
      S8=S0*S0
      S7=S8*S0
      S5=S8*S7
      S3=S5*S5
      S2=S8*S3
      S1=S7*S2
      S4=S5*s1
      Z=S4+S1+S2+S3+S4*S3
```

□

Example 7

The special attention, given to rational exponents, is not extended to rational coefficients. The script in this example shows four different approaches for dealing with such coefficients using the expressions assigned to `f` and `g`. We start with a literal parsing of the two assignments, leading to a form of D_0, which is based on the present REDUCE strategy for dealing with fixed float numbers in the default integer coefficient domain setting. The four rational numbers $\frac{31}{5}$, $\frac{31}{10}$, $\frac{93}{5}$ and $\frac{93}{10}$ are just like $b^{\frac{1}{5}}$, $\sqrt{\sin(\cdots)}$ and $\sin(\cdots)^{\frac{5}{3}}$ considered as kernels.

The second approach illustrates the effect of simplification in an `OFF ROUNDED` mode prior to parsing. The input expressions are remodeled into rational expressions, the usual internal standard quotient form.

After turning `ON` the switch `ROUNDED` we repeat the previous commands. Again some differences in evaluation can be observed. Literally taken input, the third approach, shows rational exponent optimizations prior to the production of rounded exponents in the output. The last approach, simplification before parsing, leads to a float representation for the rational exponents. SCOPE's exponent optimization features are designed for integer and rational exponents only. Floating point exponentiation is therefore assumed to be a function application.

Further illustrations of operations on quotients are shown in example 22.

```
ON INPUTC$

OPTIMIZE
f:= cos(6.2*a+18.6*(b)^(1/5))/sqrt(sin(3.1*a+9.3*(b)^(1/5))),
g:= sin(6.2*a+18.6*(b)^(1/5))/sin(3.1*a+9.3*(b)^(1/5))^(5/3)
INAME s$
```

```
              31       93   1/5
         cos(----*a + ----*b   )
              5        5
f := -------------------------------
                31       93   1/5
      sqrt(sin(----*a + ----*b   ))
                10       10

            31       93   1/5
       sin(----*a + ----*b   )
            5        5
g := ----------------------------
           31       93   1/5 5/3
      sin(----*a + ----*b   )
           10       10



        1/5
s15 := b
            93       31
s12 := s15*---- + a*----
            5        5
           93         31
s6 := sin(----*s15 + ----*a)
           10         10
          1/6
s14 := s6
s5 := s14*s14*s14
      cos(s12)
f := ----------
         s5
      sin(s12)
g := -----------
      s5*s14*s6


OPTIMIZE
f:=: cos(6.2*a+18.6*(b)^(1/5))/sqrt(sin(3.1*a+9.3*(b)^(1/5))),
g:=: sin(6.2*a+18.6*(b)^(1/5))/sin(3.1*a+9.3*(b)^(1/5))^(5/3)
INAME t$


                    1/5
              93*b    + 31*a
         cos(----------------)
                    5
f := -----------------------------
                    1/5
                93*b    + 31*a
      sqrt(sin(----------------))
                      10
```

```
                             1/5
                         93*b    + 31*a
                   sin(----------------)
                                5
g := ------------------------------------------------
                 1/5                         1/5
             93*b    + 31*a  2/3         93*b    + 31*a
      sin(----------------)   *sin(----------------)
                   10                          10


          1/5
t7 := 93*b    + 31*a
       t7
t2 := ----
       5
           t7
t5 := sin(----)
           10
         1/6
t11 := t5
t4 := t11*t11*t11
      cos(t2)
f := ---------
        t4
       sin(t2)
g := -----------
      t4*t11*t5


ON ROUNDED$

OPTIMIZE
f:= cos(6.2*a+18.6*(b)^(1/5))/sqrt(sin(3.1*a+9.3*(b)^(1/5))),
g:= sin(6.2*a+18.6*(b)^(1/5))/sin(3.1*a+9.3*(b)^(1/5))^(5/3)
INAME s$


                          1/5
        cos(6.2*a + 18.6*b   )
f := -----------------------------
                            1/5
      sqrt(sin(3.1*a + 9.3*b   ))
```

```
                          1/5
       sin(6.2*a + 18.6*b    )
g := --------------------------
                        1/5 5/3
      sin(3.1*a + 9.3*b   )
```

```
           0.2
s5 := 9.3*b    + 3.1*a
s8 := 2*s5
s4 := sin(s5)
         0.166666666667
s10 := s4
s3 := s10*s10*s10
      cos(s8)
f := ---------
        s3
       sin(s8)
g := -----------
      s3*s10*s4


OPTIMIZE
f:=: cos(6.2*a+18.6*(b)^(1/5))/sqrt(sin(3.1*a+9.3*(b)^(1/5))),
g:=: sin(6.2*a+18.6*(b)^(1/5))/sin(3.1*a+9.3*(b)^(1/5))^(5/3)
INAME t$



                 0.2
       cos(18.6*b    + 6.2*a)
f := --------------------------
               0.2        0.5
      sin(9.3*b    + 3.1*a)

                      0.2
            sin(18.6*b    + 6.2*a)
g := ------------------------------------
               0.2        1.66666666667
      sin(9.3*b    + 3.1*a)


           0.2
t6 := 9.3*b    + 3.1*a
t9 := 2*t6
t5 := sin(t6)
      cos(t9)
f := ---------
         0.5
       t5
          sin(t9)
g := -----------------
```

```
    1.66666666667
  t5
```

□

Example 8

The effect of `ON EXP` or `OFF EXP` on the result of a SCOPE-application is illustrated by optimizing the representation of the determinant of a symmetric (3,3) matrix `m`. Besides differences in computing time we also observe that the arithmetic complexity of the optimized version of the expanded representation of the determinant is about the same as the not optimized form of the unexpanded representation.

```
MATRIX m(3,3)$

m(1,1):=18*cos(q3)*cos(q2)*m30*p^2-sin(q3)^2*j30y+sin(q3)^2*j30z-
        9*sin(q3)^2*m30*p^2+j1oy+j30y+m10*p^2+18*m30*p^2$

m(2,1):=
m(1,2):=9*cos(q3)*cos(q2)*m30*p^2-sin(q3)^2*j30y+sin(q3)^2*j30z-
        9*sin(q3)^2*m30*p^2+j30y+9*m30*p^2$

m(3,1):=
m(1,3):=-9*sin(q3)*sin(q2)*m30*p^2$

m(2,2):=-sin(q3)^2*j30y+sin(q3)^2*j30z-9*sin(q3)^2*m30*p^2+j30y+
        9*m30*p^2$

m(3,2):=
m(2,3):=0$

m(3,3):=9*m30*p^2+j30x$

ON ACINFO,FORT$ OFF PERIOD$

OPTIMIZE detm:=:det(m) INAME s;

Number of operations in the input is:

Number of (+/-) operations      : 36
Number of unary - operations    : 0
Number of * operations          : 148
Number of integer ^ operations  : 84
Number of / operations          : 0
Number of function applications : 32

      S2=SIN(REAL(Q2))
      S30=S2*S2
      S3=SIN(REAL(Q3))
      S29=S3*S3
      S31=P*P
      S8=S31*M30
      S32=S8*S8
      S4=S32*J30Y
      S28=S32*S8
      S9=S29*M10
      S10=S30*S29*S29
      S44=COS(REAL(Q3))*COS(REAL(Q2))
      S11=S44*S44
      S20=S31*S8
      S23=S31*J30X
      S22=S29*J30X
      S24=S8*J10Y
      S19=M10*J30Y
      S43=81*S32*J30X
      S35=-S43-(81*S32*J10Y)
      S36=-(729*S29*S28)-(81*S29*S4)
      S37=J30Z-J30Y
      S39=9*S37
      S40=9*J30X
      S41=81*S32*J30Z
      S42=81*S4
      DETM=S42+S36-S35+729*S28+S37*(S22*J10Y+9*S29*S24+S23*S9)+S10*(S42-
     . S41)+S20*S8*81*(M10-S9)+S20*S9*(S39-S40)+S22*S8*(S39-(9*J10Y))+
     . S20*(9*S19+S40*M10)+S24*(S40+9*J30Y)+J30Y*J30X*(J10Y+9*S8)+S28*
     . 729*(S10-S11)+S29*(S41+S35)+S36*S30+S23*S19-(S43*S11)

Number of operations after optimization is:

Number of (+/-) operations      : 30
Number of unary - operations    : 0
Number of * operations          : 59
Number of integer ^ operations  : 0
Number of / operations          : 0
Number of function applications : 4

OFF EXP$

OPTIMIZE detm:=:det(m) INAME t;

Number of operations in the input is:
```

```
Number of (+/-) operations      : 23
Number of unary - operations    : 1
Number of * operations          : 38
Number of integer ^ operations  : 21
Number of / operations          : 0
Number of function applications : 10

      T1=SIN(REAL(Q3))
      T9=T1*T1
      T8=P*P
      T5=T8*M30
      T16=9*T5
      T10=-T16-(9*T5*COS(REAL(Q3))*COS(REAL(Q2)))
      T13=(T16+J30Y-J30Z)*T9
      T15=T13-J30Y
      T0=T15+T10
      T14=T13-T16-J30Y
      T17=T5*SIN(REAL(Q2))
      DETM=81*T17*T17*T14*T9-((T16+J30X)*(T0*T0-(T14*(T15+2*T10-J10Y-(T8
     . *M10)))))

Number of operations after optimization is:

Number of (+/-) operations      : 13
Number of unary - operations    : 0
Number of * operations          : 18
Number of integer ^ operations  : 0
Number of / operations          : 0
Number of function applications : 4
```

We can also use this example to show that correctness of results is easily verified. When storing the result of a SCOPE application in a file, it is of course possible to read the result in again. Then we apply a normal REDUCE evaluation strategy. This implies that all references to cse-names are automatically replaced by their values. We show the "correctness" of SCOPE by storing the optimized version of the expanded form of the determinant of `M`, called `detm1` in file `out.1` and the result of a SCOPE-application on the unexpanded form, `detm2`, in file `out.2`, followed by reading in both files and by subtracting `detm2` from `detm1`, resulting in the value 0. This is of course an ad hoc correctness-proof for one specific example. It is in fact another way of testing the code of the package. We show it as a direct continuation of the previous determinant calculations.

This example also serves to show that the `OPTIMIZE` command can be extended with the `OUT` option. The keyword `OUT` has to be followed by a file-name. This file is properly closed and left in a readable form, assuming printing is produced in a `OFF NAT` fashion. SCOPE's file management features are discussed in more detail in section 5.

```
OFF ACINFO,FORT,NAT$ ON EXP$

OPTIMIZE detm1:=:det(M) OUT "out.1" INAME s;

OFF EXP$

OPTIMIZE detm2:=:det(M) OUT "out.2" INAME t;

ON NAT$
IN "out.1","out.2"$

detm1-detm2;

0
```

□

So far we presented via some examples straightforward algebraic mode use. The output is produced as a side-effect. However, optimization results can easily be made operational in algebraic mode. The parameterless function

`ARESULTS`

delivers the result of the directly preceding `OPTIMIZE` command in the form of a list of equations, corresponding with the sequence of assignment statements, shown either in REDUCE syntax or in the syntax of one of GENTRAN's target languages. But, we need to operate carefully. Application of a variety of assignment operators can easily bring in identifiers, representing earlier produced algebraic values. They will be substituted automatically, when referenced in rhs's in the list, produced with an `ARESULTS` application. Therefore, we implemented a protection mechanism. Before delivering output produced by `ARESULTS`, we make the system temporarily deaf for such references. The possibly game-spoiling algebraic values are stored at a seemingly anonymous place. All identifiers, subjected to this special treatement, can be made visible with the command

`RESTORABLES;`

Their original status can be restored, either globally with the command

    RESTOREALL$

or selectively with the instruction

    ARESTORE <subsequence>$

This subsequence is built with names, selected from the list of `RESTORABLES`, and separated by comma's. Information restoration is only possible before the next `OPTIMIZE` command.

Example 9

The use of these commands is now illustrated. A further explanation is given in the form of comment in the script.

```
u:=a*x+2*b$ v:=sin(u)$ w:=cos(u)$ f:=v^2*w;

                                           2
f := cos(a*x + 2*b)*sin(a*x + 2*b)

OFF EXP$

OPTIMIZE f:=:f,g:=:f^2+f INAME s;

s3 := x*a + 2*b
s2 := sin(s3)
f := s2*s2*cos(s3)
g := f*(f + 1)

alst:=ARESULTS;

alst := {s3=a*x + 2*b,

         s2=sin(s3),

                      2
         f=cos(s3)*s2 ,

         g=(f + 1)*f}

% ---
% SCOPE is made deaf for the standard reference mechanism for algebraic
% variables. However the rhs's in the list alst are simplified before
% being shown. It explains the differences between the layout in the
% alst items and the results, presented by the OPTIMIZE-command itself.
% ---
```

```
RESTORABLES;

{f}

f;

f

ARESTORE f$

f;
                                 2
cos(a*x + 2*b)*sin(a*x + 2*b)

% ---
% f is re-associated with its original value. This can lead to a modified
% presentation of some of the rhs's of alst.
% ---

alst;

{s3=a*x + 2*b,

 s2=sin(s3),

             2
 f=cos(s3)*s2 ,

                                    2                                    2
 g=(cos(a*x + 2*b)*sin(a*x + 2*b)  + 1)*cos(a*x + 2*b)*sin(a*x + 2*b) }

OPTIMIZE f:=:f,g:=:f^2+f INAME s;

s3 := x*a + 2*b
s2 := sin(s3)
f := s2*s2*cos(s3)
g := f*(f + 1)

alst2:=ARESULTS$

OPTIMIZE f:=:f,g:=:f^2+f INAME s;

g := f*(f + 1)
```

```
% ---
% The algebraic value, which was associated with f, is permanently
% lost. It ought to be restored before a new OPTIMIZE command is given.
% Therefore f:=:f produced an identity, which is redundant in terms of
% code production. More details about removal of redundant code are
% given in section 7, when discussing data dependencies and related topics.
% ---

RESTOREALL$

f;

f
```

□

3.2 The function ALGOPT: Straightforward use

The function `ALGOPT` accepts up to three arguments. It can be used in stead of the `OPTIMIZE` command. It returns the optimization result, like `ARESULTS`, in the form of a list of equations. Since the `ARESULTS` mechanism is applied as well, the pre-`ALGOPT`-application situation can be restored with `RESTOREALL` or partly and selective with `ARESTORE`, using information, providable by an application of the function `RESTORABLES`.

The first argument of `ALGOPT`, like the other two optional, is the equivalent of the alglist or alglist_production in the earlier introduced syntax of a SCOPE_application. The second argument can be used to inform SCOPE that input from file(s) have to be processed. We survey SCOPE's file management features in section 5. So we omit a further discussion now. The last argument correspondents with the cse_prefix of the `INAME` option of the `OPTIMIZE` command. The extension of the SCOPE_application syntax, needed to include possible `ALGOPT` activities, is:

```
<SCOPE_application>  ::=  ··· |
                          ALGOPT(<a_object_list>[,<string_id_list>][,<cse_prefix>]) |
                          ALGOPT([<a_object_list>,]<string_id_list>[,<cse_prefix>])
<a_object_list>  ::=  <a_object>  | {<a_object>[,<a_object_seq>]}
<a_object_seq>   ::=  <a_object>[,<a_object_seq>]
<a_object>       ::=  <id>  |  <alglist>  |  <alglist_production>  | {<a_object>}
```

We require at least one actual parameter, here the a_object_list. Its syntactical structure allows to apply a GENTRAN-like repertoire in an algebraic mode setting. The a_object's can either be an alglist identifier, an alglist producing function application, or an alglist itself. An alglist identifier can be either a scalar or a matrix or array entry. The alglist producing functions will be discussed in section 4. An alglist has the structure of an algebraic mode list; its elements are either a_object's or equations of the form lhs = rhs. Such equations correspondent with the "take literal" GENTRAN operator `:=` facility in the setting of an `OPTIMIZE` command (see also section 4 for a further discussion). The alternatives, i.e. uses of `::=`, `:=:` or `::=:`, are also covered by the a_object syntax. The examples, given in this subsection, show that simplification of an algebraic list of equations leads to right hand side simplification, corresponding with the effect of the colon-added-to-the-right-extension of the assignment operator. However, as illustrated by example 13, some care has to be taken when operating in `OFF EXP` mode. Turning `ON` the switch `EVALLHSEXP`, can lead to lhs evaluations, corresponding with the extra-colon-to-the-left strategy. But we have to be aware of the instanteneous evaluation mechanism for matrix and array entries, when referenced.

We present some examples of possible use of the `ALGOPT` function. In example 10 a straightforward application is given. In example 11 follows an ilustration of a possible strategy concerning optimizing sets of array- and/or matrix-entries. Then, in example 12, possible SCOPE assistance in problem analysis is shown. Finally in example 13 some differences in simplification and their influence on optimization are discussed. We also introduce and explain the role of the SCOPE switch `SIDREL`.

Example 10

A number of possible alglist elements is presented in the script. The first three elements of the actual parameter define values, obtained via the usual algebraic mode list evaluation mechanism. The last two will be processed literally. So, the actual parameter for `ALGOPT` is composed of the scalar `alist`, a list consisting of the matrix element `m(1,1)`, the array element `ar(2,2)`, nested even deeper, and two equations. Before an `ALGOPT` argument is optimized it is flattened by the SCOPE parser into a list of equations, the algebraic mode equivalent of the sequence of assignments in the `OPTIMIZE` context. Evaluation of an `ALGOPT` application leads to an algebraic mode list of equations, with optimized rhs's. The cse_prefix was seemingly superfluous, because all its references disappeared by back-substitution before output-processing started. See also example 13.

Since an `ALGOPT` application always results in an algebraic mode list, one can not use this feature for production of code in one of GENTRAN's target languages. To facilitate the translation of the result of an `ALGOPT` application, we extended the syntax of the `OPTIMIZE` input repertoire, such that alglst_production's are processable by `OPTIMIZE` as well, as illustrated in the script of this example and in example 13

```
OFF EXP$
ARRAY ar(2,2)$ MATRIX m(2,2)$

alst:={p1=a+b,p2=(a+b)^2}$

m(1,1):={q1=c+d,q2=(c+d)^2}$

ar(2,2):={r1=(a+b)*(c+d),r2=(a+b)^2*(c+d)^2}$

optlst:=ALGOPT({alst,{m(1,1)},{{ar(2,2)}},
                t1=(a+b)*(c+d)^2,t2=(c+d)*(a+b)^2},s);

optlst := {p1=a + b,
                2
           p2=p1 ,
           q1=c + d,
                2
           q2=q1 ,
           r1=p1*q1,
                2
           r2=r1 ,
           t1=q1*r1,
           t2=p1*r1}

OPTIMIZE optlst$

p1 := a + b
p2 := p1*p1
q1 := c + d
q2 := q1*q1
r1 := q1*p1
r2 := r1*r1
t1 := r1*q1
t2 := r1*p1
```

□

Example 11

In example 8 we introduced a symmetric (3,3)-matrix `m`. We present an alternative computation of its determinant. We start with building a list of equations, with rhs's, being the non-zero entries of `m`, relevant for the computation. The lhs's are produced with the `mkid` function. These newly generated names are assigned to the matrix-entries as well. Finally we add the definition of the computation of the determinant of `m`, in terms of the redefined entries, to this list. For the construction of the value of `mlst` we applied both the lhs and rhs evaluation mechanism. Observe also that, due to the redefinition process, the original values of the entries of the matrix `m` are lost. We can optimize `mlst`, using either an `OPTIMIZE` command or an `ALGOPT` application. The reduction in arithmetic is not yet impressive here, certainly comparing it with the non-expanded, optimized form in the earlier example 8. (See also example 13 for additional comment). However, working with larger and non-symmetric matrices will certainly improve results, when applying a comparable strategy.

Observe that the syntax of permissible `ALGOPT` a_object's does not allow to use matrix or array names to compactly identify the complete set of their entries. The script in this example shows that such a facility is easily made. This possibility exists already for matrices in a GENTRAN setting (see also example 22 in section 8).

```
% ---
% We assume the matrix m to be known already.
% ---

mlst:={}$ l:=-1$

OFF EXP$ ON EVALLHSEQP$

FOR j:=1:3 DO FOR k:=j:3 DO
 IF m(j,k) neq 0 THEN
  << s:=mkid(t,l:=l+1);
     mlst:=append(mlst,{s=m(j,k)});
     m(j,k):=m(k,j):=s
  >>$

OFF EVALLHSEQP$
```

```
m;

[t0  t1  t2]
[          ]
[t1  t3  0 ]
[          ]
[t2  0   t4]


mlst:=append(mlst,{detm=det(m)});

                                     2         2
mlst := {t0= - (j30y - j30z + 9*m30*p )*sin(q3)

                                    2                      2
          + 18*cos(q2)*cos(q3)*m30*p  + j10y + j30y + m10*p

                    2
          + 18*m30*p ,

                                     2         2
         t1= - (j30y - j30z + 9*m30*p )*sin(q3)

                                   2                 2
          + 9*cos(q2)*cos(q3)*m30*p  + j30y + 9*m30*p ,

                                      2
         t2= - 9*sin(q2)*sin(q3)*m30*p ,

                                     2         2                 2
         t3= - (j30y - j30z + 9*m30*p )*sin(q3)  + j30y + 9*m30*p ,

                          2
         t4=j30x + 9*m30*p ,

                         2         2
         detm=(t0*t3 - t1 )*t4 - t2 *t3}

ON ACINFO,FORT$ OFF PERIOD$

OPTIMIZE mlst INAME s;

Number of operations in the input is:
```

```
Number of (+/-) operations      : 19
Number of unary - operations    : 1
Number of * operations          : 33
Number of integer ^ operations  : 16
Number of / operations          : 0
Number of function applications : 9

      S0=SIN(REAL(Q3))
      S7=P*P
      S5=S7*M30
      S4=S5*COS(REAL(Q3))*COS(REAL(Q2))
      S13=9*S5
      S11=(S13+J30Y-J30Z)*S0*S0
      T0=J30Y+J10Y+18*(S4+S5)+S7*M10-S11
      T3=S13+J30Y-S11
      T1=T3+9*S4
      T2=-(S13*SIN(REAL(Q2))*S0)
      T4=S13+J30X
      DETM=T4*(T3*T0-(T1*T1))-(T2*T2*T3)

Number of operations after optimization is:

Number of (+/-) operations      : 13
Number of unary - operations    : 1
Number of * operations          : 17
Number of integer ^ operations  : 0
Number of / operations          : 0
Number of function applications : 4
```

□

Example 12

We now illustrate that information, produced by SCOPE, can possibly also play a role in computations in algebraic mode. Let $A.\vec{x} = \vec{b}$ be given by

$$\begin{bmatrix} -1 & 2 & -2 & 1 & 3 & 2 \\ -2 & 4 & -4 & 2 & -2 & 3 \\ 1 & 1 & 1 & 1 & 2 & 4 \\ 2 & -2 & -1 & 1 & -1 & -2 \\ 3 & 1 & -4 & 1 & 1 & 2 \\ -1 & -5 & 1 & 1 & 3 & 6 \end{bmatrix} . \begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \\ 10 \\ -3 \\ 4 \\ 5 \end{bmatrix} .$$

This artificial system is constructed for illustrative purposes. Its solution is simply $x_i = 1, i = 1,..,6$. But straightforward inspection shows that

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \text{ where } A_1 = \begin{bmatrix} -1 & 2 & -2 & 1 \\ -2 & 4 & -4 & 2 \end{bmatrix} \text{ and } A_4 = \begin{bmatrix} 2 & 4 \\ -1 & -2 \\ 1 & 2 \\ 3 & 6 \end{bmatrix}.$$

We can use ALGOPT to "discover" and thus to employ this information. The system is introduced in the form of assignment statements $e_i = (\sum_{j=1}^{6} a_{ij} \cdot x_j) - b_i,\ i = 1, \cdots, 6$. We use alst, identifying the set of equations (see command 9), as actual parameter for ALGOPT, leading to an algebraic list, identified by reslst (see command 10). We recognize `g2 = g6 + x5 (= x5 + 2x6)` and `g1 = g3 + g5 + -2x3 (= -x1 + 2x2 - 2x3 + x4)`. Through command 12 we require cse's to have an arithmetic complexity of a least 4. We then find g1 directly, now called g8, because we continue applying the function gensym; the cse_prefix was left out as actual parameter. The solve function is applied (command 14) to obtain rootset1, a list of values for x5 and x6, expressed in the parameter g8. After assigning g8 its value in algebraic mode and resetting the algebraic values of ei, $i = 1, \cdots, 6$ with RESTOREALL instructions (the commands 11 and 16), we can obtain the solution of the subsets, denoted by rootset1 and rootset2.

```
1: LOAD_PACKAGE nscope$

2: e1:=2*x6+3*x5+x4-2*x3+2*x2-x1-5$

3: e2:=3*x6-2*x5+2*x4-4*x3+4*x2-2*x1-1$

4: e3:=2*x5+4*x6+x1+x2+x3+x4-10$

5: e4:=-x5-2*x6+2*x1-2*x2-x3+x4+3$

6: e5:=x5+2*x6+3*x1+x2-4*x3+x4-4$

7: e6:=3*x5+6*x6-x1-5*x2+x3+x4-5$

8: solve({e1,e2,e3,e4,e5,e6},{x1,x2,x3,x4,x5,x6});

{{x1=1,x2=1,x3=1,x4=1,x5=1,x6=1}}

9: alst:={e1=e1,e2=e2,e3=e3,e4=e4,e5=e5,e6=e6}$

10: reslst:=ALGOPT alst;
```

```
reslst := {g3= - x1 + x4,

           g5=2*x2,

           g1=g3 + g5 - 2*x3,

           g6=2*x6,

           e1=g1 + g6 + 3*x5 - 5,

           e2=2*g1 - 2*x5 + 3*x6 - 1,

           g2=g6 + x5,

           g4=x2 + x4,

           e3=2*g2 + g4 + x1 + x3 - 10,

           e4= - g2 - g5 + 2*x1 - x3 + x4 + 3,

           e5=g2 + g4 + 3*x1 - 4*x3 - 4,

           e6=3*g2 + g3 - 5*x2 + x3 - 5}

11: RESTOREALL$

12: SETLENGTH 4$

13: reslst:=ALGOPT alst;

reslst := {g8= - x1 + 2*x2 - 2*x3 + x4,

           e1=g8 + 3*x5 + 2*x6 - 5,

           e2=2*g8 - 2*x5 + 3*x6 - 1,

           e3=x1 + x2 + x3 + x4 + 2*x5 + 4*x6 - 10,

           e4=2*x1 - 2*x2 - x3 + x4 - x5 - 2*x6 + 3,

           e5=3*x1 + x2 - 4*x3 + x4 + x5 + 2*x6 - 4,

           e6= - x1 - 5*x2 + x3 + x4 + 3*x5 + 6*x6 - 5}
```

```
14: rootset1:=solve({part(reslst,2,2),part(reslst,3,2)},{x5,x6});

                  g8 + 13       - 8*g8 + 13
rootset1 := {{x5=---------,x6=--------------}}
                    13               13
15: g8:=part(reslst,1,2);

g8 :=  - x1 + 2*x2 - 2*x3 + x4

16: RESTOREALL$

17: rootset2:=solve(sub(rootset1,{e3,e4,e5,e6}),{x1,x2,x3,x4});

rootset2 := {{x1=1,x2=1,x3=1,x4=1}}

18: rootset1:=sub(rootset2,rootset1);

rootset1 := {{x5=1,x6=1}}
```

□

Example 13

The script in example 11 suggests that we can easily copy GENTRAN's assignment features by some listprocessing in algebraic mode. However, we have to operate carefully. In the script of the present example we introduce an expression denoted by `f`. Production of a number of its partial (higher) derivatives is a straightforward mechanism to assist in constructing a set of assignment statements, containing lots of cse's. Inspection of the values, in `OFF EXP` mode assigned to `faa`, `tst1` and `tst2`, respectively, learns that the value of `mlst` in example 11 may be improvable.

```
u:=a*x+2*b$ v:=sin(u)$ w:=cos(u)$ f:=v^2*w$

OFF EXP$

faa:=df(f,a,2);

                          2                    2                  2
faa := (2*cos(a*x + 2*b)  - 7*sin(a*x + 2*b) )*cos(a*x + 2*b)*x

tst1:={faa=df(f,a,2)};

                             3   2                                      2   2
```

```
tst1 := {faa=2*cos(a*x + 2*b) *x  - 7*cos(a*x + 2*b)*sin(a*x + 2*b) *x }

tst2:={faa=(faa:=df(f,a,2))};

                              2                   2                  2
tst2 := {faa=(2*cos(a*x + 2*b)  - 7*sin(a*x + 2*b) )*cos(a*x + 2*b)*x }
```

We produce an optimized version of the value of `tlst`, using `ALGOPT`. Switching `ON INPUTC` and `PRIMAT` results in an input echo, indeed showing expanded rhs's and a vizualized picture of the `Sumscheme` of D_λ. We skipped the D_0-picture and the rest of the D_λ-picture from the script. The value of `reslst` shows the patterns `s14 = -7.f.x + 2.s9.x` and `fbb = -28.f + 8.s9`. The presented `Sumscheme` of D_λ suggests that `fbb` and `s13` (see the Fa(the)r entries) seemingly have nothing in common. But `s14` stands for `2.s6 - 7.s7`, because, column 8 has to be identified with `s7`, etc. Since both `S6` and `S7` occur only once in D_λ, their value replaces them in the output. It is an illustration of the heuristic character of the optimization process. Optimization of the value of `reslst` shows that the repeated pattern is now recognized.

```
tlst:={f=f,fa=df(f,a),fb=df(f,b),faa=df(f,a,2),
       fab=df(f,a,b),fba=df(f,b,a),fbb=df(f,b,2)}$

ON INPUTC,PRIMAT$

reslst:=ALGOPT(tlst,s);

                                  2
f := cos(a*x + 2*b)*sin(a*x + 2*b)

                      2                                  3
fa := 2*cos(a*x + 2*b) *sin(a*x + 2*b)*x - sin(a*x + 2*b) *x

                      2                                  3
fb := 4*cos(a*x + 2*b) *sin(a*x + 2*b) - 2*sin(a*x + 2*b)

                       3  2                                  2  2
faa := 2*cos(a*x + 2*b) *x  - 7*cos(a*x + 2*b)*sin(a*x + 2*b) *x

                       3                                     2
fab := 4*cos(a*x + 2*b) *x - 14*cos(a*x + 2*b)*sin(a*x + 2*b) *x

                       3                                     2
fba := 4*cos(a*x + 2*b) *x - 14*cos(a*x + 2*b)*sin(a*x + 2*b) *x
```

```
                       3                                  2
fbb := 8*cos(a*x + 2*b)  - 28*cos(a*x + 2*b)*sin(a*x + 2*b)

Sumscheme :

   |  3  4  5  6  7  8  9 10 11 12 32| EC|Far
---------------------------------------------
  3|                           1  2|  1| s3
 20|         -28  8                |  1| fbb
 37|                -7  2          |  1| s14
 38|                      -1  2    |  1| s15
---------------------------------------------
3  : s3
4  : s15
5  : s14
6  : s4
7  : s9
8  : s7
9  : s6
10 : s10
11 : s5
12 : s8
32 : b

reslst := {s3=a*x + 2*b,

           s0=cos(s3),

                3
           s9=s0 ,

           s2=sin(s3),

           s12=s0*s2,

           f=s12*s2,

                            3
           s15=2*s0*s12 - s2 ,

           fa=s15*x,

           fb=2*s15,
           s14= - 7*f*x + 2*s9*x,

           faa=s14*x,

           fab=2*s14,

           fba=fab,

           fbb= - 28*f + 8*s9}

OFF INPUTC,PRIMAT$

OPTIMIZE reslst$

s3 := 2*b + x*a
s0 := cos(s3)
s9 := s0*s0*s0
s2 := sin(s3)
s12 := s2*s0
f := s12*s2
s15 := 2*s12*s0 - s2*s2*s2
fa := s15*x
fb := 2*s15
g3 := 2*s9 - 7*f
s14 := g3*x
faa := s14*x
fab := 2*s14
fba := fab
fbb := 4*g3
```

Repeating this process, this time with an `OPTIMIZE` command to begin with, learns that the `OFF EXP` mode is now effective. But this time, and for similar reasons, the assignments `fbb = 4.s9.s0` and `s12 = s9.s0.x` still have a subexpression in common. Now the `Productscheme` of D_λ helps understanding the phenomenon; again we skipped for shortness the rest of the information, provided by the `ON PRIMAT` status of SCOPE. Internally `s12` denotes the product `s4.s9`, where `s4 = x.s0`. The cse `s4` disappeared from the output. An `ALGOPT` application leads to the "discovery" of the cse `g10 = s0.s9`.

```
f:=v^2*w;
                                   2
f := cos(a*x + 2*b)*sin(a*x + 2*b)

ON INPUTC,PRIMAT$
```

```
OPTIMIZE f:=:f,fa:=:df(f,a),fb:=:df(f,b),faa:=:df(f,a,2),
         fab:=:df(f,a,b),fba:=:df(f,b,a),fbb:=:df(f,b,2) INAME s$

                                    2
f := cos(a*x + 2*b)*sin(a*x + 2*b)

                        2                 2
fa := (2*cos(a*x + 2*b)  - sin(a*x + 2*b) )*sin(a*x + 2*b)*x

                          2                 2
fb := 2*(2*cos(a*x + 2*b)  - sin(a*x + 2*b) )*sin(a*x + 2*b)

                         2                   2                  2
faa := (2*cos(a*x + 2*b)  - 7*sin(a*x + 2*b) )*cos(a*x + 2*b)*x

                           2                   2
fab := 2*(2*cos(a*x + 2*b)  - 7*sin(a*x + 2*b) )*cos(a*x + 2*b)*x

                           2                   2
fba := 2*(2*cos(a*x + 2*b)  - 7*sin(a*x + 2*b) )*cos(a*x + 2*b)*x

                           2                   2
fbb := 4*(2*cos(a*x + 2*b)  - 7*sin(a*x + 2*b) )*cos(a*x + 2*b)

s3 := x*a + 2*b
s0 := cos(s3)
s2 := sin(s3)
s6 := s2*s2
f := s6*s0
s14 := 2*s0*s0
s13 := (s14 - s6)*s2
fa := s13*x
fb := 2*s13
s9 := s14 - 7*s6
s12 := s9*s0*x
faa := s12*x
fab := 2*s12
fba := fab
fbb := 4*s9*s0

Productscheme :

   |  0  2  3  4  5 12 14 18 19 20 21 22 23| EC|Far
```

```
---------------------------------------------------
  0|                    1           1      |  1| f
  5|     1                             1   |  1| fa
  9|     1                                 |  2| fb
 13|        1                          1   |  1| faa
 17|  1                                    |  1| fab
 21|  1                                    |  1| fba
 25|           1                    1      |  4| fbb
 29|                                1  1   |  1| s4
 30|                                   1  1|  1| s5
 31|                             2         |  1| s6
 33|                                2      |  1| s8
 37|           1              1            |  1| s12
 38|              1              1         |  1| s13
 39|                 1                     |  2| s14
 40|        1                              |  2| s15
---------------------------------------------------
0  : s15
2  : s13
3  : s12
4  : s9
5  : s10
12 : s8
14 : s6
18 : s5
19 : s4
20 : s2=sin(s3)
21 : s0=cos(s3)
22 : x
23 : a

ALGOPT ARESULTS;

{s3=a*x + 2*b,

 s0=cos(s3),

 s2=sin(s3),

      2
 s6=s2 ,

 f=s0*s6,
         2
 s14=2*s0 ,

 s13=(s14 - s6)*s2,

 fa=s13*x,

 fb=2*s13,

 s9=s14 - 7*s6,

 g10=s0*s9,

 s12=g10*x,

 faa=s12*x,

 fab=2*s12,

 fba=fab,

 fbb=4*g10}
```

This script is shown for different reasons. It illustrates the heuristic character of the optimization process. We optimize, but do not guarantee the optimal solution. It also shows how easily repeated SCOPE applications can be accomplished. Hence commands like "`ALGOPT ARESULTS`;", "`ALGOPT ALGOPT` ⋯ ;" or "`OPTIMIZE ALGOPT` ⋯ ;" are all possible. However, it is sometimes better to avoid such a combination when a `RESTOREALL` instruction has to follow the first application. A more detailed discussion about these possibilities is given in section 4, and especially in section 4.1. An additional reason was, to stipulate that SCOPE's actual parameters have to be built carefully.

This example is also used to illustrate the role, which the switch `SIDREL` can possibly play. When turned it `ON` the finishing touch F (see subsection 2.3) is omitted and all non-additive cse's are substituted back, thus producing a possibly still rewritten input set, which consists of toplevel input and additive cse's only. A simple straightforward backsubstitution mechanism is applied on the optimization result before it is presented to the user. Seemingly, it can lead to surprises as shown below by the differences between the presentations of `s15`, `s14` and `fbb` when again optimizing the contents of `tlst`. This effect disappeares when using `SETLENGTH`.

The switch `SIDREL` was introduced in SCOPE quite long ago. By that time Hearn was wondering [**?**, **?**] if (parts of) SCOPE output, presented in algebraic mode, can be used as input for a Gröbner-base algorithm application, thus attempting to assist in expression restructuring leading to improved expression representations.

```
ON SIDREL$

ALGOPT(tlst,s);

{s3=a*x + 2*b,

                  2
 f=cos(s3)*sin(s3) ,

              2                   3
 s15=2*cos(s3) *sin(s3) - sin(s3) ,

 fa=s15*x,

 fb=2*s15,

                                              2              3
 s14= - 7*cos(a*x + 2*b)*sin(a*x + 2*b) *x + 2*cos(s3) *x,

 faa=s14*x,

 fab=2*s14,

 fba=2*s14,

                                               2            3
 fbb= - 28*cos(a*x + 2*b)*sin(a*x + 2*b)  + 8*cos(s3) }

SETLENGTH 4$

ALGOPT(tlst,s);

                                 2
{f=cos(a*x + 2*b)*sin(a*x + 2*b) ,

                      2                                  3
 s15=2*cos(a*x + 2*b) *sin(a*x + 2*b) - sin(a*x + 2*b) ,
 fa=s15*x,

 fb=2*s15,

                      3                                   2
 s14=2*cos(a*x + 2*b) *x - 7*cos(a*x + 2*b)*sin(a*x + 2*b) *x,

 faa=s14*x,

 fab=2*s14,

 fba=2*s14,

                      3                                  2
 fbb=8*cos(a*x + 2*b)  - 28*cos(a*x + 2*b)*sin(a*x + 2*b) }
```

□

4 Special SCOPE 1.5 Features

Part of the input syntax for the function ALGOPT was left undiscussed in section 3.2. It was the permissable form for (parts of) the actual parameter, defining function applications, producing an alglist. The alglist is an algebraic mode list, consisting either of equations of the form lhs = rhs or of constructs evaluating into an alglist or referencing an alglist. In section 4.1 is explained which type of user defined functions lead to permissable function applications as (part of an) actual parameter for an OPTIMIZE command or an ALGOPT application. Tools are provided for building a SCOPE library. Already available facilities, designed along these lines, cover **structure recognition**, presented in section 4.2 and **Horner-rule** based expression rewriting, surveyed in section 4.3.

4.1 Towards a SCOPE 1.5 Library

Design and implementation of an algebraic or symbolic procedure, returning a list of equations in algebraic mode, is straightforward as long as the number of formal parameters is exactly known. Let us call such procedures functions of NORMAL-type. When formal parameters are not required, the so-called ENDSTAT-variant can be used. One simply associates an indicator stat, with value endstat, with the function name f-name, using lisp(put('f-name,'stat,'endstat))$ as instruction. Such a function will be said to be of ENDSTAT-type.

The so-called PSOPFN-type function is similar to the FEXPR-type function in symbolic mode. It may have an arbitrary number of unevaluated parameters. Special attention is made possible by modifying the function reval1, used in both reval and aeval. The relevant section of the evaluator is:

```
symbolic procedure aeval u; reval1(u,nil);
symbolic procedure reval u; reval1(u,t);

symbolic procedure reval1(u,v);
.......................
else if x:=get(car u,'psopfn)
 then << u:=apply(x,list cdr u);
         if x:=get(x,'cleanupfn) then u:=apply(x,list(u,v));
         return u
      >>
.......................
```

The actual parameter `u` of `reval1` is a function application in prefixform: (`function-name arg1 ... argn`). The `function-name` is replaced by the value of the indicator `psopfn`. The thus modified S-expression is evaluated. This mechanism leaves control over evaluation of (part of) the arguments, collected in `cdr u`, to the designer of the function hidden behind the `psopfn` value. We employed this simple mechanism to implement `ALGOPT` and some of the features, to be discussed in section 4.2 and section 4.3. A possible re-evaluation step, based on the use of the value of the indicator `cleanupfn` was not necessary; it is not yet allowed in the SCOPE context.

The different combinations, suggested in example 13, such as `ALGOPT ALGOPT` or `ALGOPT ARESULTS`, are merely examples of a general rule:

`ENDSTAT`-*,* `NORMAL`- *and* `PSOPFN`-*type functions, delivering an alglist when applied, are all applicable as actual parameter or as element of an alglist, functioning as actual parameter in an* `OPTIMIZE` *command or an* `ALGOPT` *application.*

So, in principle, special features, providing a form of preprocessing, can be designed and implemented as extension of the default optimization repertoire. Of course additional function types are conceivable. We illustrate the potential of this facility with a simple example. Further examples follow in section 4.2 and section 4.3.

Example 14

The procedures `asquares` and `repeated_squaring` define the production of lists of equations. The lhs's function as name and the rhs's as the computational rules. Application of these functions shows how easy a user can provide new features, usable in a SCOPE context. The procedures `asquares` and `repeated_squaring` are essentially different The first has one parameter, a list of equations, while the latter accepts an arbitrary number of such lists as actual parameters. The `psopfn` indicator value is `repeated_squaringeval`, the name of the function, which is actually introduced. `asquares` is of `NORMAL`-type and applicable in both algebraic and symbolic mode.

```
OPERATOR square$

sq_rule:={square(~u) => u^2}$

ALGEBRAIC PROCEDURE asquare u;
 square(u) WHERE sq_rule$
```

```
SYMBOLIC PROCEDURE rsquare u;
 reval asquare aeval u$

SYMBOLIC PROCEDURE asquares u;
 append(list('list),
        FOREACH el IN cdr u COLLECT list('equal,cadr el,rsquare caddr el))$

SYMBOLIC OPERATOR asquares$

SYMBOLIC PROCEDURE repeated_squaringeval u;
BEGIN SCALAR res; INTEGER j;
 j=0;
 FOREACH el IN u DO
  << j:=j+1; el:=asquares el;
     FOR k:=2:j DO el:=asquares el;
     res:= IF j=1 THEN el ELSE append(res,cdr el)
  >>;
 RETURN res
END$

LISP( put('repeated_squaring,'psopfn,'repeated_squaringeval))$

% ---
% Examples of the use of asquares and repeated_squaring.
% Although the psopfn-mechanism can be easily avoided,
% it is used for illustrative purposes here.
% ---

OFF EXP$

asquare sin(x);

      2
sin(x)

LISP(rsquare('(sin x)));

(expt (sin x) 2)

asq:=asquares {s1=a+b,s2=(a+b)^2,s3=(a+b)^3};

                  2           4           6
asq := {s1=(a + b) ,s2=(a + b) ,s3=(a + b) }
```

```
repeated_squaring({s1=a+b,s2=(a+b)^2},{s3=(a+b)^3,s4=(a+b)^4},
                           {s5=(a+b)^5,s6=(a+b)^6});


          2
{s1=(a + b) ,

          4
 s2=(a + b) ,

          12
 s3=(a + b)  ,

          16
 s4=(a + b)  ,

          40
 s5=(a + b)  ,

          48
 s6=(a + b)  }

% ---
% The "ALGOPT asquares ...;" application is similar to the "ALGOPT asq;"
% instruction.
% ---

ALGOPT(asquares {s1=a+b,s2=(a+b)^2,s3=(a+b)^3},t);

                2      2
{t2=a + b,s1=t2 ,s2=s1 ,s3=s1*s2}

ALGOPT asq;
```

```
               2      2
{g6=a + b,s1=g6 ,s2=s1 ,s3=s1*s2}

% ---
% The OPTIMIZE variant is now applied on a repeated_squaring application.
% ---

OPTIMIZE repeated_squaring({s1=a+b,s2=(a+b)^2},{s3=(a+b)^3,s4=(a+b)^4},
                           {s5=(a+b)^5,s6=(a+b)^6}) INAME t;
t5 := a + b
s1 := t5*t5
s2 := s1*s1
t12 := s2*s2
s3 := s2*t12
s4 := s2*s3
s5 := t12*t12*t12*s4
s6 := t12*s5
```

□

4.2 Structure Recognition: `GSTRUCTR` and `ALGSTRUCTR`

The `structr` command in REDUCE 3.6 (see the manual, section 8.3.8) can be used to display the skeletal structure of its evaluated argument, a single expression. After setting `ON SAVESTRUCTR` a `structr` command will return a list, whose first element is a presentation for the expression and subsequent elements are the subexpression relations.
A special SCOPE feature provides an extended display facility, called `GSTRUCTR`. The syntax of this generalized command is:

<`REDUCE`_command>	::=	··· |
		`GSTRUCTR` <stat_group> [`NAME` <cse_prefix>]
<stat_group>	::=	≪ <stat_list> ≫
<stat_list>	::=	<gstat> [; <stat_list>]
<gstat>	::=	<name> := < expression> | <matrix_id>

The stat_group consists of one assignment statement or a group of such statements. Application of a `GSTRUCTR` command provides a display of the structure of the whole set of assignments. Such an assignment can be replaced by a matrix reference. That leads to the display of all the non-zero entries of the referenced matrix as well. The `NAME` part is optional. The cse-name mechanism is applied in the usual way.

The equivalent of a possible `ON SAVESTRUCTR` setting is provided in the form of a `PSOPFN`-type function, called `ALGSTRUCTR`. Its syntax is:

<function_application>	::=	`ALGSTRUCTR` (<arg_list> [, <cse_prefix>])
<arg_list>	::=	<arg_list_name> | {<arg_seq>}
<arg_seq>	::=	<arg>[,<arg_seq>]
<arg>	::=	<matrix_id> | <name>=<expression>
<arg_list_name>	::=	<id>

The result is presented in the form of an algebraic mode list.

Earlier SCOPE-versions allowed to use a `GSTRUCTR` command as (part of an) actual parameter for an `OPTIMIZE` command. This facility is not longer supported. In stead, an `ALGSTRUCTR` application can now be used as (part of an) actual parameter in both an `OPTIMIZE` command or an `ALGOPT` application. We now illustrate these features in:

Example 15

The script hardly requires explanation. However, observe that `v1`, `v3`, `v4`, `v6` and `v7` occur only once in the result of the `GSTRUCTR` application. When this application is used as actual parameter for an `OPTIMIZE` command these redundancies are removed before the actual optimization process starts. Likewise, an `ALGSTRUCTR` application only leads to identification of repeatedly occuring sub-structures in its input. `ALGSTRUCTR`, `ALGHORNER`, see the next subsection, and `ALGOPT` all apply the same output production strategy, i.e. it might be necessary to restore the previous algebraic mode status by applying the function `RESTOREALL`.

```
OFF EXP,PERIOD$

MATRIX a(2,2);

a:=mat((x+y+z,x*y),((x+y)*x*y,(x+2*y+3)^3-x));

     [ x + y + z          x*y        ]
     [                               ]
a := [                           3   ]
     [(x + y)*x*y  (x + 2*y + 3)  - x]

GSTRUCTR <<a;b:=(x+y)^2;c:=(x+y)*(y+z);d:=(x+2*y)*(y+z)*(z+x)^2>> NAME v$
```

```
a(1,1) := v1
a(1,2) := x*y
a(2,1) := v2*x*y
a(2,2) := v4
       2
b := v2
c := v2*v5
          2
d := v6*v7 *v5

   where

      v7 := x + z
      v6 := x + 2*y
      v5 := y + z
              3
      v4 := v3  - x
      v3 := x + 2*y + 3
      v2 := x + y
      v1 := x + y + z

ALGSTRUCTR({a,b=(x+y)^2,c=(x+y)*(y+z),d=(x+2*y)*(y+z)*(z+x)^2},v);

{a(1,1)=x + y + z,

 a(1,2)=x*y,

 v2=x + y,

 a(2,1)=v2*x*y,

                     3
 a(2,2)=(x + 2*y + 3)  - x,

     2
 b=v2 ,

 v5=y + z,

 c=v2*v5,

                    2
 d=(x + 2*y)*(x + z) *v5}
```

```
RESTORABLES;

{a}

ARESTORE a$

alst:=
 ALGOPT(ALGSTRUCTR({a,b=(x+y)^2,c=(x+y)*(y+z),d=(x+2*y)*(y+z)*(z+x)^2},v),s);

*** a declared operator

alst := {s5=x + z,

         a(1,1)=s5 + y,

         a(1,2)=x*y,

         v2=x + y,

         a(2,1)=a(1,2)*v2,

         s6=x + 2*y,

         s4=s6 + 3,

                  3
         a(2,2)=s4  - x,

             2
         b=v2 ,

         v5=y + z,

         c=v2*v5,

             2
         d=s5 *s6*v5}

% ---
% The above delivered warning is caused by the decloupling of a and its
% status as matrix. Therefore a(1,2) can function in the rhs of a(2,1).
% After an ARESTORE instruction a can restart its life as matrix_id.
% ---
```

```
a;

a

ARESTORE a$

a;

[ x + y + z          x*y        ]
[                               ]
[                          3    ]
[(x + y)*x*y  (x + 2*y + 3)  - x]
```

□

4.3 Horner-rules: GHORNER and ALGHORNER

Horner-rule based expression modification is a SCOPE facility, called GHORNER. The syntax of the command is similar to the GSTRUCTR syntax:

<REDUCE_command> ::= ··· |
GHORNER <stat_group> [VORDER <id_seq>];

The VORDER part is optional. Application of a (generalized) Horner-rule assumes an identifier ordering. The syntax of the identifier sequence is:

<id_seq> ::= <id>[,<id_seq>].

We assume the rhs's in the stat_group to be polynomials in the identifiers, partly or completely given in the id_seq. The left-to-right ordering of this sequence replaces the existing system identifier ordering. Identifiers, omitted from the vorder sequence have a lower preference and follow the existing system ordering. The rewritten rhs's are presented as a side-effect. FORTRAN notation is of course permitted. It is simply an extended print facility.

The PSOPFN-type variant of the GHORNER command is called ALGHORNER. Its syntax is:

<function_application> ::= ··· |
ALGHORNER (<arg_list> [,{<id_seq>}])

The syntax for the arg_list can be found in subsection 4.2. The result is presented in the form of an algebraic mode list. An ALGHORNER application can be used as (part of an) actual parameter of either an OPTIMIZE command or an `ALGOPT` application.

Example 16

We illustrate the Horner-facilities by rewriting the expression of example 2, before optimizing it. Observe that application of `ALGHORNER` in the default algebraic mode setting is useless. Due to the algebraic mode regime the rewritten expression is expanded again. We also show some Taylor-series remodelling.

```
ON EXP$

z:=a^2*b^2+10*a^2*m^6+a^2*m^2+2*a*b*m^4+2*b^2*m^6+b^2*m^2;

      2  2       2  6    2  2           4      2  6    2  2
z := a *b  + 10*a *m  + a *m  + 2*a*b*m  + 2*b *m  + b *m

GHORNER z:=z VORDER a;

         2  6    2  2            4       2       6    2
z := (2*b *m  + b *m ) + a*(2*b*m  + a*(b  + 10*m  + m ))

GHORNER z:=z VORDER b;

          2  6    2  2            4       2      6    2
z := (10*a *m  + a *m ) + b*(2*a*m  + b*(a  + 2*m  + m ))

hlst:={z=z}$

ALGHORNER(hlst,{a,b,m});

    2  2       2  6    2  2           4      2  6    2  2
{z=a *b  + 10*a *m  + a *m  + 2*a*b*m  + 2*b *m  + b *m }

OPTIMIZE ALGHORNER(hlst,{a,b,m}) INAME s;

s1 := m*m
s0 := s1*s1
s2 := b*b
s4 := 2*s0
z := a*(a*(s2 + s1*(10*s0 + 1)) + s4*b) + s2*s1*(s4 + 1)

OPTIMIZE ALGHORNER(hlst,{b,m}) INAME s;

s2 := m*m
```

```
s0 := s2*s2
s1 := a*a
s4 := 2*s0
z := b*(b*(s1 + s2*(s4 + 1)) + s4*a) + s2*(s1 + 10*s1*s0)
```

```
% Hornering Taylor-series:

PROCEDURE taylor(fx,x,x0,n);
sub(x=x0,fx)+(FOR k:=1:n SUM(sub(x=x0,df(fx,x,k))*(x-x0)^k/factorial(k)))$

hlst2:={f1=taylor(e^x,x,0,4),f2=taylor(cos x,x,0,6)};

               4      3       2
              x  + 4*x  + 12*x  + 24*x + 24
hlst2 := {f1=-------------------------------,
                           24

                   6       4        2
               - x  + 30*x  - 360*x  + 720
          f2=------------------------------}
                          720

OPTIMIZE ALGHORNER(hlst2,{x});


       24 + x*(24 + x*(12 + x*(4 + x)))
f1 := ----------------------------------
                      24
g7 := x*x
       720 + g7*(g7*(30 - g7) - 360)
f2 := -------------------------------
                     720

ON ROUNDED$


hlst2:=hlst2;

                              4                    3        2
hlst2 := {f1=0.0416666666667*x  + 0.166666666667*x  + 0.5*x  + x + 1,

                                  6                     4        2
          f2= - 0.00138888888889*x  + 0.0416666666667*x  - 0.5*x  + 1

          }

OPTIMIZE ALGHORNER(hlst2,{x});
```

```
f1 := 1 + x*(1 + x*(0.5 + x*(0.0416666666667*x + 0.166666666667)))
g9 := x*x
f2 := 1 + g9*(g9*(0.0416666666667 - 0.00138888888889*g9) - 0.5)
```

□

5 File Management and Optimization Strategies

Both the `OPTIMIZE` command and the `ALGOPT` function accept input from file(s). Obviously, this input ought to obey the usual syntactical rules, as introduced in the previous (sub)sections.

The `OPTIMIZE` command is designed as a syntactical extension of REDUCE itself, i.e. the meaning of its actual parameters is understood from the token-context in the command. However, an `ALGOPT` application requires one, two or three actual parameters without additional provisions or conditions. The `ALGOPT` facility is added to provide a simple, user friendly, algebraic mode tool. Therefore -in contrast with the `OPTIMIZE` command- it does not allow to direct output to a file; the default REDUCE features for dealing with output files can be applied. The previously given syntax requires some extensions:

<SCOPE_application> ::= <`OPTIMIZE` command> | <`ALGOPT` application>
<`OPTIMIZE` command> ::=
  `OPTIMIZE` <object_seq> [`IN` <file_id_seq>] [`OUT` <file_id>] [`INAME` <cse_prefix>] |
  `OPTIMIZE` [<object_seq>] `IN` <file_id_seq> [`OUT` <file_id>] [`INAME` <cse_prefix>]
<`ALGOPT` application> ::=
  `ALGOPT`(<a_object_list>[,<string_id_list>][,<cse_prefix>]) |
  `ALGOPT`([<a_object_list>,]<string_id_list>[,<cse_prefix>])

The different variations for the object_seq and the a_object_list and the meaning of cse_prefix are introduced in the subsections 3.1 and 3.2. The syntax of the file handling features is:

<file_id_seq>	::=	<file_id> [,<file_id_seq>]
<file_id>	::=	<id> | <string_id>
<string_id_list>	::=	<string_id> | {<string_id_seq>}
<string_id_seq>	::=	<string_id> [,<string_id_seq>]
<string_id>	::=	`"`<id>`"` | `"`<id> . <f_extension>`"`

The differences in input-file management are introduced for practical reasons. As stated above, the `ALGOPT` function can have up to three arguments. To be able to distinguish the optional second argument from the first and the last requires file-names to be given in the form of strings. The `OPTIMIZE` command follows the ordinary REDUCE rules for file names.

File management can be used as a tool for input partioning. If $m > 1$ then $N^m > \sum_{i=1}^{k} n^k{}_i$ for positive integers N and n_i , such that $N = \sum_{i=1}^{k} n_i$. In view of the time-complexity of the optimization algorithm, it may be worth the effort to partition SCOPE input of size N in k partitions, of sizes $n_i,\ i = 1, ..., k$. We can start optimizing the contents of file fi.1, containing the initial n_1-sized piece of code, and store the result of this operation in file fo.1. Consecutive steps provide an optimization of the combined contents of the files fo.i and fi.(i+1), i=1,..., k-1. During this iterative process, or during variations of this strategy, it is better not to perform a finishing touch. The switch `AGAIN`, which is normally `OFF`, can be used, when set `ON`, to avoid this. The switch serves an additional purpose. When switched `ON` storage of partly optimized code in a file will include all relevant information, needed to restore the required status of system generated sub-expression names.

We illustrate SCOPE's file management facilities with example.

Example 17

We assume to have three files, called f1, f2 and f3. Each file contains only one assignment. We simply show different variations of the use of these files. With `ON INPUTC` the contents of the files is made visible.

```
ON INPUTC$

OPTIMIZE IN f1,f2,f3 INAME s;


                                    2
                 2        (x + y)                8        2      2
        2*(sin(x)  - cos(e        ) + 3*cos(x)) *(x + y)  + 4*y  + 4*y
e1 := ---------------------------------------------------------------
                                     3*x + 2*y


                                   2
                2        (x + y)               2       3
e2 := (4*(sin(x)  - cos(e        ) + 2*cos(x)) *(x + y)

              2                   2
        + (4*x  - 4*y) - 6*x)/(8*x  + 3*y - 2*x)


                            2
                   (x + y)                     2           2
       4*sin(cos(e        )) + sin(x + y) + (4*x  - x + 2*y)
e3 := -------------------------------------------------------
```

```
                    3*y + f(x,g( - cos(x)))

s3 := sin(x)
s20 := x + y
s6 := s20*s20
           s6
s4 := cos(e  )
s8 := cos(x)
s31 := s3*s3 - s4
s2 := s31 + 3*s8
s44 := s2*s2
s43 := s44*s44
s36 := 4*y
s34 := 2*y
s10 := s34 + 3*x
       s36 + s36*y + 2*s6*s43*s43
e1 := ----------------------------
                  s10
s13 := s31 + 2*s8
s33 := 4*x*x
s30 := s33 - x
s35 := 3*y
```

```
       s33 - 2*s10 + 4*s6*s20*s13*s13
e2 := -------------------------------
                s35 + 2*s30
s21 := s34 + s30
       4*sin(s4) + sin(s20) + s21*s21
e3 := -------------------------------
            s35 + f(x,g( - s8))
```

We repeat the same process. However, this time we apply input partitioning. The switch `AGAIN` is turned `ON`. Output is redirected to the output file `fo.1` in an `OFF NAT` fashion and ended with the required `;end;` closure, thus made ready for re-use during a next step. The default mode of operation is `OFF AGAIN` and `ON NAT`. If the switch `NAT` is turned `OFF` file output is automatically ended by `;end;`.

Due to the `ON INPUTC` effect we can also observe that the identifiers `gsym` and `cses` are apparently used to store relevant information about cse names.

```
ON AGAIN,INPUTC$

OPTIMIZE IN f1 OUT "fo.1" INAME s$


                                 2
                2         (x + y)              8        2      2
       2*(sin(x)  - cos(e        ) + 3*cos(x)) *(x + y)  + 4*y  + 4*y
e1 := ----------------------------------------------------------------
                                 3*x + 2*y


OPTIMIZE IN "fo.1",f2 OUT "fo.2" INAME t$

gsym := g0001
cses := s6
            2
s6 := (x + y)
                2                           2        s6  8
       4*y + 4*y  + 2*s6*(3*cos(x) + sin(x)  - cos(e  ))
e1 := ----------------------------------------------------
                           3*x + 2*y
                                 2
                2         (x + y)              2        3
e2 := (4*(sin(x)  - cos(e        ) + 2*cos(x)) *(x + y)
             2                   2
     + (4*x  - 4*y) - 6*x)/(8*x  + 3*y - 2*x)

OFF AGAIN$

ALGOPT({"fo.2","f3"},u);

gsym := g0002

cses := t23 + t11 + t26 + t7 + t17 + t19

t19 := x + y

         2
t17 := t19

t7 := cos(x)

             2         t17
t26 := sin(x)  - cos(e    )

t11 := 3*x + 2*y

               2                 8
       4*y + 4*y  + 2*(t26 + 3*t7) *t17
e1 := ----------------------------------
                      t11

        2
t23 := x

                                          2
       4*t23 - 2*t11 + 4*t19*(t26 + 2*t7) *t17
e2 := -----------------------------------------
                  8*t23 - 2*x + 3*y

                          2
                  (x + y)                        2           2
       4*sin(cos(e        )) + sin(x + y) + (4*x  - x + 2*y)
e3 := ---------------------------------------------------------
                      3*y + f(x,g( - cos(x)))
```

```
*** f declared operator

*** g declared operator

{u23=x + y,

        2
 u20=u23 ,

 t7=cos(x),

 u5=sin(x),

         u20
 u6=cos(e   ),
       2
 t26=u5  - u6,

 u33=2*y,

 t11=u33 + 3*x,

 u10=t26 + 3*t7,

        2
 u46=u10 ,

        2
 u45=u46 ,

 u35=4*y,

              2
     2*u20*u45  + u35*y + u35
 e1=--------------------------,
               t11

      2
 t23=x ,

 u13=t26 + 2*t7,

 u36=4*t23,

 u31=u36 - x,

 u34=3*y,

                     2
      - 2*t11 + 4*u13 *u20*u23 + u36
 e2=---------------------------------,
               2*u31 + u34

 u24=u31 + u33,

                               2
     sin(u23) + 4*sin(u6) + u24
 e3=-----------------------------}
         f(x,g( - t7)) + u34
```

Observe that the initial characters of the sub-expression names indicate their moment of generation. We used `f` and `g` as operators. Therefore, a warning was produced ahead of the `ALGOPT` output. Since an `OPTIMIZE` command produces output as a side-effect these warnings were not given earlier.

□

6 Generation of Declarations

GENTRAN's `DECLARE` statement can be used as an optional extension of the `OPTIMIZE` command, and as ilustrated in example 18. The syntax of such an extension is in accordance with the GENTRAN rules:

```
<OPTIMIZE command>  ::=
    OPTIMIZE <object_seq> [IN <file_id_seq>] [OUT <file_id>]
             [INAME <cse_prefix>] [DECLARE <declaration_group>] |
    OPTIMIZE [<object_seq>] IN <file_id_seq> [OUT <file_id>]
             [INAME <cse_prefix>] [DECLARE <declaration_group>]
```

The syntax of the declaration_group is:

<declaration_group>	::=	<declaration> | ≪ <declaration_list> ≫
<declaration_list>	::=	<declaration>[; <declaration_list>]
<declaration>	::=	<range_list>: `IMPLICIT` <type> | <id_list>:<type>
<range_list>	::=	<range>[,<range_list>]
<range>	::=	<id> | <id> − <id>
<id_list>	::=	<id>[,<id_list>]
<type>	::=	`integer` | `real` | `complex` | `real*8` | `complex*16`

The symbol table features of GENTRAN are used. During the subtask R (see subsection 2.3) of an `OPTIMIZE` command evaluation, all typing information is installed in the symbol table. Once optimization is ready all relevant information for completing the declarations ought to be known, i.e. the contents of the symbol table and the result of the optimization operations, collected in prefix form in a list, called `prefixlist`. This `prefixlist` is employed do decide which not yet typed identifiers and system selected cse names have to be entered in the symbol table. We make use of earlier provided information, delivered via the `DECLARE` option, (sub)expression structure and the normal hierarchy in data types. The strategy to achieve this form of dynamic typing is based on chapter 6 of [?]. Once the table is completed a list of declarations is produced and precedes the other SCOPE output. SCOPE output is by default given in REDUCE notation. Therefore such lists of declarations are also given in REDUCE text. Incomplete initial typing information can lead to overtyping after optimization, such as `complex` in stead of `real`, for instance. It can therefore lead to erroneous results and even to an error message. A safe procedure is to use the `DECLARE` option of the `OPTIMIZE` command for typing all identifiers, occuring in the input set E_0

Alternative output can be obtained via an application of the function `OPTLANG`. This function accepts one argument from the set {`fortran`, `c`, `ratfor`, `pascal`[1], `f90`, `nil`}. The `fortran`(77) choice can also be made by turning `ON` the switch `FORT`. The `nil` option is necessary if one wants to switch back to the usual REDUCE output. not yet generally available. The output modules of GENTRAN are used for producing formatted code in the user selected target language. The `f90` option, for the production of `fortran90` code, is not yet provided by the standard GENTRAN version [?].

Especially the above given syntax rules for typing require some additional explanation:

- The corresponding types in Fortran are `integer`, `real`, `complex`, `double precision` and `complex*16`.
- The GENTRAN switch `DOUBLE` is automatically turned `ON`, when a type `real*8` or type `complex*16` is introduced in a `DECLARE` option. The same mode of operation is introduced when floating point numbers appear in SCOPE input. Fixed floats do not produce this side effect.
- When generating `fortran` code we have to be aware of a possibly existing statement length limitation. If one is afraid that a declaration statement will become too long, for instance due to a huge number, dynamically added cse-names, it may be better to use `IMPLICIT` typing.
- C neither supports `IMPLICIT` types nor has the types `complex` and `complex*16`. The remaining types are denoted by `int`, `float` and `double`, respectively.
- Array and/or matrix definitions are also considered to be id's in id_list's in declarations. However, we have to be aware of the instantaneous replacement of array- and/or matrix entries, when expressions are simplified. Therefore, we have to use operators, functioning as array and/or matrix names in code we want to optimize. We return to this question in the sections 7 and 8.

When the `ON/OFF AGAIN` strategy is applied we have to be aware of the above outlined declaration strategy. The last `OPTIMIZE` command, executed

[1]The `pascal` module of GENTRAN is not error free. Especially the template file features do not function correctly.

directly after choosing `OFF AGAIN`, has to be extended with the `DECLARE` option.

Array and/or matrix names only occur in literally parsed information. In all other situations we have to make use of REDUCE `operators`. Normally, function applications inside SCOPE input are instantaneously replaced by newly selected cse names after putting them in the function table. Usually array and/or matrix entries are considered to be function applications. However, when due to a `DECLARE` option array and/or matrix names are known via the contents of the symbol table, such entries are substituted back before SCOPE produces output.

Example 18

A simple `OPTIMIZE` command, extended with a `DECLARE` option, is executed for the various output options of GENTRAN, including the `f90` alternative.

```
OPTLANG fortran$
OPTIMIZE x(i+1):=a(i+1,i-1)+b(i),y(i-1):=a(i-1,i+1)-b(i)
INAME s
DECLARE << a(4,4),x(4),y(5):real; b(5):integer>>$

      INTEGER B(5),I,S10,S9
      REAL A(4,4),X(4),Y(5)
      S10=I+1
      S9=I-1
      X(S10)=A(S10,S9)+B(I)
      Y(S9)=A(S9,S10)-B(i)

OPTLANG ratfor$
OPTIMIZE x(i+1):=a(i+1,i-1)+b(i),y(i-1):=a(i-1,i+1)-b(i)
INAME s
DECLARE << a(4,4),x(4),y(5):real; b(5):integer>>$

integer b(5),i,s10,s9
real a(4,4),x(4),y(5)
{
    s10=i+1
    s9=i-1
    x(s10)=a(s10,s9)+b(i)
    y(s9)=a(s9,s10)-b(i)
}

OPTLANG c$
```

```
OPTIMIZE x(i+1):=a(i+1,i-1)+b(i),y(i-1):=a(i-1,i+1)-b(i)
INAME s
DECLARE << a(4,4),x(4),y(5):real; b(5):integer>>$

int b[6],i,s10,s9;
float a[5][5],x[5],y[6];
{
    s10=i+1;
    s9=i-1;
    x[s10]=a[s10][s9]+b[i];
    y[s9]=a[s9][s10]-b[i];
}

OPTLANG pascal$
OPTIMIZE x(i+1):=a(i+1,i-1)+b(i),y(i-1):=a(i-1,i+1)-b(i)
INAME s
DECLARE << a(4,4),x(4),y(5):real; b(5):integer>>$

var
    s9,s10,i: integer;
    b: array[0..5] of integer;
    y: array[0..5] of real;
    x: array[0..4] of real;
    a: array[0..4,0..4] of real;
begin
    s10:=i+1;
    s9:=i-1;
    x[s10]:=a[s10,s9]+b[i];
    y[s9]:=a[s9,s10]-b[i]
end;

OPTLANG nil$
OPTIMIZE x(i+1):=a(i+1,i-1)+b(i),y(i-1):=a(i-1,i+1)-b(i)
INAME s
DECLARE << a(4,4),x(4),y(5):real; b(5):integer>>$

integer b(5),i,s10,s9
real a(4,4),x(4),y(5)

s10 := i + 1
s9 := i - 1
x(s10) := a(s10,s9) + b(i)
y(s9) := a(s9,s10) - b(i)
```

```
OPTLANG fortran$
OPTIMIZE x(i+1):=a(i+1,i-1)+b(i),y(i-1):=a(i-1,i+1)-b(i)
INAME s
DECLARE << a(4,4),x(4),y(5):real*8; b(5):integer>>$

      INTEGER B(5),I,S10,S9
      DOUBLE PRECISION A(4,4),X(4),Y(5)
      S10=I+1
      S9=I-1
      X(S10)=A(S10,S9)+B(I)
      Y(S9)=A(S9,S10)-B(I)

OPTIMIZE x(i+1):=a(i+1,i-1)+b(i),y(i-1):=a(i-1,i+1)-b(i)
INAME s
DECLARE << x(4),y(5):real; b(5):complex>>$

***** Type error:
real x(4),y(5)
complex b(5)
(integer all) s9
integer s5,i
real := complex(all)
***** Wrong typing
Cont? (Y or N)
```

We can restart REDUCEand rerun the example with the `Fortran90` version of SCOPE. It results in:

```
LOAD_PACKAGE scope90$
OPTLANG f90$
OPTIMIZE x(i+1):=a(i+1,i-1)+b(i),y(i-1):=a(i-1,i+1)-b(i)
INAME s
DECLARE << a(4,4),x(4),y(5):real; b(5):integer>>$

      REAL,DIMENSION(4,4)::A
      INTEGER,DIMENSION(5)::B
      INTEGER::I,S10,S9
      REAL,DIMENSION(4)::x
      REAL,DIMENSION(5)::y
      S10=I+1
      S9=I-1
      X(S10)=A(S10,S9)+B(I)
      Y(S9)=A(S9,S10)-B(I)
```

□

6.1 Coefficient Arithmetic and Precision Handling

REDUCE knows a variety of coefficient domains, as presented in subsection 9.11 of the REDUCE 3.6 manual [**?**], entitled *Polynomial Coefficient Arithmetic*. As stated in subsection 3.1 SCOPE supports integer and real coefficients. By turning `ON` the switch `ROUNDED` we introduce float arithmetic for coefficients. The operator `PRECISION` can be applied to change the default, machine dependent *precision*. Internally, REDUCE uses floating point numbers up to the precison supported by the underlying machine hardware, and so-called *bigfloats* for higher precision. The internal precision is two decimals greater than the exernal precision to guard against roundoff inaccuracies. Rounded numbers are normally printed to the specified precision. If the user wishes to print such numbers with less precision, the printing precision can be set by the command `PRINT_PRECISION`. If a case arises where use of the machine arithmetic leads to problems, a user can force REDUCE to use the bigfloat representation by turning `ON` the switch `ROUNDBF`, which is normally `OFF`. GENTRAN, and thus SCOPE as well, support bigfloat notation. However the precision is a responsibility of the user. A possibility is to use the `PRINT_PRECISION` command, both for algebraic mode output and for output in a selected target language, like `fortran77`. SCOPE uses the given precision for selecting cse's. Although complex arithmetic is not supported in SCOPE, a simple alternative is provided. When using float arithmetic in REDUCE the protected name `I` can be used to denote $\sqrt{-1}$. If the `I` is included in a declaration list as an identifier of type `complex(!*)`, its assumed value is automatically put ahead of the resulting optimized code. We illustrate the different possibilities in example 19. Comment is included.

Example 19

```
OPTLANG fortran$
ON ROUNDED, DOUBLE$

% ---
% We start with precision 6. The returned value is the internal
% precision supported by the underlying machine hardware.
% ---

PRECISION 6;

12
```

```
OPTIMIZE x1:= 2        *a + 10         * b,
         x2:= 2.00001  *a + 10         * b,
         x3:= 2        *a + 10.00001   * b,
         x4:= 6        *a + 30         * b,
         x5:= 2.0000001*a + 10.000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,X1,X2,X3,X4,X5
      S1=10*B
      X1=S1+2*A
      X2=S1+2.00001D0*A
      X3=X1
      X4=3*X1
      X5=X1

% ---
% Explanation: X1 is a cse of X3, X4 and X5, but not of X2, because
% the coefficient 2.00001 is given in 6 decimal digits.
% Increase in precision will show this.
% ---

PRECISION 7$

OPTIMIZE x1:= 2        *a + 10         * b,
         x2:= 2.00001  *a + 10         * b,
         x3:= 2        *a + 10.00001   * b,
         x4:= 6        *a + 30         * b,
         x5:= 2.0000001*a + 10.000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.00001D0*A
      X3=S1+1.000001D1*B
      X4=3*X1
      X5=X1

PRECISION 8$
```

```
OPTIMIZE x1:= 2        *a + 10         * b,
         x2:= 2.00001  *a + 10         * b,
         x3:= 2        *a + 10.00001   * b,
         x4:= 6        *a + 30         * b,
         x5:= 2.0000001*a + 10.000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.00001D0*A
      X3=S1+1.000001D1*B
      X4=3*X1
      X5=2.0000001D0*A+1.0000001D1*B

% ---
% All rhs's were taken literally. Let us now increase precision and
% simplify the rhs's before optimization. It is in fact a repetition
% of the examples above, this time with a larger precision.
% ---

PRECISION 20$

OPTIMIZE x1:=:2                     *a + 10                      * b,
         x2:=:2.0000000000000000001  *a + 10                      * b,
         x3:=:2                     *a + 10.0000000000000000001   * b,
         x4:=:6                     *a + 30                      * b,
         x5:=:2.00000000000000000001*a + 10.00000000000000000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.0000000000000000001D0*A
      X3=S1+1.0D1*B
      X4=3*X1
      X5=1.0D0*X1

PRECISION 21$
```

```
OPTIMIZE x1:=:2                     *a + 10                     * b,
         x2:=:2.0000000000000000001  *a + 10                     * b,
         x3:=:2                     *a + 10.0000000000000000001   * b,
         x4:=:6                     *a + 30                     * b,
         x5:=:2.00000000000000000001*a + 10.00000000000000000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.0000000000000000001D0*A
      X3=S1+1.00000000000000000001D1*B
      X4=3*X1
      X5=2.0D0*A+1.0D1*B

PRECISION 22$

OPTIMIZE x1:=:2                     *a + 10                     * b,
         x2:=:2.0000000000000000001  *a + 10                     * b,
         x3:=:2                     *a + 10.0000000000000000001   * b,
         x4:=:6                     *a + 30                     * b,
         x5:=:2.00000000000000000001*a + 10.00000000000000000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.0000000000000000001D0*A
      X3=S1+1.00000000000000000001D1*B
      X4=3*X1
      X5=2.00000000000000000001D0*A+1.0D1*B

% ---
% However, we can observe some differences in both modes of operation, when
% selecting a precision around the precision supported by the undelying
% machine hardware. Then the switch ROUNDBF can better be turned ON.
% ---

OFF ROUNDBF$
PRECISION 12$
```

```
OPTIMIZE x1:= 2.00         *a + 10.00       * b,
         x2:= 2.00000000001*a + 10         * b,
         x3:= 2           *a + 10.000000001* b,
         x4:= 6           *a + 30         * b,
         x5:= 2.0000000000001*a + 10.0000000000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.00000000001D0*A
      X3=S1+1.0000000001D1*B
      X4=3*X1
      X5=X1


OPTIMIZE x1:=:2.00         *a + 10.00       * b,
         x2:=:2.00000000001*a + 10         * b,
         x3:=:2           *a + 10.000000001* b,
         x4:=:6           *a + 30         * b,
         x5:=:2.0000000000001*a + 10.0000000000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.0D0*A
      X3=S1+10.0D0*B
      X4=3*X1
      X5=X1

% ---
% Observe that simplification prior to optimization leads to internal
% roundings, which differ from the rounding used for literally taken
% coefficients. This difference disappeares with ON ROUNDBF.
% ---

ON ROUNDBF$
```

```
OPTIMIZE x1:= 2.00         *a + 10.00       * b,
         x2:= 2.00000000001*a + 10         * b,
         x3:= 2           *a + 10.000000001* b,
         x4:= 6           *a + 30         * b,
         x5:= 2.0000000000001*a + 10.0000000000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.00000000001D0*A
      X3=S1+1.0000000001D1*B
      X4=3*X1
      X5=X1

OPTIMIZE x1:=:2.00         *a + 10.00       * b,
         x2:=:2.00000000001*a + 10         * b,
         x3:=:2           *a + 10.000000001* b,
         x4:=:6           *a + 30         * b,
         x5:=:2.0000000000001*a + 10.0000000000001 * b
INAME s
DECLARE <<x1,x2,x3,x4,x5,a,b: real>>$

      DOUBLE PRECISION A,B,S1,S2,X1,X2,X3,X4,X5
      S1=2*A
      S2=10*B
      X1=S2+S1
      X2=S2+2.00000000001D0*A
      X3=S1+1.0000000001D1*B
      X4=3*X1
      X5=X1

% ---
% Complex arithmetic is not supported in SCOPE. However the Fortan equivalent
% of I, a protected name in REDUCE, is automatically created, ahead of the
% optimized code, whenever I is included in the declaration as a type complex
% or a type complex*16 identifier.
% ---

OPTIMIZE a:=b+c
INAME s
DECLARE <<a,b,i,c: complex>>;
```

```
      COMPLEX*16 B,I,C,A
      I=(0.0D0, 1.0D0)
      A=B+C

OFF DOUBLE$

OPTIMIZE a:=b+c
INAME s
DECLARE <<a,b,i,c: complex>>;

      COMPLEX B,I,C,A
      I=(0.0, 1.0)
      A=B+C
```

□

7 Dealing with Data Dependencies

SCOPE is designed to optimize blocks B of straight line code, i.e. sequences of n assignment statements S_i of the form $\lambda_i := \rho_i$, where $i = 1, \cdots, n$. If an identifier occurs in λ_i, it is said to be *defined* in S_i. All identifiers occuring in ρ_i are said to be *used* in S_i. The set DEF(S_i) is formed by the identifiers defining S_i, usually only one. The set USE(S_i) is formed by the identifiers, which are used in S_i. The relation DEF$(S_i) \in$ USE(S_j), for $1 \leq i < j \leq n$, is called a *flow dependency* and denoted by $S_i \rightarrow S_j$. The relation DEF$(S_i) \in$ USE(S_j), for $1 \leq j \leq i \leq n$ is called an *anti dependency* and denoted by $S_i \not\rightarrow S_j$. The *set of inputs* of B, denoted by $I(B)$, consists of identifiers, which are used in B, before being defined, if defined at all. The *set of outputs* of B, denoted by $O(B)$, consists of the set of all last definitions of identifiers, occuring in B. So a block of straight line code can be introduced as a triple $B = \{S, I, O\}$, where S stands for the sequence $S_1; S_2; \cdots; S_{n-1}; S_n$, and where I and O define the inputs and outputs, respectively. When optimizing source code defined by B, i.e. the sequence S, the intention is to mechanically produce an equivalent, but computationally less complex sequence, preserving the relation between inputs and outputs. Due to anti dependencies, i.e. redefinitions of the rules for computing identifier values or stepwise computing such values, $\mid O(B) \mid < n$ is possible. But that in turn implies that some of the used identifiers, although being literally identical, represent different values. Therefore, a mechanical search for cse's can only be maintained if these critical identifiers are adequately renamed internally before the optimization process itself is started. As long as the relation between $I(B)$ and $O(B)$ is preserved it is even allowed to partly maintain these additional names, when presenting the results of an optimization operation. Furthermore it is worth noting that *dead code* can be left out, when ever occuring. Such code can be introduced through redundant assignments. The SCOPE features for dealing with data dependencies are illustrated, using the following artificial piece of code:

$$\begin{array}{llll}
S_1 & : & a_{1,x+1} & := & g + h.r^f \\
S_2 & : & b_{y+1} & := & a_{1,2.x+1}.(g + h.r^f) \\
S_3 & : & c1 & := & h.r.a_{2,1+x}/g \\
S_4 & : & c2 & := & c1.a_{1,x+1} + sin(d) \\
S_5 & : & a_{1,x+1} & := & c1 + 2 \\
S_6 & : & d & := & b_{y+1}.a_{1,x+1} \\
S_7 & : & a_{1,1+2.x} & := & a_{1,x+1}.b_{y+1}.c/(d.g^2) \\
S_8 & : & b_{y+1} & := & a_{1.x+1} + b_{y+1} + sin(d) \\
S_9 & : & a_{1,x+1} & := & b_{y+1}.c + h/(g + sin(d)) \\
S_{10} & : & d & := & k.e + d.(a_{1,1+x} + 3) \\
S_{11} & : & e & := & d.(a_{1,1+x} + 3) + sin(d) \\
S_{12} & : & f & := & d.(a_{1,1+x} + 3) + sin(d) \\
S_{13} & : & g & := & d.(3 + a_{1,1+x}) + f
\end{array}$$

The different flow and anti dependencies can be vizualized in the following way:

$$\begin{array}{lll}
x & : & \{\lambda_1, \rho_2, \rho_3, \rho_4, \lambda_5, \rho_6, \lambda_7, \rho_7, \rho_8, \lambda_9, \rho_{10}, \rho_{11}, \rho_{12}, \rho_{13}\} \\
y & : & \{\lambda_2, \rho_6, \rho_7, \lambda_8, \rho_8, \rho_9\}
\end{array}$$

The identifiers, occuring in the piece of code given above, can be defined, can be used or can play both roles. Identifiers, used in one or more of the $\lambda_i, i = 1 \cdots n$ figure in subscript expressions. The set notation $\{ \cdots \}$ is used to explicitly describe USE sets. Since all DEF sets consist of one element only, we omitted the set notation for the DEF sets. This provides a simple tool to distinguish flow and anti dependencies:

$a_{1,x+1}$	:	$\lambda_1 \rightarrow \{\rho_3, \rho_4\} \not\rightarrow \lambda_5 \rightarrow \{\rho_6, \rho_7, \rho_8\} \not\rightarrow \lambda_9 \rightarrow \{\rho_{10}, \rho_{11}, \rho_{12}, \rho_{13}\}$
g	:	$\{\rho_1, \rho_2, \rho_3, \rho_7, \rho_9\} \not\rightarrow \lambda_{13}$
h	:	$\{\rho_1, \rho_2, \rho_3, \rho_9\}$
r	:	$\{\rho_1, \rho_2, \rho_3\}$
f	:	$\{\rho_1, \rho_2\} \not\rightarrow \lambda_{12} \rightarrow \rho_{13}$
b_{y+1}	:	$\lambda_2 \rightarrow \{\rho_6, \rho_7, \rho_8\} \not\rightarrow \lambda_8 \rightarrow \{\rho_9\}$
$a_{1,2.x+1}$	:	$\{\rho_2\} \not\rightarrow \lambda_7$
$c1$	:	$\lambda_3 \rightarrow \{\rho_4, \rho_5\}$
$a_{2,1+x}$	:	$\{\rho_3\}$
$c2$	:	λ_4
d	:	$\{\rho_4\} \not\rightarrow \lambda_6 \rightarrow \{\rho_7, \rho_8, \rho_9, \rho_{10}\} \not\rightarrow \lambda_{10} \rightarrow \{\rho_{11}, \rho_{12}, \rho_{13}\}$
c	:	$\{\rho_7, \rho_9\}$
k	:	$\{\rho_{10}\}$
e	:	$\{\rho_{10}\} \not\rightarrow \lambda_{12}$

We observe that

$$I(B) = \{x, y, g, h, r, f, a_{1,2.x+1}, a_{2,1+x}, d, c, k, e\},$$

$$O(B) = \{a_{1,x+1}, g, f, b_{y+1}, a_{1,2.x+1}, c1, c2, d, e\}$$

and thus, that

$$I(B) \cap O(B) \neq \emptyset.$$

The identifiers $a_{1,1+x}$ and d are redefined twice and the identifier b_{y+1} once. Only the input occurrences and the last output definitions need to be preserved.

We also observe that some of the identifiers are subscripted. In our example the subscript expressions are constructed with input identifiers only. More general situations are conceivable. The set of subscript expressions contains cse's. Since our optimization strategy is based on an all level approach expressions, like $1 + x$ and $y + 1$ are certainly discovered as cse's.

An `OPTIMIZE` command can be extended with a `DECLARE` option, indicating that both a and b are array names. Their subscript expressions are optimized. In addition, the a and b entries are considered to be array entries, and not as function applications. The latter will happen when the `DECLARE` option is omitted. Vector architectures make often use of machine specific optimizing compilers. Therefore it may be better not to optimize the subset of subscript expressions. We implemented some facilities to take such machine specific limitations into account. When turning `ON` the switch `VECTORC` the finishing touch is omitted and subscript expressions are not optimized. The function

    `VECTORCODE` <a_or_m_id_seq>

can be used to operate more selectively. The a_or_m_id_seq consists of one or more array and/or matrix names, separated by comma's. Only the actual parameters of `VECTORCODE` are assumed to be names of arrays. So only their subscript expressions are disregarded during an optimization process. We can undo the effect of the `VECTORCODE` setting with a command of the form:

    `VCLEAR` <a_or_m_id_(sub)seq>

The actual parameters are supposed to be taken from the sequence of actual parameters of its counterpart, the function `VECTORCODE`.

The different settings are now illustrated in:

Example 20

The above given block of code $B = \{S, I, O\}$ is optimized in five different situations. To start with we hand it over to SCOPE, using an `OPTIMIZE` command, without using the `DECLARE` option. We observe, looking at the results, that some renaming of non significant identifier definitions have been performed. The first occurrence of $a_{1,1+x}$ is replaced by `s34`, the second by `s4`. The first occurrence of b_{y+1} is replaced by `s3`, but the occurrences of d, a scalar identifier, are maintained.

Especially the role of the scalar d is quite interesting. The first definition of d is given in S_6 In S_7 d is used twice: explicitly in the denominator and in a hidden way in the numerator as well. The optimized version of S_7 shows a factor d/d. The reason is that d locally replaces an internally created cse name, substituted for ρ_6 and for part of the numerator of ρ_7. Like illustrated in example 13 a second SCOPE application can further simplify ρ_7. The scalar d is also used as argument of the *sin*-function in S_4, S_8, S_9, S_{11} and S_{12}. The d-values in S_4, in $\{S_8, S_9\}$ and in $\{S_{11}, S_{12}\}$ are all different, due to the redefinitions in S_6 and in S_{10}. Therefore we recognize two different cse's, containing $sin(d)$: `s24` and `e`. The role of $a_{1,x+1}$ is of course very similar, albeit less transparant, due to the renamings.

The second run showes that adding a `DECLARE` option does not influence the form of the output in this particular case. Besides the production of declarations, the result of both runs is identical. All relevant input and output names are preserved in both runs.

```
OPTIMIZE a(1,x+1)  := g+h*r^f,
         b(y+1)    := a(1,2*x+1)*(g+h*r^f),
         c1        := (h*r)/g*a(2,1+x),
         c2        := c1*a(1,x+1) + sin(d),
         a(1,x+1)  := c1^(5/2),
         d         := b(y+1)*a(1,x+1),
         a(1,1+2*x):= (a(1,x+1)*b(y+1)*c)/(d*g^2),
         b(y+1)    := a(1,1+x)+b(y+1) + sin(d),
         a(1,x+1)  := b(y+1)*c+h/(g + sin(d)),
         d         := k*e+d*(a(1,1+x)+3),
         e         := d*(a(1,1+x)+3) + sin(d),
         f         := d*(3+a(1,x+1)) + sin(d),
         g         := d*(3+a(1,x+1))+f
INAME s$


s0 := x + 1
        f
s34 := r *h + g
s2 := 1 + y
s6 := 2*x + 1
s3 := s34*a(1,s6)
              r*h
c1 := a(2,s0)*-----
                g
c2 := sin(d) + s34*c1
s4 := sqrt(c1)*c1*c1
d := s4*s3
             d*c
a(1,s6) := -------
            g*g*d
s24 := sin(d)
b(s2) := s4 + s3 + s24
              h
a(1,s0) := --------- + b(s2)*c
            g + s24
s29 := 3 + a(1,s0)
d := s29*d + e*k
s33 := s29*d
e := s33 + sin(d)
f := e
g := s33 + f

ON FORT$
```

```
OPTIMIZE a(1,x+1)  := g+h*r^f,
         b(y+1)    := a(1,2*x+1)*(g+h*r^f),
         c1        := (h*r)/g*a(2,1+x),
         c2        := c1*a(1,x+1) + sin(d),
         a(1,x+1)  := c1^(5/2),
         d         := b(y+1)*a(1,x+1),
         a(1,1+2*x):= (a(1,x+1)*b(y+1)*c)/(d*g^2),
         b(y+1)    := a(1,1+x)+b(y+1) + sin(d),
         a(1,x+1)  := b(y+1)*c+h/(g + sin(d)),
         d         := k*e+d*(a(1,1+x)+3),
         e         := d*(a(1,1+x)+3) + sin(d),
         f         := d*(3+a(1,x+1)) + sin(d),
         g         := d*(3+a(1,x+1))+f
INAME s
DECLARE <<a(5,5),b(7),c,c1,c2,d,f,g,h,r:real*8; x,y:integer>>$

      INTEGER X,Y,S0,S2,S6
      DOUBLE PRECISION C,H,R,S34,S3,C1,C2,S4,S24,B(7),A(5,5),S29,K,D,S33
     . ,E,F,g
      S0=X+1
      S34=R**F*H+G
      S2=1+Y
      S6=2*X+1
      S3=S34*A(1,S6)
      C1=A(2,S0)*((R*H)/G)
      C2=DSIN(D)+S34*C1
      S4=DSQRT(C1)*C1*C1
      D=S4*S3
      A(1,S6)=(D*C)/(G*G*D)
      S24=DSIN(D)
      B(S2)=S4+S3+S24
      A(1,S0)=H/(G+S24)+B(S2)*C
      S29=3+A(1,S0)
      D=S29*D+DEXP(1.0D0)*K
      S33=S29*D
      E=S33+DSIN(D)
      F=DEXP(1.0D0)
      G=S33+F

OFF FORT$
```

Observe the differences in the `f` and `F` assignments. When generating `fortran77` code all right hand side occurrences of `e` are automatically considered as appearances of the base of the natural logarithm and are translated accordingly.

The third run is influenced by turning ON the switch VECTORC. We observe that all array references are substituted back, without having replaced repeatedly occuring identical subscript expressions by internally selected cse names.

The fourth and the last run are governed by the functions VECTORCODE and VCLEAR, after having turned OFF the switch VECTORC.

```
ON VECTORC$

OPTIMIZE a(1,x+1)  := g+h*r^f,
         b(y+1)    := a(1,2*x+1)*(g+h*r^f),
         c1        := (h*r)/g*a(2,1+x),
         c2        := c1*a(1,x+1) + sin(d),
         a(1,x+1)  := c1^(5/2),
         d         := b(y+1)*a(1,x+1),
         a(1,1+2*x):= (a(1,x+1)*b(y+1)*c)/(d*g^2),
         b(y+1)    := a(1,1+x)+b(y+1) + sin(d),
         a(1,x+1)  := b(y+1)*c+h/(g + sin(d)),
         d         := k*e+d*(a(1,1+x)+3),
         e         := d*(a(1,1+x)+3) + sin(d),
         f         := d*(3+a(1,x+1)) + sin(d),
         g         := d*(3+a(1,x+1))+f
INAME s$

               f
a(1,x + 1) := r *h + g
b(y + 1) := a(1,x + 1)*a(1,2*x + 1)
                   r*h
c1 := a(2,x + 1)*-----
                    g
c2 := sin(d) + a(1,x + 1)*c1
                          2
a(1,x + 1) := sqrt(c1)*c1
d := a(1,x + 1)*b(y + 1)
```

```
                 d*c
a(1,1 + 2*x) := ------
                  2
                 g *d
s20 := sin(d)
b(y + 1) := a(1,x + 1) + b(y + 1) + s20
                  h
a(1,x + 1) := --------- + b(y + 1)*c
              g + s20
s25 := a(1,x + 1) + 3
d := s25*d + e*k
s28 := s25*d
e := s28 + sin(d)
f := e
g := s28 + f

OFF VECTORC$
VECTORCODE a,b$

OPTIMIZE a(1,x+1)  := g+h*r^f,
         b(y+1)    := a(1,2*x+1)*(g+h*r^f),
         c1        := (h*r)/g*a(2,1+x),
         c2        := c1*a(1,x+1) + sin(d),
         a(1,x+1)  := c1^(5/2),
         d         := b(y+1)*a(1,x+1),
         a(1,1+2*x):= (a(1,x+1)*b(y+1)*c)/(d*g^2),
         b(y+1)    := a(1,1+x)+b(y+1) + sin(d),
         a(1,x+1)  := b(y+1)*c+h/(g + sin(d)),
         d         := k*e+d*(a(1,1+x)+3),
         e         := d*(a(1,1+x)+3) + sin(d),
         f         := d*(3+a(1,x+1)) + sin(d),
         g         := d*(3+a(1,x+1))+f
INAME s$

              f
a(1,x + 1) := r *h + g
b(y + 1) := a(1,x + 1)*a(1,2*x + 1)
                  r*h
c1 := a(2,x + 1)*-----
                   g
c2 := sin(d) + a(1,x + 1)*c1
a(1,x + 1) := sqrt(c1)*c1*c1
d := a(1,x + 1)*b(y + 1)
                  d*c
```

```
a(1,1 + 2*x) := -------
                 g*g*d
s22 := sin(d)
b(y + 1) := a(1,x + 1) + b(y + 1) + s22
                  h
a(1,x + 1) := --------- + b(y + 1)*c
               g + s22
s27 := 3 + a(1,x + 1)
d := s27*d + e*k
s30 := s27*d
e := s30 + sin(d)
f := e
g := s30 + f


VCLEAR b$

OPTIMIZE a(1,x+1)  := g+h*r^f,
         b(y+1)    := a(1,2*x+1)*(g+h*r^f),
         c1        := (h*r)/g*a(2,1+x),
         c2        := c1*a(1,x+1) + sin(d),
         a(1,x+1)  := c1^(5/2),
         d         := b(y+1)*a(1,x+1),
         a(1,1+2*x):= (a(1,x+1)*b(y+1)*c)/(d*g^2),
         b(y+1)    := a(1,1+x)+b(y+1) + sin(d),
         a(1,x+1)  := b(y+1)*c+h/(g + sin(d)),
         d         := k*e+d*(a(1,1+x)+3),
         e         := d*(a(1,1+x)+3) + sin(d),
         f         := d*(3+a(1,x+1)) + sin(d),
         g         := d*(3+a(1,x+1))+f
INAME s$

               f
a(1,x + 1) := r *h + g
s1 := y + 1
s2 := a(1,x + 1)*a(1,1 + 2*x)
                  r*h
c1 := a(2,1 + x)*-----
                   g
c2 := sin(d) + a(1,x + 1)*c1
a(1,x + 1) := sqrt(c1)*c1*c1
d := a(1,x + 1)*s2
                  d*c
a(1,1 + 2*x) := -------
```

```
                 g*g*d
s23 := sin(d)
b(s1) := a(1,x + 1) + s2 + s23
                  h
a(1,x + 1) := --------- + b(s1)*c
               g + s23
s28 := 3 + a(1,x + 1)
d := s28*d + e*k
s31 := s28*d
e := s31 + sin(d)
f := e
g := s31 + f
```

□

8 A Combined Use of GENTRAN and SCOPE 1.5

As already stated in subsection 2.4 each GENTRAN command is evaluated separately, implying that the symbol table, required for the production of declarations, is fresh at the beginning of a GENTRAN command evaluation. Turning `ON` the switch `GENTRANOPT` leads to the optimization of the arithmetic code, defined in the GENTRAN command, obeying the new `GENTRANOPT` regime. In addition, we can observe that each separate GENTRAN command can produce its own declarations.

To increase flexibility we introduced facilities for making declarations, associated with a group of GENTRAN commands and for the optimization of the arithmetic in such a group as well. We implemented two function pairs of parameter-less functions:

(`DELAYDECS`, `MAKEDECS`)

and

(`DELAYOPTS`, `MAKEOPTS`).

Both pairs function as "brackets" around groups of statements. The `OPTS` pair can be used (repeatedly) inside a `DECS` pair. Both pairs achieve an alterned GENTRAN mode of operation. All GENTRAN productions between such a pair are collected in an internal format, say `if_list`. The `DELAY...` functions initialize the modified mode of operation. The `MAKE...` functions restore the previous mode of operation in combination with the production either of declarations, associated with the contents of `if_list`, or of an optimized representation of the contents of `if_list`.

Example 21 serves to introduce a variety of approaches to code generation, based on the use of these pairs of brackets and of the switch `GENTRANOPT`. We illustrate a more realistic use in example 22: generation of optimized fortran77 code for the computation of the entries of the inverse of a symmetric (3,3) matrix. It is a continuation of example 8 in subsection 3.1 and example 11 in subsection 3.2. It was also presented in [**?**], albeit in a slightly different form.

Example 21

The output of combined GENTRAN and SCOPE operations is by default given in fortran77 notation. We illustrate the different possibilities in the form of small pieces of code.

- The pair (`DELAYDECS`, `MAKEDECS`) encloses four GENTRAN commands. The first is needed to initialize the symbol table. The literal translation in internal format of the last three commands is stored in the `if_list`. The application of `MAKEDECS` leads to the restoration of the default GENTRAN regime, applied to the `if_list` and leading to the result, presented in the form of fortran77 code.

```
DELAYDECS$

  GENTRAN DECLARE <<a,b,c,d,q,w:real>>$
  GENTRAN a:=b+c$
  GENTRAN d:=b+c$
  GENTRAN <<q:=b+c;w:=b+c>>$

MAKEDECS$

      REAL A,B,C,D,Q,W
      A=B+C
      D=B+C
      Q=B+C
      W=B+C
```

- We repeat the previous commands, but execute them in a slightly different setting by turning `ON` the switch `GENTRANOPT`. This time the arithmetical rules in each of the three last GENTRAN commands are optimized separately. This is illustrated by the form of the output. The last piece of code contains the cse `B+C`, which is presented under the name `Q` in the fortran77 output.

```
ON GENTRANOPT$

DELAYDECS$

  GENTRAN DECLARE <<a,b,c,d,q,w:real>>$
  GENTRAN a:=b+c$
  GENTRAN d:=b+c$
  GENTRAN <<q:=b+c;w:=b+c>>$

MAKEDECS$

      REAL B,C,A,D,Q,W
      A=B+C
      D=B+C
      Q=B+C
      W=Q
```

```
OFF GENTRANOPT$
```

- We can improve the optimization strategy by using the function pair (`DELAYOPTS`, `MAKEOPTS`) in stead of the pair (`DELAYDECS`, `MAKEDECS`). All blockwise combinable arithmetic is collected. These blocks of straight line code are optimized as separate input sets E_{in}, when activating `MAKEOPTS`.

```
DELAYOPTS$

  GENTRAN a:=b+c$
  GENTRAN d:=b+c$
  GENTRAN <<q:=b+c;w:=b+c>>$

MAKEOPTS$

      A=B+C
      D=A
      Q=A
      W=A
```

- We can furhter improve the optimization strategy by using the function pair (`DELAYOPTS`, `MAKEOPTS`) inside the pair (`DELAYDECS`, `MAKEDECS`). All blockwise combinable arithmetic is collected. These blocks of straight line code are optimized as separate input sets E_{in}, when activating `MAKEOPTS`. But this time the results are added in internal format to the `if_list` version, being maintained, as to obey the `DELAYDECS` regime.

```
DELAYDECS$

  GENTRAN DECLARE <<a,b,c,d,q,w:real>>$

  DELAYOPTS$

    GENTRAN a:=b+c$
    GENTRAN d:=b+c$
    GENTRAN <<q:=b+c;w:=b+c>>$

  MAKEOPTS$

MAKEDECS$

      REAL B,C,A,D,Q,W
      A=B+C
      D=A
      Q=A
      W=A
```

- A slightly more realistic example suggests how easily optimized code for sets of matrix entries can be obtained. We use two identical matrices `a` and `d`. The latter is not introduced at the REDUCE level, but simply used inside a GENTRAN command. The entries of `a` are initialized inside a double for-loop. After each initialization a GENTRAN command is applied, using the special assignment operator `::=:` for correctly combining entry names and entry values. The REDUCE algebraic mode assignments are again used, when identifying the matrix `d` with the matrix `a`, applying the special assignment operator `:=:` in a separate GENTRAN command. The latter command is internally expanded into separate GENTRAN commands for each entry assignment. By using the pair (`DELAYOPTS`, `MAKEOPTS`) one block of straigt line code is constructed and optimized. It consists of two sets of assignments, one for the entries of the matrix `a` and another for the entries of the matrix `d`. The presented output shows that both matrices are indeed identical.

```
MATRIX a(4,4);

DELAYDECS$

  GENTRAN DECLARE <<a(4,4),d(4,4),b,c:real>>$

  DELAYOPTS$

    FOR i:=1:4 DO FOR j:=1:4 DO << a(i,j):=(i+j)*(b+c)+i*b+j*c;
                                   GENTRAN a(i,j)::=:a(i,j)>>$
    GENTRAN d:=:a$

  MAKEOPTS$

MAKEDECS$

      REAL B,C,G56,G54,G57,G55,A(4,4),D(4,4)
      A(1,1)=3.0*(B+C)
      G56=5.0*C
      A(1,2)=G56+4.0*B
      G54=5.0*B
      G57=7.0*C
      A(1,3)=G57+G54
      A(1,4)=6.0*B+9.0*C
      A(2,1)=G54+4.0*c
      A(2,2)=2.0*A(1,1)
      G55=7.0*B
      A(2,3)=G55+8.0*C
      A(2,4)=2.0*A(1,2)
      A(3,1)=G56+G55
      A(3,2)=G57+8.0*B
      A(3,3)=3.0*A(1,1)
      A(3,4)=10.0*B+11.0*C
      A(4,1)=9.0*B+6.0*C
      A(4,2)=2.0*A(2,1)
      A(4,3)=11.0*B+10.0*C
      A(4,4)=4.0*A(1,1)
      D(1,1)=A(1,1)
      D(1,2)=A(1,2)
      D(1,3)=A(1,3)
      D(1,4)=A(1,4)
      D(2,1)=A(2,1)
      D(2,2)=A(2,2)
      D(2,3)=A(2,3)
      D(2,4)=A(2,4)
      D(3,1)=A(3,1)
      D(3,2)=A(3,2)
      D(3,3)=A(3,3)
      D(3,4)=A(3,4)
      D(4,1)=A(4,1)
      D(4,2)=A(4,2)
      D(4,3)=A(4,3)
      D(4,4)=A(4,4)
```

- Finally, and again only for illustrative purposes the fifth piece of code is again executed in an almost identical manner. We omit the declarations and replace the instruction `GENTRAN d:=:a$` by the command `GENTRAN a:=:a$`. Hence the matrix a is simply redefined. As stated in section 7 redundant assignments — producing dead code, for instance by copying previous assignments — are automatically removed. as part of the optimization process. Therefore the optimized entry values of the matrix `a` are presented only once.

```
DELAYOPTS$

  FOR i:=1:4 DO FOR j:=1:4 DO << a(i,j):=(i+j)*(b+c)+i*b+j*c;
                                GENTRAN a(i,j)::=:a(i,j)>>$
  GENTRAN a:=:a$

MAKEOPTS$
```

```
A(1,1)=3.0*(B+C)
G111=5.0*C
A(1,2)=G111+4.0*B
G109=5.0*B
G112=7.0*C
A(1,3)=G112+G109
A(1,4)=6.0*B+9.0*C
A(2,1)=G109+4.0*C
A(2,2)=2.0*A(1,1)
G110=7.0*B
A(2,3)=G110+8.0*C
A(2,4)=2.0*A(1,2)
A(3,1)=G111+G110
A(3,2)=G112+8.0*B
A(3,3)=3.0*A(1,1)
A(3,4)=10.0*B+11.0*C
A(4,1)=9.0*B+6.0*C
A(4,2)=2.0*A(2,1)
A(4,3)=11.0*B+10.0*C
A(4,4)=4.0*A(1,1)
```

□

Example 22

Let us now again look at the symmetric (3,3) matrix `m`, already used in the examples 8 and 11. For completeness we begin by showing the entry values. We generate optimized fortran77 code for the inverse `mnv` of `m` and store it in a file, named `inverse.code`, using the function pair (`GENTRANOUT`, `GENTRANSHUT`). Inside this pair we apply the pair (`DELAYDECS`, `MAKEDECS`). The latter pair encloses in turn the pair (`DELAYOPTS`, `MAKEOPTS`). Inside these innermost brackets four different sections of code can be distinguished. The first section consists of three `LITERAL` commands, for inserting comment in the target code. The second is formed by a double for-loop. Essential are the applications of the GENTRAN functions `tempvar` and `markvar` for assigning internal names to matrix entry values. These applications are very similar to the approach, chosen in example 11. The third section is again formed by `LITERAL` commands and the last orders the creation of the entries of the inverse matrix `mnv`. Before introducing the file `inverse.code` we selected `S0` as initial cse name, using the function `INAME`, and turned `ON` the switches `ACINFO` and `DOUBLE`.

Observe that directly after the MAKEOPTS instruction two sets of tables are printed with information about the arithmetic complexity of the two blocks of code, which are generated in the second and the last section. We activated ACINFO to show this effect. The tables are printed as a side effect. The output itself is directed to the file `inverse.code`. Looking at the contents of this file, given below, shows three different kinds of internally generated names. A number of S-names is created during the optimization of the first block of straight line code, created with the second section. In this piece of code we also notice T-names, generated by `tempvar` applications. The intial T character is the default internal GENTRAN selection for (temporarily needed) names. And finally G-names, introduced by applications of `gensym`, during the second optimization operation for reducing the arithmetic complexity of the entries of `mnv`. Because the code splitting is internally performed, the second SCOPE application is missing its INAME initialisation, thus leading to the application of `gensym`. Observe as well that INAME can be used as a separate facility as well.

```
OFF EXP$

MATRIX m(3,3),mnv(3,3)$
IN "matrix.M"$

m(1,1) :=  - ((j30y - j30z + 9*m30*p )*sin(q3)

                                        2                        2
               - 18*cos(q2)*cos(q3)*m30*p  - j10y - j30y - m10*p

                           2
               - 18*m30*p )

                                                    2         2
m(2,1) := m(1,2) :=  - ((j30y - j30z + 9*m30*p )*sin(q3)

                              2                  2
    - 9*cos(q2)*cos(q3)*m30*p  - j30y - 9*m30*p )

                                                    2
m(3,1) := m(1,3) :=  - 9*sin(q2)*sin(q3)*m30*p

                                   2         2                 2
m(2,2) :=  - ((j30y - j30z + 9*m30*p )*sin(q3)  - j30y - 9*m30*p )

m(3,2) := m(2,3) := 0
```

```
                         2
m(3,3) := j30x + 9*m30*p

INAME s0$
ON ACINFO,DOUBLE$

GENTRANOUT "inverse.code"$

DELAYDECS$

  GENTRAN DECLARE <<mnv(3,3),p,m30,j30y,j30z,q3,m10,q2,j10y,j30x:real>>$
  DELAYOPTS$

    GENTRAN LITERAL "C",tab!*,"  ",cr!*$
    GENTRAN LITERAL "C",tab!*," -- Computation of relevant M-entries --",cr!*$
    GENTRAN LITERAL "C",tab!*,"  ",cr!*$

    FOR j:=1:3 DO FOR k:=j:3 DO
     IF m(j,k) NEQ 0 THEN
      << s:=tempvar('real); markvar s;
         GENTRAN eval(s):=:m(j,k);
         m(j,k):=m(k,j):=s
      >>$

    GENTRAN LITERAL "C",tab!*,"  ",cr!*$
    GENTRAN LITERAL "C",tab!*," -- Computation of the inverse of M --",cr!*$
    GENTRAN LITERAL "C",tab!*,"  ",cr!*$

    GENTRAN mnv:=:m^(-1)$

  MAKEOPTS$

Number of operations in the input is:

Number of (+/-) operations      : 17
Number of unary - operations    : 4
Number of * operations          : 30
Number of integer ^ operations  : 14
Number of / operations          : 0
Number of function applications : 9


Number of operations after optimization is:
```

```
Number of (+/-) operations      : 11
Number of unary - operations    : 1
Number of * operations          : 12
Number of integer ^ operations  : 0
Number of / operations          : 0
Number of function applications : 4


Number of operations in the input is:

Number of (+/-) operations      : 20
Number of unary - operations    : 4
Number of * operations          : 45
Number of integer ^ operations  : 20
Number of / operations          : 9
Number of function applications : 0


Number of operations after optimization is:

Number of (+/-) operations      : 4
Number of unary - operations    : 2
Number of * operations          : 11
Number of integer ^ operations  : 0
Number of / operations          : 4
Number of function applications : 0

MAKEDECS$

GENTRANSHUT "inverse.code"$
```

The contents of the file `inverse.code` is:

```
      DOUBLE PRECISION P,M30,J30Y,J30Z,Q3,M10,Q2,J10Y,J30X,S0,S7,S5,S4,
     . S13,S11,T0,T3,T1,T2,T4,G153,G152,G151,G147,G155,G156,MNV(3,3)
C
C      -- Computation of relevant M-entries --
C
      S0=DSIN(Q3)
      S7=P*P
      S5=S7*M30
      S4=S5*DCOS(Q3)*DCOS(Q2)
      S13=9.0D0*S5
      S11=(S13+J30Y-J30Z)*S0*S0
      T0=J30Y+J10Y+18.0D0*(S4+S5)+S7*M10-S11
      T3=S13+J30Y-S11
      T1=T3+9.0D0*S4
      T2=-(S13*DSIN(Q2)*S0)
      T4=S13+J30X
C
C      -- Computation of the inverse of M --
C
      G153=T2*T2
      G152=T1*T1
      G151=T0*T4
      G147=G151*T3-(G153*T3)-(G152*T4)
      G155=T4/G147
      MNV(1,1)=G155*T3
      MNV(1,2)=-(G155*T1)
      G156=T2/G147
      MNV(1,3)=-(G156*T3)
      MNV(2,1)=MNV(1,2)
      MNV(2,2)=(G151-G153)/G147
      MNV(2,3)=G156*T1
      MNV(3,1)=MNV(1,3)
      MNV(3,2)=MNV(2,3)
      MNV(3,3)=(T0*T3-G152)/G147
```

Let us now repeat the generation process in a slightly different setting. We leave out the comment generating instructions, thus creating only one block of straight line code to be optimized. We choose for an `S`-name selection based on `tempvar` applications and for `T`-names for cse's. This time the default use of `gensym` is not necessary.

The contents' of both output files illustrate quotient optimization. All denominators, being the determinant of the matrix `m`, are identical. The set of rational entries of `MNV` contains the cse's `G155 (T45)` and `G156 (T46)`.

```
TEMPVARNAME!*:='s$
TEMPVARNUM!*:=0$
INAME t0$

GENTRANOUT "inverse.code2"$

DELAYDECS$

  GENTRAN DECLARE <<mnv(3,3),p,m30,j30y,j30z,q3,m10,q2,j10y,j30x:real>>$
  DELAYOPTS$
```

```
    FOR j:=1:3 DO FOR k:=j:3 DO
     IF m(j,k) NEQ 0 THEN
      << s:=tempvar('real); markvar(s);
         GENTRAN eval(s):=:m(j,k);
         m(j,k):=m(k,j):=s
      >>$

    GENTRAN mnv:=:m^(-1)$

  MAKEOPTS$

Number of operations in the input is:

Number of (+/-) operations      : 37
Number of unary - operations    : 8
Number of * operations          : 75
Number of integer ^ operations  : 34
Number of / operations          : 9
Number of function applications : 9


Number of operations after optimization is:

Number of (+/-) operations      : 15
Number of unary - operations    : 3
Number of * operations          : 23
Number of integer ^ operations  : 0
Number of / operations          : 4
Number of function applications : 4

MAKEDECS$

GENTRANSHUT "inverse.code2"$
```

The contents of the file `inverse.code2` is:

```
      DOUBLE PRECISION P,M30,J30Y,J30Z,Q3,M10,Q2,J10Y,J30X,T9,T40,T32,
     . T31,T49,T47,S0,S3,S1,S2,S4,T39,T38,T36,T30,T45,T46,MNV(3,3)
      T9=DSIN(Q3)
      T40=P*P
      T32=T40*M30
      T31=T32*DCOS(Q3)*DCOS(Q2)
      T49=9.0D0*T32
      T47=(T49+J30Y-J30Z)*T9*T9
      S0=J30Y+J10Y+18.0D0*(T31+T32)+T40*M10-T47
      S3=T49+J30Y-T47
      S1=S3+9.0D0*T31
      S2=-(T49*DSIN(Q2)*T9)
      S4=T49+J30X
      T39=S2*S2
      T38=S1*S1
      T36=S0*S4
      T30=T36*S3-(T39*S3)-(T38*S4)
      T45=S4/T30
      MNV(1,1)=T45*S3
      MNV(1,2)=-(T45*S1)
      T46=S2/T30
      MNV(1,3)=-(T46*S3)
      MNV(2,1)=MNV(1,2)
      MNV(2,2)=(T36-T39)/T30
      MNV(2,3)=T46*S1
      MNV(3,1)=MNV(1,3)
      MNV(3,2)=MNV(2,3)
      MNV(3,3)=(S0*S3-T38)/T30
```

A comparison between the arithmetic complexities given here and in example 11 shows that computing the entries of MNV($=$M^{-1}) instead of the value of the determinant of M, only requires 2 extra additions, 1 extra negation, 6 extra multilications and 4 extra divisions.

□

Other examples of this combined use of GENTRAN and SCOPE can be found in [**?**, **?**]. The symbolic-numeric strategy discussed in [**?**] also relies on the ALGOPT facilities, which were introduced earlier.

9 Symbolic Mode Use of SCOPE 1.5

Both the `OPTIMIZE` command and the `ALGOPT` function are transformed into the same symbolic mode function, called `SYMOPTIMIZE`. It is this function, which governs the optimization process as a whole, delivering the results of an optimization run as a side effect, for instance by making it visible on a screen or by storing it in a file. Using `SYMOPTIMIZE` is straighforward, once the syntax for its five actual parameters is known. If we set `ON INTERN` a `SYMOPTIMIZE` application will deliver a list, containing the correctly ordered results of an optimization operation in the form of assignment statements in prefix form in Lisp notation. The thus provided results can function as one of the five actual parameters for `SYMOPTIMIZE` as well. This simple feature helps avoiding file traffic when stepwise optimizing code and as illustrated earlier in example 17 in section 5. Before illustrating that in example 23 we present the syntax of the actual parameters for `SYMOPTIMIZE`:

<SCOPE_application> ::= $\cdots$ |
    `SYMOPTIMIZE`(<ssetq_list>, <infile_list>, <outfile_name>, <cse_prefix>
,
    <declaration_list>)

<ssetq_list>	::=	(<ssetq_seq>)
<ssetq_seq>	::=	<ssetq_stat> [<ssetq_seq>]
<ssetq_stat>	::=	(**setq** <lhs_id> <rhs>) | (**rsetq** <lhs_id> <rhs>) |
		(**lsetq** <lhs_id> <rhs>) | (**lrsetq** <lhs_id> <rhs>)
<lhs_id>	::=	<id> | <subscripted_id>
<subscripted_id>	::=	(<id> <s_subscript_seq>)
<s_subscript_seq>	::=	<s_subscript> [<s_subscript_seq>]
<s_subscript>	::=	<integer> | <integer prefix_expression>
<rhs>	::=	<prefix_expression>
<infile_list>	::=	(<infile_seq>)
<infile_seq>	::=	<infile_name> [<infile_seq>]
<infile_name>	::=	<string_id>
<outfile_name>	::=	<string_id>
<declaration_list>	::=	(<declaration_seq>)
<declaration_seq>	::=	<declaration> <declaration_seq>
<declaration>	::=	(<type> <lhs_id_seq>)
<lhs_id_seq>	::=	<lhs_id> <lhs_id_seq>

The above given syntax requires some explanation:

- The presented ssetq syntax is incomplete. The prefix equivalent of any object, introduced in subsection 3.1 and of any a_object, defined in subsection 3.2, is accepted as ssetq item. Such prefix equivalents can be obtained quite easily by using the function `show`:

  ```
  SYMBOLIC PROCEDURE show u; prettyprint u$
  SYMBOLIC OPERATOR show$
  ```

  The explicit presentation of a subset of the syntax rules for ssetq is given to suggest that local simplification in symbolic mode can be brought in easily by using the assignment operators `lsetq`, `lrsetq` and `rsetq`. The algebraic mode equivalent of these operators is `::=`, `::=:` and `:=:`, respectively. Their effect on simplification is discussed in subsection 2.4 and already shown in a number of examples. In addition it is worth noting that any (sub)expression in a lhs_id or a rhs may contain any number of calls to `eval`. These calls lead to simplification of their arguments, prior to optimization. Details about the use of `eval` are presented in the GENTRAN manual [**?**].

- Since we operate in symbolic mode the last four formal parameters have possibly to be replaced by quoted actual parameters. This is illustrated in example 23.

- The infile_seq consists of file names in string notation. The contents' of such input files may contain any form of infix input, obeying the syntax rules for objects and/or a_objects, as introduced in the subsections 3.1 and 3.2, respectively.
- The single output file name outfile_name ought to be given in string notation as well. The outfile_name is properly closed. The default output is REDUCE infix in an `ON NAT` fashion. Alternatives are discussed above: `ON AGAIN` or `OFF NAT`, both leading to re-readable output, or an application of `OPTLANG` for a non `nil` argument.
- The declaration list presents declarations in prefix notation. The list is used to initialize the symbol table prior to optimization. This information is used for dynamically typing the result of an optimization process. In addition it is used to determine wether subscripted names denote array elements or a function call. The latter is replaced by a cse name in the presented output, whereas the former is not.
- The five parameters of `SYMOPTIMIZE` correspondent with optional extensions of the `OPTIMIZE` command. When part of these options remains unused, `nil` has to taken as value for the corresponding actual parameters.

We illustrate the symbolic mode variant of the `OPTIMIZE` command by repeating example 17 from section 5, albeit in a modified setting.

Example 23

The script explains itself.

```
LISP$
ON ACINFO,INPUTC,INTERN,AGAIN$

prettyprint(prefixlist:=SYMOPTIMIZE(nil,'("f1" "f2"),nil,'c,nil))$

                                    2
              2        (x + y)                8        2      2
      2*(sin(x)  - cos(e        ) + 3*cos(x)) *(x + y)  + 4*y  + 4*y
e1 := ---------------------------------------------------------------
                                     3*x + 2*y

                                    2
              2        (x + y)                2        3
e2 := (4*(sin(x)  - cos(e        ) + 2*cos(x)) *(x + y)
             2                   2
       + (4*x  - 4*y) - 6*x)/(8*x  + 3*y - 2*x)

Number of operations in the input is:

Number of (+/-) operations      : 16
Number of unary - operations    : 0
Number of * operations          : 16
Number of integer ^ operations  : 11
Number of / operations          : 2
Number of function applications : 8


Number of operations after optimization is:

Number of (+/-) operations      : 16
Number of unary - operations    : 0
Number of * operations          : 16
Number of integer ^ operations  : 6
Number of / operations          : 2
Number of function applications : 4

((setq gsym c23)
   (setq cses (plus c18 c10 c20 c8 c6 c14))
   (setq c14 (plus x y))
   (setq c6 (expt c14 2))
   (setq c8 (cos x))
   (setq c20 (plus (expt (sin x) 2) (minus (cos (expt e c6)))) )
   (setq c10 (plus (times 3 x) (times 2 y)))
   (setq e1
      (quotient
         (plus
            (times 4 y)
            (times 4 (expt y 2))
            (times 2 c6 (expt (plus c20 (times 3 c8)) 8)))
         c10))
   (setq c18 (expt x 2))
   (setq e2
      (quotient
         (plus
            (times 4 c18)
            (minus (times 2 c10))
            (times 4 c6 c14 (expt (plus c20 (times 2 c8)) 2)))
         (plus (times 8 c18) (minus (times 2 x)) (times 3 y)))) )
```

```
OFF INTERN,AGAIN,PERIOD$
ON DOUBLE,FORT$

SYMOPTIMIZE(prefixlist,'("f3"), '"f7",'d,'((real e1 e2 e3 x y)))$

gsym := c23

cses := c18 + c10 + c20 + c8 + c6 + c14

c14 := x + y

         2
c6 := c14

c8 := cos(x)

             2        c6
c20 := sin(x)  - cos(e  )

c10 := 3*x + 2*y

                2                   8
       4*y + 4*y  + 2*c6*(c20 + 3*c8)
e1 := --------------------------------
                     c10

        2
c18 := x

                                            2
       4*c18 - 2*c10 + 4*c6*c14*(c20 + 2*c8)
e2 := ---------------------------------------
                8*c18 - 2*x + 3*y

                         2
                  (x + y)                      2          2
       4*sin(cos(e        )) + sin(x + y) + (4*x  - x + 2*y)
e3 := -------------------------------------------------------
                      3*y + f(x,g( - cos(x)))

Number of operations in the input is:

Number of (+/-) operations      : 23
Number of unary - operations    : 1
Number of * operations          : 20
Number of integer ^ operations  : 9
Number of / operations          : 3
Number of function applications : 11


Number of operations after optimization is:

Number of (+/-) operations      : 15
Number of unary - operations    : 1
Number of * operations          : 24
Number of integer ^ operations  : 0
Number of / operations          : 3
Number of function applications : 8
```

The contents of the output file `f7` is:

```
      DOUBLE PRECISION X,Y,D19,D16,C8,D1,D2,C20,D29,C10,D6,D38,D37,D31,
     . E1,C18,D9,D32,D27,D30,E2,D20,E3
      D19=X+Y
      D16=D19*D19
      C8=DCOS(X)
      D1=DSIN(X)
      D2=DCOS(DEXP(D16))
      C20=D1*D1-D2
      D29=2*Y
      C10=D29+3*X
      D6=C20+3*C8
      D38=D6*D6
      D37=D38*D38
      D31=4*Y
      E1=(D31+D31*Y+2.0D0*D16*D37*D37)/C10
      C18=X*X
      D9=C20+2*C8
      D32=4*C18
      D27=D32-X
      D30=3*Y
      E2=(D32-(2.0D0*C10)+4.0D0*D9*D9*D19*D16)/(D30+2.0D0*D27)
      D20=D29+D27
      E3=(4.0D0*DSIN(D2)+DSIN(D19)+D20*D20)/(D30+F(X,G(-C8)))
```

□

We especially designed these symbolic mode facilities for our joint research with Delft Hydraulics concerning code generation for an incompressible

Navier-Stokes problem [**?**].

A final remark: The `ON PREFIX` mode of operation, in both algebraic and symbolic mode causes the results of a SCOPE application to be presented in the form of an association list, called `Prefixlist`. The pairs are formed by lhs_id's and rhs values in prefixform. This lisp S-expression can be used to create an alternative version of the optimization results, in whatever target language the user prefers to choose.

Example 24

We show the `ON PREFIX` effect. When switching to symbolic mode (command 5) we can again obtain the output, assigned as value to the global identifer `prefixlist`. The `ON PREFIX` facility allows storage in a file for later use. When working in symbolic mode it is of course possible to apply `ON INTERN` in stead and to remove the `setq` extensions from the provided output value, if desired.

```
REDUCE 3.6, 15-Jul-95 ...

1: LOAD_PACKAGE nscope$

2: OPTIMIZE a:=b+c*sin(x), d:=c*sin(x)*cos(y);

g7 := sin(x)*c
a := g7 + b
d := g7*cos(y)

3: ON PREFIX$

4: input 2;

Prefixlist:=
((g3 times (sin x) c) (a plus g3 b) (d times g3 (cos y)))

5: LISP$

6: prettyprint prefixlist$
((g3 times (sin x) c) (a plus g3 b) (d times g3 (cos y)))

7: caar prefixlist;

g3

7: cdar prefixlist;
```

```
(times (sin x) c)

9: BYE;
```

□

10 A Syntax Summary of SCOPE 1.5

REDUCE is extended with some commands, designed to apply the facilities offered by SCOPE in a flexible way. The syntactical rules, defining how to activate SCOPE in both algebraic and symbolic mode, are given in subsection 10.1. A short overview of the set of additional functions is given in subsection 10.2 and the relevant switches are again presented in subsection 10.3.

10.1 SCOPE's Toplevel Commands

We assume the syntax of `id`'s, `integer`'s and the like to be already known. Hence we do not present an exhaustive description of the rules.

```
<REDUCE_command>  ::=  ···  |  <SCOPE_application>
      <GSTRUCTR_application>  | <GHORNER_application>  |
<SCOPE_application>  ::=  <OPTIMIZE command>  |
      <ALGOPT application>  | <SYMOPTIMIZE application>


<OPTIMIZE command>  ::=
     OPTIMIZE <object_seq> [IN <file_id_seq>] [OUT <file_id>]
                [INAME <cse_prefix>] [DECLARE <declaration_group>] |
     OPTIMIZE [<object_seq>] IN <file_id_seq> [OUT <file_id>]
                [INAME <cse_prefix>] [DECLARE <declaration_group>]


<ALGOPT application>  ::=
     ALGOPT(<a_object_list>[,<string_id_list>][,<cse_prefix>]) |
     ALGOPT([<a_object_list>,]<string_id_list>[,<cse_prefix>])


<SYMOPTIMIZE application>  ::=
  SYMOPTIMIZE(<ssetq_list>, <infile_list>, <outfile_name>, <cse_prefix>,
             <declaration_list>)
```

<`GSTRUCTR`_application>	::=	`GSTRUCTR` <stat_group> [`NAME` <cse_prefix>]
<stat_group>	::=	≪ <stat_list> ≫
<stat_list>	::=	<gstat> [; <stat_list>]
<gstat>	::=	<name> := < expression> | <matrix_id>
<`GHORHER`_application>	::=	`GHORNER` <stat_group> [`VORDER` <id_seq>]
<id_seq>	::=	<id>[,<id_seq>]

```
<object_seq>             ::=  <object>[,<object_seq>]
<object>                 ::=  <stat>  |  <alglist>  |  <alglist_production>
<stat>                   ::=  <name>  <assignment operator>  <expression>
<assignment operator>    ::=  :=  |  ::=  |  ::=:  |  :=:
<alglist>                ::=  {<eq_seq>}
<eq_seq>                 ::=  <name>  =  <expression>[,<eq_seq>]
<alglist_production>     ::=  <name>  |  <function_application>
<name>                   ::=  <id>  |  <id> (<a_subscript_seq>)
<a_subscript_seq>        ::=  <a_subscript>[,<a_subscript_seq>]
<a_subscript>            ::=  <integer>  |  <integer infix_expression>
<cse_prefix>             ::=  <id>

<a_object_list>          ::=  <a_object>  | {<a_object>[,<a_object_seq>]}
<a_object_seq>           ::=  <a_object>[,<a_object_seq>]
<a_object>               ::=  <id>  |  <alglist>  |  <alglist_production>  |
                              {<a_object>}

<function_application>   ::=  ALGSTRUCTR (<arg_list> [, <cse_prefix> ]) |
                              ALGHORNER (<arg_list> [,{<id_seq>}]) |
                              ...
<arg_list>               ::=  <arg_list_name>  |  {<arg_seq>}
<arg_seq>                ::=  <arg>[,<arg_seq>]
<arg>                    ::=  <matrix_id>  |  <name>=<expression>
<arg_list_name>          ::=  <id>

<file_id_seq>            ::=  <file_id> [,<file_id_seq>]
<file_id>                ::=  <id> | <string_id>
<string_id_list>         ::=  <string_id> | {<string_id_seq>}
<string_id_seq>          ::=  <string_id> [,<string_id_seq>]
<string_id>              ::=  "<id>" | "<id> . <f_extension>"

<declaration_group>      ::=  <declaration>  |  ≪  <declaration_list>  ≫
<declaration_list>       ::=  <declaration>[; <declaration_list>]
<declaration>            ::=  <range_list>: IMPLICIT <type>  | <id_list>:<type>
<range_list>             ::=  <range>[,<range_list>]
<range>                  ::=  <id>  |  <id> – <id>
<id_list>                ::=  <id>[,<id_list>]
<type>                   ::=  integer | real | complex | real*8 | complex*16

<ssetq_list>             ::=  (<ssetq_seq>)
<ssetq_seq>              ::=  <ssetq_stat>  [<ssetq_seq>]
<ssetq_stat>             ::=  (setq <lhs_id>  <rhs>)  | (rsetq <lhs_id>  <rhs>)  |
                              (lsetq <lhs_id>  <rhs>)  | (lrsetq <lhs_id>  <rhs>)
```

```
<lhs_id>              ::=   <id>  |  <subscripted_id>
<subscripted_id>      ::=   (<id>  <s_subscript_seq>)
<s_subscript_seq>     ::=   <s_subscript> [ <s_subscript_seq>]
<s_subscript>         ::=   <integer>  |  <integer prefix_expression>
<rhs>                 ::=   <prefix_expression>
<infile_list>         ::=   (<infile_seq>)
<infile_seq>          ::=   <infile_name> [ <infile_seq>]
<infile_name>         ::=   <string_id>
<outfile_name>        ::=   <string_id>
<declaration_list>    ::=   (<declaration_seq>)
<declaration_seq>     ::=   <declaration>  <declaration_seq>
<declaration>         ::=   (<type>  <lhs_id_seq>)
<lhs_id_seq>          ::=   <lhs_id>  <lhs_id_seq>
```

10.2 Additional SCOPE-functions

Fifteen additional functions can be used. We shortly summarize their name and use:

Name(s)	Introduced in	See the examples:
`SCOPE_SWITCHES`	3	1
`SETLENGTH, RESETLENGTH`	3.1	2, 5, 12 and 13
`ARESULTS, RESTORABLES, ARESTORE, RESTOREALL`	3.1	9, 12 and 15
`OPTLANG`	6	18
`VECTORCODE, VCLEAR`	7	20
`DELAYDECS, MAKEDECS, DELAYOPTS, MAKEOPTS`	8	21 and 22
`INAME`	8	22

10.3 The relevant REDUCE, GENTRAN and SCOPE switches

We also shortly summarize the use of the switches, which were introduced in section 3 in example 1:

Name	Origin	Illustrated in the examples:
`ACINFO`	SCOPE	8, 11, 22 and 23
`AGAIN`	SCOPE	17 and 23
`DOUBLE`	GENTRAN	23
`EVALLHSEQP`	REDUCE	11
`EXP`	REDUCE	8, 13, 16 and 22
`FORT`	REDUCE	4, 6, 8, 11, 20 and 23
`FTCH`	SCOPE	4 and 5

Name	Origin	Illustrated in the examples:
`GENTRANOPT`	GENTRAN	21
`INPUTC`	SCOPE	3, 6, 7, 13, 17 and 23
`INTERN`	SCOPE	23
`NAT`	REDUCE	8 and 17
`PERIOD`	REDUCE	4
`PREFIX`	SCOPE	24
`PRIALL`	SCOPE	2
`PRIMAT`	SCOPE	13
`POUNDBF`	REDUCE	19
`ROUNDED`	REDUCE	7 and 22
`SIDREL`	SCOPE	13
`VECTORC`	SCOPE	20

11 SCOPE 1.5 Installation Guide

SCOPE 1.5 is easily installed. The usual code compilation facilities of REDUCE can be applied. In view of the frequent interaction between SCOPE and GENTRAN a compiled version of GENTRAN is required during the creation of the load module for SCOPE 1.5. The compilation process itself is vizualized below.

```
faslout "~infhvh/mkscope/scope_15";

lisp in "~infhvh/mkscope/cosmac.red"$
lisp in "~infhvh/mkscope/codctl.red"$
lisp in "~infhvh/mkscope/codmat.red"$
lisp in "~infhvh/mkscope/codopt.red"$
lisp in "~infhvh/mkscope/codad1.red"$
lisp in "~infhvh/mkscope/codad2.red"$
lisp in "~infhvh/mkscope/coddec.red"$
lisp in "~infhvh/mkscope/codpri.red"$
lisp in "~infhvh/mkscope/codgen.red"$
lisp in "~infhvh/mkscope/codhrn.red"$
lisp in "~infhvh/mkscope/codstr.red"$
lisp in "~infhvh/mkscope/coddom.red"$
%lisp in "~infhvh/mkscope/apatch.red"$
algebraic;

faslend ;
end;
```

The subdirectory `mkscope` in the author's directory system contains the files with the source code of SCOPE 1.5. The order in which the files are read in is irrelevant except the first and the last. The file `cosmac.red` contains one module, named `cosmac`, which consists of a set of smacro procedures, designed to simplify access to the lower levels of the expression data base, employed during optimization. These smacro's are used in all other code sections. The last file in optional and usually executed to include new patches into a recompilable version of the package. Once it is stored in the `fasl` directory of the local REDUCE system it is available as a `load_package`.

In short, the files contain the following code sections:

- `cosmac.red` contains the module `cosmac`, consisting of smacro procedures, allowing access to the expression data base.

- `codctl.red` consists of the three modules `codctl`, `restore` and `minlenght`. The first is a large module, containing the optimization process managing facilities. The second module is added to regulate the interplay with the REDUCE simplifier, when entering optimizer output in algebraic mode using functions like `ARESULTS`. The last module serves to vary the minimal length of cse's using `SETLENGTH` and `RESETLENGTH`.
- `codmat.red` contains the module `codmat`, definig the parsing process and the expression data base setup and access facilities.
- `codopt.red`'s content is formed by the module `codopt`. It is the kernel of the optimization process, the implementation of the extended Breuer algorithm.
- `codad1.red` contains the module `codad1`, formed by additional facilities for improving the lay-out of the overall result, for information migration between different sections of the expression data base and for the application of the distributive law to remodel and compactify (sub)expression structure at any level.
- `codad2.red` contains the module `codad2`. This module defines five different possible activities during the optimization process. The first four regulate the so called *finishing touch*. The last one is a new section, defining how to optimize *rational forms* as part of the overall optimization process.
- `coddec.red` covers the declaration facilities, presented in section 6, collected in the module `coddec` and based on chapter 6 of [**?**]. The symbol table setup of GENTRAN is used.
- `codpri.red` is also formed by one module, called `codpri`. It covers all printing facilities. The first section is applied when the switch `PRIMAT` is turned `ON`. The latter is used to produce an internal list of pairs, consisting of the left hand side and the right hand side of assignment statements in prefix notation, and defining the output of the optimization process in sequential order. This prefix list is delivered to GENTRAN or REDUCE to make the results visible in the user prefered form. The intial version of this list, created directly after the optimization process, is improved using a collection of functions, also grouped together in the module `codpri`. These improvements may for instance be necessary to remove temporarily introduced names, internally employed as a result of a data dependency analysis.

- `codgen.red` consists of the module `codgen`. The interface between GENTRAN and SCOPE 1.5, introduced in section 8 of this manual is defined in this module.
- `codhrn.red` is formed by the module `ghorner`. It defines the facilities, presented in section 4.3 of this manual.
- `codstr.red` contains the module `gstructr`. This module defines the possibilities for expression structure recognition, as presented in section 4.2 of this manual.
- `coddom.red` finally, consists of one module, called `coddom`. It covers additional coefficient domain functions, needed to make the extended Breuere algorithm and the additional functions, collected in the modules `codad1` and `codad2` for instance, applicable for (multiple presicion) floating point coefficients.

A few additional remarks:

- GENTRAN plays an important role when creating declarations and output. The package is automatically loaded when executing one of the first instructions in the module `codctl`. Hence it may be necessary to look critically to the load instruction in `codctl` before installing SCOPE 1.5. By changing this load instruction we easily created a `fortran90` compatable SCOPE version [**?**]. At present it is only available for our own internal and experimental use.
- We believe the code to be almost version independent. Over the past years all uses of `nil` have been critically reviewed. However the `coddom` module may require version based maintenance when installing SCOPE 1.5.
- The present size of the source code is given in the table below. Comment is included in these figures

File naam	number of lines	number of characters
`cosmac.red`	172	4761
`codctl.red`	1439	61466
`codmat.red`	1488	72733
`codopt.red`	1243	68809
`codad1.red`	801	39175
`codad2.red`	1314	59217
`coddec.red`	928	41069
`codpri.red`	1371	62600
`codgen.red`	214	9120
`codhrn.red`	752	30549
`codstr.red`	308	11199
`coddom.red`	204	6638

- The modules `ghorner` and `gstructr` can be left out without harming the other facilities, presented in this manual.

References

Index