GENTRAN USER'S MANUAL

- VAXIMA VERSION -

* Draft *

Barbara L. Gates

Department of Mathematical Sciences
Kent State University
Kent, Ohio 44242

December 1985

ABSTRACT

GENTRAN is an automatic code GENerator and TRANslator which runs under VAXIMA and REDUCE. It constructs complete numerical programs based on sets of algorithmic specifications and symbolic expressions. Formatted FORTRAN, RATFOR or C code can be generated through a series of interactive commands or under the control of a template processing routine. Large expressions can be optimized and segmented into subexpressions of manageable size, and a special file-handling mechanism maintains stacks of open I/O channels to allow output to be sent to any number of files simultaneously and to facilitate recursive invocation of the whole code generation process. GENTRAN provides the flexibility necessary to handle most code generation applications.

This manual describes the usage of the GENTRAN package for VAXIMA.

CONTENTS

1. INTRODUCTION 4

1.1. The GENTRAN Code Generator and Translator
1.2. Organization of the Manual
1.2.1. Typographic Conventions

2. CODE GENERATION 6

2.1. Target Language Selection
2.2. Translation
2.2.1. Special Arguments
2.3. Code Generation: Evaluation Prior to Translation
2.3.1. The **eval** Function
2.3.2. The **rsetq** Function
2.3.3. The **lsetq** Function
2.3.4. The **lrsetq** Function
2.4. Type Declarations
2.4.1. Subprogram Type Declarations
2.4.2. A Note on Subscripted Variables
2.5. Comments and Literal Strings
2.6. The **fortran**, **ratfor**, and **c** Mode Switches
2.7. Code Optimization
2.8. Expression Segmentation
2.8.1. Implicit Type Declarations
2.8.2. Controlling Generation of Declarations
2.9. Generation of Temporary Variable Names
2.9.1. Marking Temporary Variables
2.9.2. Unmarking Temporary Variables
2.9.3. The Marked Variable Predicate
2.10. Generation of Statement Numbers

3. TEMPLATE PROCESSING 41

3.1. The Template Processing Command
3.1.1. Copying Files into Template Files
3.1.2. The Template File Stack
3.2. Insertion of Generated Declarations
3.3. Special Template Variables

4. OUTPUT REDIRECTION 53

4.1. File Selection Commands
4.1.1. **gentranout**
4.1.2. **gentranshut**
4.2. Operations on the Output File Stack
4.2.1. **gentranpush**
4.2.2. **gentranpop**
4.3. Temporary Output Redirection

5. MODIFICATION OF THE CODE GENERATION PROCESS 64

5.1. Mode Switches
5.2. Variables
5.2.1. Target Language Selection
5.2.2. Expression Optimization & Segmentation Control
5.2.3. Temporary Variable Names & Statement Numbers
5.2.4. Code Formatting

6. EXAMPLES 70

6.1. Example 1: Interactive Code Generation
6.2. Example 2: Template Processing
6.3. Example 3: Segmentation & Optimization Techniques
6.3.1. Code Generation
6.3.2. Segmentation
6.3.3. Expression Optimization
6.3.4. Generation of Temporary Variables to Suppress Simplification
6.3.5. Expression Optimization & Generation of Temporary Variables
6.3.6. Comparison of Code Optimization Techniques

REFERENCES 99

A. TRANSLATABLE VAXIMA STATEMENTS & EXPRESSIONS 101

A.1. Formal Definition
A.2. Examples

B. LIST OF GENTRAN FUNCTIONS, SWITCHES AND VARIABLES 108

1. INTRODUCTION

Solving a problem in science or engineering is often a two-step process. First the problem is modeled mathematically and derived symbolically to provide a set of formulas which describe how to solve the problem numerically. Next numerical programs are written based on this set of formulas to efficiently compute specific values for given sets of input. Computer algebra systems such as MACSYMA /9,10/ provide powerful tools for use in the formula-derivation phase but only provide primitive program-coding tools. The GENTRAN package has been constructed to automate the tedious, time consuming and error-prone task of writing numerical programs based on a set of symbolic expressions.

1.1. The GENTRAN Code Generator and Translator

The GENTRAN code GENeration and TRANslation package /4,5/, originally implemented in Franz LISP /3,14/ to run under VAXIMA /2/, has also been implemented in RLISP to run under REDUCE /6/. Although GENTRAN was originally created specifically to generate numerical code for use with an existing FORTRAN-based finite element analysis package /12,13/, it was designed to provide the flexibility required to handle most code generation applications. GENTRAN contains code generation functions, file-handling functions, mode switches, and global variables, all of which are accessible from both the user level and the LISP level of VAXIMA to give the user maximal control over the code generation process. Formatted FORTRAN /1/, RATFOR /7/ or C /8/ code can be generated from algorithmic specifications, i.e., descriptions of the behavior of the target numerical program expressed in the VAXIMA programming language, and from symbolically derived expressions and formulas.

In addition to arithmetic expressions and assignment statements, GENTRAN can also generate type declarations and control-flow structures. Code generation can be guided by user-supplied template file(s) to insert generated code into pre-existing program skeletons, or it can be accomplished interactively through a series of translation commands without the use of template files. Special mode switches enable the user to turn on or off specific features such as automatic segmentation of large expressions, and global variables allow the user to modify the code formatting process. Generated code can be sent to one or more files and, optionally, to the user's terminal. Stacks of open I/O channels facilitate temporary output redirection and recursive invocation of the code generation process.

1.2. Organization of the Manual

The remainder of this manual is divided into five sections. Sections 2 and 3 describe code generation. Section 2 explains interactive code generation, the expression segmentation facility, code optimization, and how temporary variables can be generated; then section 3 explains how code generation can be guided by a user-supplied template file. Section 4 describes file-handling functions which control output redirection, and section 5 describes user-accessible global variables and mode switches which alter the code generation process. Finally, section 6 presents three complete examples.

Throughout this manual, examples are given from both the user and the LISP levels of VAXIMA. It is assumed that the user knows that a <control>D typed at the user level puts VAXIMA into LISP mode and vice versa, and that programming at the LISP level is in Franz LISP.

1.2.1. Typographic Conventions

The following conventions are used in the syntactic definitions of functions throughout this manual:

- Command parts which must be typed exactly as shown are *underlined*.
- User-supplied arguments are not underlined.
- { ... } indicate optional command parts and are not actually part of the function call syntax.

The syntax of each GENTRAN function is shown terminated with a **;**. However, either **;** or **$** can be used to terminate any command with the usual VAXIMA meaning: **;** indicates that the returned value is to be printed, while **$** indicates that printing of the returned value is to be suppressed.

2. CODE GENERATION

GENTRAN generates numerical programs based on algorithmic specifications in the VAXIMA programming language and derived symbolic expressions produced by VAXIMA evaluations. FORTRAN, RATFOR, or C code can be produced. Type declarations can be generated, and comments and other literal strings can be inserted into the generated code. In addition, large arithmetic expressions can be optimized and segmented into a sequence of subexpressions of manageable size.

This section explains how to select the target language, generate code, control expression optimization and segmentation, and how to generate temporary variable names.

2.1. Target Language Selection

Before generating code, the target numerical language must be selected. GENTRAN is currently able to generate FORTRAN, RATFOR, and C code. The global lisp variable **gentranlang* determines which type of code is produced. **gentranlang* can be set at the user and LISP levels. It can be set to any value, but only three atoms have special meaning: *fortran*, *ratfor*, and *c*. Any other value is assumed to mean *fortran*. **gentranlang* is always initialized to *fortran*. At the Macsyma user level,

```
gentranlang(ratfor);
```

for example, sets **gentranlang* to *ratfor*.

2.2. Translation

The *gentran* (*gen*erate/*tran*slate) command is used to generate numerical code and also to translate code from algorithmic specifications in the VAXIMA programming language to code in the target numerical language. Section 2.3 explains code *generation*. This section explains code *translation*.

A substantial subset of all expressions and statements in the VAXIMA programming language can be translated directly into numerical code. The *gentran* command takes VAXIMA statements or procedure definitions, and translates them into code in the target language.

User-Level Syntax:

*gentran(*stmt1,stmt2,...,stmtn {,[f1,f2,...,fm]});

where stmt1,stmt2,...,stmtn is a sequence of one or more stmt's, each of which is any VAXIMA user level expression, (simple, group, or block) statement, or procedure definition that can be translated by GENTRAN into the target language[1]. Each stmt may contain calls to the special functions described in sections 2.2.1 - 2.2.5 and 2.3.1 - 2.3.4.

[f1,f2,...,fm] is an optional argument list containing one or more f's, where each f is one of:

`a string` = an output file
true = the terminal
false = the current output file(s)
all = all files currently open for output by GENTRAN
(see sect. 4)

LISP Level Syntax:

*(gentran '(*stmt1 stmt2 ... stmtn) '(f1 f2 ... fm))

where stmt1 stmt2 ... stmtn is a sequence of one or more stmt's, each of which is any VAXIMA LISP level expression, (simple, group, or block) statement, or procedure definition that can be translated by GENTRAN into the target language[2]. Each stmt may contain calls to the special functions described in sections 2.2.1 - 2.2.5 and 2.3.1 - 2.3.4.

---

[1] See Appendix A for a complete list of VAXIMA user-level expression and statement types that can be translated.

[2] See Appendix A for a complete list of VAXIMA LISP level expression and statement types that can be translated.

(f1 f2 ... fm) is an argument list containing any number of f's, where each f is one of:

```
a string  =  an output file
```

t = the terminal
nil = the current output file(s)
$all = all files currently open for output by GENTRAN
(see sect. 4)

Side Effects:

gentran translates each stmt into formatted code in the target language.

If the file list is empty, generated code is simply written to the current output file. However, if it contains one or more file arguments, then the current output file is temporarily overridden. Generated code is written to each file represented by f1,f2,...,fm for this command only. Files which were open prior to the call to *gentran* will remain open after the call, and files which did not exist or were not open prior to the call will be created (if necessary), opened, written to (or appended onto), and closed. The output file stack will be exactly the same both before and after the call.

Return Value:

gentran returns (a list of) the name(s) of the file(s) to which code was written.

Diagnostic Messages:

wrong type of arg

exp
cannot be translated

User Level Examples:

```
(c1) gentranlang(fortran)$

(c2) gentran( for i:1 thru n do v[i]:0.0 );
      do 25001 i=1,n
          v(i)=0.0
25001 continue

(d2)                        true

(c3) gentranlang(ratfor)$

(c4) gentran( for i:1 thru n do
                     for j:i+1 thru n do
                     (
                         x[j,i] : x[i,j],
                         y[j,i] : y[i,j]
                     )                    );
do i=1,n
    do j=i+1,n
        {
            x(j,i)=x(i,j)
            y(j,i)=y(i,j)
        }

(d4)                        true
```

```
(c5) gentranlang(c)$

(c6) gentran( p : a*x^2 + b*x + c );
p=a*power(x,2)+b*x+c;

(d6)                       true
```

LISP Level Examples:

```
<1>: (setq *gentranlang 'fortran)
fortran

<1>: (gentran '(((mdo) i 1 nil nil n nil ((msetq) ((v) i) 0.0))) ())
      do 25001 i=1,n
          v(i)=0.0
25001 continue
t

<1>: (setq *gentranlang 'ratfor)
ratfor

<1>: (gentran '(((mdo) i 1 nil nil n nil
                       ((mdo) j ((mplus) i 1) nil nil n nil
                              ((mprogn)
                               ((msetq) ((x) j i) ((x) i j))
                               ((msetq) ((y) j i) ((y) i j)))))) ())
do i=1,n
    do j=i+1,n
        {
            x(j,i)=x(i,j)
            y(j,i)=y(i,j)
        }
t

<1>: (setq *gentranlang 'c)
c

<1>: (gentran '(((msetq) p ((mplus) ((mtimes) a ((mexpt) x 2))
                                    ((mtimes) b x)
                                    c))) ())
p=a*power(x,2)+b*x+c;
t
```

2.2.1. Special Arguments

In addition to statements in the VAXIMA programming language, the *gentran* function can also translate special arguments into numerical code. Several of these arguments are described in this section.

Subprogram Headings & Bodies

FORTRAN and RATFOR function and subroutine headings, and C procedure headings can be generated separately from their corresponding bodies of code with the arguments given in this section.

User Level Syntax:

*subroutine(*name(p1,p2,...,pn))

*function({*type,} name(p1,p2,...,pn))

cprocedure({type,} name(p1,p2,...,pn))

*body(*stmt1,stmt2,...,stmtn)

where name is an atom, and p1,p2,...,pn is a sequence of zero or more atoms, representing a subroutine, function or procedure name, and a list of parameter names, respectively; type is an optional argument which, if supplied to either *function* or *cprocedure*, is a string representing the function's or procedure's return value type (e.g., "real*8", "double", etc.); and stmt1,stmt2,...,stmtn is a sequence of zero or more user level translatable VAXIMA statements representing the body of a subprogram.

LISP Level Syntax:

(($subroutine) ((name) p1 p2 ... pn))

(($function) {type} ((name) p1 p2 ... pn))

(($cprocedure) {type} ((name) p1 p2 ... pn))

(($body) stmt1 stmt2 ... stmtn)

where name is an atom, and p1 p2 ... pn is a sequence of zero or more atoms, representing a subroutine function or procedure name, and a list of parameter names, respectively; type is an optional argument which, if supplied to either *$function* or *$cprocedure*, is an atom representing the function's or procedure's return value type (e.g., real*8, double, etc.); and stmt1 stmt2 ... stmtn is a sequence of zero or more LISP level translatable VAXIMA statements representing the body of a subprogram.

User Level Examples:

```
(c1) gentranlang(fortran)$

(c2) gentran( function("real", f(x,y)),
              body( x:x^2, y:y^2, f:abs(x-y) ) )$
      real function f(x,y)
      x=x**2
      y=y**2
      f=abs(x-y)
      return
      end

(c3) gentranlang(ratfor)$

(c4) gentran( subroutine(f(x,y)),
              body( y:abs(x^2-y^2) ) )$
subroutine f(x,y)
y=abs(x**2-y**2)
return
end

(c5) gentranlang(c)$
```

```
(c6) gentran( cprocedure("float", f(x,y)) )$
float f(x,y)
```

LISP Level Examples:

```
<1>: (setq *gentranlang 'fortran)
fortran

<1>: (gentran '( (($function) real ((f) x y))
                  (($body) ((msetq) x ((mexpt) x 2))
                           ((msetq) y ((mexpt) y 2))
                           ((msetq) f ((mabs)
                                        ((mplus) x ((mminus) y))))))
              ())
      real function f(x,y)
      x=x**2
      y=y**2
      f=abs(x-y)
      return
      end
t

<1>: (setq *gentranlang 'ratfor)
ratfor

<1>: (gentran '( (($subroutine) ((f) x y))
                  (($body) ((msetq) y
                                    ((mabs)
                                     ((mplus) ((mexpt) x 2)
                                              ((mminus) ((mexpt) y 2)))))))
              ())
subroutine f(x,y)
y=abs(x**2-y**2)
return
end
t

<1>: (setq *gentranlang 'c)
c

<1>: (gentran '( (($cprocedure) float ((f) x y)) ) ())
float f(x,y)
t
```

Other Special Arguments

The arguments described in this section allow numerical language statements to be generated which do not have semantically equivalent statements in the VAXIMA programming language.

User Level Syntax:

break()

stop()

end()

begin_group

end_group

LISP Level Syntax:

(($break))

(($stop))

(($end))

$begin_group

$end_group

User Level Examples:

```
(c1) gentranlang(fortran)$

(c2) gentran( for n:1 thru 100 do
                  if f(n)<0 then break() )$
      do 25001 n=1,100
          if (.not.f(n).lt.0.0) goto 25002
              goto 25003
25002     continue
25001 continue
25003 continue

(c3) gentran( stop(), end() )$
      stop
      end

(c4) gentranlang(ratfor)$

(c5) gentran( for n:1 thru 100 do
                  if f(n)<0 then break() )$
do n=1,100
    if (f(n)<0.0)
        break

(c6) gentranlang(c)$

(c7) gentran( begin_group,
              for n:1 thru 100 do
                  if f(n)<0 then break() )$
{
    for (n=1;!(n>100);n=n+1)
        fi(f(n)<0.0)
            break;
```

```
(c8) gentran( if n>100 then stop(),
              end_group )$
      if (n>100.0)
          exit(0);
}
```

LISP Level Examples:

```
<1>: (setq *gentranlang 'fortran)
fortran

<1>: (gentran '( ((mdo) n 1 nil nil 100 nil
                        ((mcond) ((mlessp) ((f) n) 0)
                                 (($break))
                                 t
                                 $false)) ) ())
      do 25001 n=1,100
          if (.not.f(n).lt.0.0) goto 25002
              goto 25003
25002     continue
25001 continue
25003 continue
t

<1>: (gentran '( (($stop)) (($end)) ) ())
      stop
      end
t

<1>: (setq *gentranlang 'ratfor)
ratfor

<1>: (gentran '( ((mdo) n 1 nil nil 100 nil
                        ((mcond) ((mlessp) ((f) n) 0)
                                 (($break))
                                 t
                                 (($false)))) ) ())
do n=1,100
    if (f(n)<0.0)
        break
t

<1>: (setq *gentranlang 'c)
c
```

```
<1>: (gentran '( $begin_group
                 ((mdo) n 1 nil nil 100 nil
                        ((mcond) ((mlessp) ((f) n) 0)
                                 (($break))
                                 t
                                 $false)) ) ())
{
    for (n=1;!(n>100);n=n+1)
        if (f(n)<0.0)
            break;
t

<1>: (gentran '( ((mcond) ((mgreaterp) n 100)
                          (($stop))
                          t
                          $false)
                 $end_group ) ())
    if (n>100.0)
        exit(0);
}
t
```

Translation is a convenient method of producing numerical code when the exact behavior of the resultant code is known. It gives the VAXIMA user who is familiar with the syntax of statements in the VAXIMA programming language the ability to write code in a numerical programming language without knowing the exact syntactical requirements of the language. However, the *real* power of the *gentran* command lies in its ability to generate code: it can produce numerical code from symbolic expressions derived in VAXIMA in addition to translating statements directly. This aspect is described in the next section.

2.3. Code Generation: Evaluation Prior to Translation

Section 2.2 showed how VAXIMA statements and expressions can be translated directly into the target language. This section shows how to indicate that parts of those statements and expressions are to be handed to VAXIMA to be evaluated before being translated. In other words, this section explains how to generate numerical code from algorithmic specifications (in the VAXIMA programming language) and symbolic expressions.

Each of the following four subsections describes a special function that can be used to request partial or full evaluation of expressions prior to translation. Note that these functions have the described effects *only* when used as arguments to the *gentran* function.

2.3.1. The eval Function

User Level Syntax:

*eval(*exp)

where exp is any VAXIMA user level expression or statement which, after evaluation by VAXIMA, results in an expression that can be translated by *gentran* into the target language.

LISP Level Syntax:

(($eval) exp)

where exp is any VAXIMA LISP level expression or statement which, after evaluation by VAXIMA, results in an expression that can be translated by *gentran* into the target language.

Side Effects:

When *eval* is called on an argument which is to be translated, it tells *gentran* to give the expression to VAXIMA for evaluation first, and then to translate the result of that evaluation.

User Level Example:

The following formula, f, has been derived symbolically:

```
      2
   2 x  - 5 x + 6
```

We wish to generate an assignment statement for the quotient of f and its derivative.

```
(c1) gentran( q : eval(f)/eval(diff(f,x)) );
      q=(6.0-5.0*x+2.0*x**2)/(-5.0+4.0*x)

(d1)                         true
```

LISP Level Example:

The following formula, f, has been derived symbolically:

```
     ((mplus) ((mtimes) 2 ((mexpt) x 2))
              ((mtimes) -5 x)
              6)
```

We wish to generate an assignment statement for the quotient of f and its derivative.

```
<1>: (gentran '(((msetq) q ((mquotient) (($eval) f)
                                        (($eval) (($diff) f x))))) ())
      q=(6.0-5.0*x+2.0*x**2)/(-5.0+4.0*x)
t
```

2.3.2. The rsetq Function

In many applications, assignments must be generated in which the left-hand side is some known variable name, but the right-hand side is an expression that must be evaluated. For this reason, a special function is provided to indicate that the expression on the right-hand side is to be evaluated prior to translation. This special function is *rsetq*.

User Level Syntax:

*rsetq(*var, exp)

where var is any VAXIMA user level variable, matrix or array element, and exp is any VAXIMA user level expression which, after evaluation by VAXIMA, results in an expression that can be translated by *gentran* into the target language.

LISP Level Syntax:

(($rsetq) var exp)

where var is any VAXIMA LISP level variable, matrix or array element, and exp is any VAXIMA LISP level expression which, after evaluation by VAXIMA, results in an expression that can be translated by *gentran* into the target language.

User Level Example:

```
(c1) gentran( rsetq( deriv, diff(x^4-x^3+2*x^2-1,x) ) );
      deriv=4.0*x-3.0*x**2+4.0*x**3

(d1)                       true
```

Note: This is equivalent to a call of the form

```
gentran( deriv : eval(diff(x^4-x^3+2*x^2-1,x)) );
```

LISP Level Example:

```
<1>: (gentran '((($rsetq) deriv
                          (($diff) ((mplus) ((mexpt) x 4)
                                            ((mminus) ((mexpt) x 3))
                                            ((mtimes) 2 ((mexpt) x 2))
                                            -1)
                                   x))) ())
      deriv=4.0*x-3.0*x**2+4.0*x**3
t
```

2.3.3. The lsetq Function

When assignments to matrix or array elements must be generated, many times the indices of the element must be evaluated first. The special function *lsetq* can be used within a call to *gentran* to indicate that the indices of the matrix or array element on the left-hand side of the assignment are to be evaluated prior to translation.

User Level Syntax:

*lsetq(*var, exp)

where var is any VAXIMA user level matrix or array element with indices which, after evaluation by VAXIMA, will result in expressions that can be translated by *gentran*, and exp is any VAXIMA user level expression that can be translated into the target language.

LISP Level Syntax:

(($lsetq) var exp)

where var is any VAXIMA LISP level matrix or array element with indices which, after evaluation by VAXIMA, result in expressions that can be translated by *gentran*, and exp is any VAXIMA LISP level expression that can be translated into the target language.

User Level Example:

We wish to generate assignments which assign zeros to all elements on the main diagonal of m, an n x n matrix.

```
(c1) for j:1 thru n do
         gentran( lsetq( m[j,j], 0.0 ) );
      m(1,1)=0.0
      m(2,2)=0.0
        .
        .
        .
      m(n,n)=0.0

(d1)            done
```

Note: This is equivalent to a call of the form

```
for j:1 thru n do
    gentran( m[eval(j),eval(j)] : 0.0 );
```

LISP Level Example:

We wish to generate assignments which assign zeros to all elements on the main diagonal of m, an n x n matrix.

```
<1>: (do ((j 1 (1+ j)))
         ((> j n))
         (gentran '((($lsetq) ((m) j j) 0.0)) ()))
       m(1,1)=0.0
       m(2,2)=0.0
         .
         .
         .
       m(n,n)=0.0
nil
```

2.3.4. The lrsetq Function

In applications in which evaluated expressions are to be assigned to matrix or array elements with evaluated subscripts, the *lrsetq* function can be used. It is a combination of the *lsetq* and *rsetq* functions described in sections 2.3.2 and 2.3.3.

User Level Syntax:

*lrsetq(*var, exp)

where var is any VAXIMA user level matrix or array element with indices which, after evaluation by VAXIMA, will result in expressions that can be translated by *gentran*; and exp is any user level expression which, after evaluation, will result in an expression that can be translated by *gentran* into the target language.

LISP Level Syntax:

(($lrsetq) var exp)

where var is any VAXIMA LISP level matrix or array element with indices which, after evaluation by VAXIMA, will result in expressions that can be translated by *gentran*; and exp is any LISP level expression which, after evaluation, will result in an expression that can be translated by *gentran* into the target language.

User Level Example:

The following matrix, m, has been derived symbolically:

```
[    a   0  - 1    1 ]
[                    ]
[    0   b    0    0 ]
[                    ]
[  - 1   0    c  - 1 ]
[                    ]
[    1   0  - 1    d ]
```

We wish to generate assignment statements for those elements on the main diagonal of the matrix.

```
(c1) for j:1 thru 4 do
          gentran( lrsetq( m[j,j], m[j,j] ) );
      m(1,1)=a
      m(2,2)=b
      m(3,3)=c
      m(4,4)=d

(d1)                           done
```

Note: This is equivalent to a call of the form

```
for j:1 thru 4 do
    gentran( m[eval(j),eval(j)] : eval(m[j,j]) );
```

LISP Level Example:

The following matrix, m, has been derived symbolically:

```
    [      a    0     1     1 ]
    [                         ]
    [      0    b     0     0 ]
    [                         ]
    [    - 1    0     c   - 1 ]
    [                         ]
    [      1    0   - 1     d ]
```

We wish to generate assignment statements for those elements on the main diagonal of the matrix.

```
<1>: (do ((j 1 (1+ j)))
         ((> j 4))
         (gentran '((($lrsetq) ((m array) j j)
                               ((m array) j j))) ()))
      m(1,1)=a
      m(2,2)=b
      m(3,3)=c
      m(4,4)=d
nil
```

2.4. Type Declarations

Type declarations are automatically generated each time a subprogram heading is generated. Type declarations are constructed from information stored in the GENTRAN symbol table. The user can place entries into the symbol table explicitly through calls to the special GENTRAN function *type*.

User Level Syntax:

*type(*type, v1,v2,...,vn);

where v1,v2,...,vn is one or more variables (optionally subscripted to indicate array dimensions), or variable ranges (two letters separated by "-" and enclosed in double quotes). v's are not evaluated unless given as arguments to *eval*. type is a variable type in the target language. It should be a string or an atom. type is not evaluated unless given as an argument to *eval*.

LISP Level Syntax:

(($type) type v1 v2 ... vn)

where v1 v2 ... vn is one or more variables (optionally in lists with array dimensions), or variable ranges (two letters separated by "-"). v's are not evaluated unless given as arguments to *$eval*. type is a variable type in the target language. It must be an atom. type is not evaluated unless given as an argument to *$eval*.

Side Effects:

Entries are placed in the symbol table for each variable or variable range declared in the call to this function. The function call itself is removed from the statement group being translated. Then after translation, type declarations are generated from these symbol table entries before the resulting executable statements are printed.

User Level Example:

```
(c1) gentran( type("implicit real*8", "a-h","o-z"),
              type("real*8", m(4,4)),
              for i:1 thru 4 do
                  for j:1 thru 4 do
                      if i=j
                         then m[i,j] : 1.0
                         else m[i,j] : 0.0,
              type("integer", i,j),
                .
                .
                .
            );

      implicit real*8 (a-h,o-z)
      real*8 m(4,4)
      integer i,j
      do 25001 i=1,4
          do 25002 j=1,4
              if (.not.i.eq.j) goto 25003
                  m(i,j)=1.0
                  goto 25004
25003         continue
                  m(i,j)=0.0
25004         continue
25002     continue
25001 continue
        .
        .
        .

(d1)                           true
```

LISP Level Example:

```
<1>: (gentran '((($type) |implicit real*8| a-h o-z)
                ((\$type) real*8 ((m) 4 4))
                ((mdo) i 1 nil nil 4 nil
                       ((mdo) j 1 nil nil 4 nil
                              ((mcond) ((mequal) i j)
                                       ((msetq) ((m) i j) 1.0)
                                       t
                                       ((msetq) ((m) i j) 0.0))))
                (($type) integer i j)
                  .
                  .
                  .
               ) ())
      implicit real*8 (a-h,o-z)
      real*8 m(4,4)
      integer i,j
      do 25001 i=1,4
          do 25002 j=1,4
              if (.not.i.eq.j) goto 25003
                  m(i,j)=1.0
                  goto 25004
25003         continue
                  m(i,j)=0.0
25004         continue
25002     continue
25001 continue
       .
       .
       .
t
```

2.4.1. Subprogram Type Declarations

The *type* function can also be used to declare subprogram types (i.e., subroutine or function) for FORTRAN and RATFOR code, and function types for FORTRAN, RATFOR, and C code.

User Level Examples:

```
(c1) gentranlang(ratfor)$
```

```
(c2) gentran( fac(n) := block( type(function, fac),
                               type(integer, fac,n),
                               f : 1,
                               for i:2 thru n do
                                   f : f*i,
                               type(integer, f,i),
                               return(f) ) );
integer function fac(n)
integer n,f,i
f=1
do i=2,n
    f=f*i
return(f)
end

(d2)                       true

(c3) gentranlang(c)$

(c4) gentran( fac(n) := block( type(int, fac,n,i,f),
                               f : 1,
                               for i:2 thru n do
                                   f : f*i,
                               return(f) ) );
int fac(n)
int n;
{
    int i,f;
    f=1;
    for (i=2;!(i>n);i=i+1)
        f=f*i;
    return(f);
}

(d4)                       done
```

LISP Level Examples:

```
<1>: (setq *gentranlang 'ratfor)
ratfor
```

```
<1>: (gentran '(((mdefine) ((fac) n)
                           ((mprog)
                            (($type) function fac)
                            (($type) integer fac n)
                            ((msetq) f 1)
                            ((mdo) i 2 nil nil n nil
                                   ((msetq) f ((mtimes) f i)))
                            (($type) integer f i)
                            ((mreturn) f)))) ())
integer function fac(n)
integer n,f,i
f=1
do i=2,n
    f=f*i
return(f)
end
t

<1>: (setq *gentranlang 'c)
c

<1>: (gentran '(((mdefine) ((fac) n)
                           ((mprog)
                            (($type) int fac n i f)
                            ((msetq) f 1)
                            ((mdo) i 2 nil nil n nil
                                   ((msetq) f ((mtimes) f i)))
                            ((mreturn) f)))) ())
int fac(n)
int n;
{
    int i,f;
    f=1;
    for (i=2;!(i>2);i=i+1)
        f=f*i;
    return(f);
}
t
```

2.4.2. A Note on Subscripted Variables

When generating code for subscripted variables (i.e., matrix and array elements), it is important to keep several things in mind. First of all, when a VAXIMA array is declared with a declaration such as

`array(a,` *n*`);`

where n is a positive integer, a is actually being declared to be of size n+1. Each of the elements a[0], a[1], ..., a[n] can be used. However, a FORTRAN or RATFOR declaration such as

`dimension a(`*n*`)`

only declares a to be of size n. Only the elements a(1), a(2), ..., a(n) can be used. Furthermore, a C declaration such as

`float a[`*n*`];`

declares a to be of size n with elements referred to as a[0], a[1], ..., a[n-1].

To resolve these array size and subscripting conflicts, the user should remember the following:

- All VAXIMA array subscripts are translated literally. Therefore it is the user's responsibility to be sure that array elements with subscript 0 are not translated into FORTRAN or RATFOR.
- Since C arrays allow elements with a subscript of 0, when an array is declared to be of size *n* by the user, the actual generated C type declaration will be of size n+1 so that the user can translate elements with subscripts from 0, up to and including *n*.

2.5. Comments and Literal Strings

Comments and other strings of characters can be inserted directly into the stream of generated code through a call to the special function *literal*.

User Level Syntax:

*literal(*arg1,arg2,...,argn)

where arg1,arg2,...,argn is an argument list containing one or more arg's, each of which either is, or evaluates to, an atom. The atoms tab and cr have special meanings. arg's are not evaluated unless given as arguments to *eval*.

LISP Level Syntax:

(($literal) arg1 arg2 ... argn)

where arg1 arg2 ... argn is an argument list containing one or more arg's, each of which either is, or evaluates to, an atom. The atoms $tab and $cr have special meanings. arg's are not evaluated unless given as arguments to *$eval*.

Side Effect:

This function call is replaced by the character sequence resulting from concatenation of the given atoms. Double quotes are stripped from all string type arg's, and each occurrence of the reserved atom tab or cr ($tab or $cr at the LISP level) is replaced by a tab to the current level of indentation, or an end-of-line character.

User Level Examples:

```
(c1) gentranlang(fortran)$
```

Suppose **n** has value 10.

```
(c2) gentran( literal( "c", tab, "--This is a FORTRAN Comment--", cr,
                       "c", cr ),
              literal( tab, "data ", "n/", eval(n), "/", cr ) );
c     --This is a FORTRAN Comment--
c
      data n/10/

(d2)                        true

(c3) gentranlang(ratfor)$
```

```
(c4) gentran( for i:1 thru n do
              (
                  literal( tab, "# --This is a RATFOR Comment--", cr ),
                  literal( tab, "write(6,10) (m(i,j),j=1,n)", cr ),
                  literal( 10, tab, "format(1x,10(i5,3x))", cr )
              ) );
do i=1,n
    {
        # --This is a RATFOR Comment--
        write(6,10) (m(i,j),j=1,n)
10      format(1x,10(i5,3x))
    }

(d4)                       true

(c5) gentranlang(c)$

(c6) gentran( x : 0.0,
              literal( tab, "/*", cr,
                       tab, " * This is a C Comment", cr,
                       tab, " */", cr ) );
x=0.0;
/*
 * This is a C Comment
 */

(d6)                       true
```

LISP Level Examples:

```
<1>: (setq *gentranlang 'fortran)
fortran
```

Suppose **n** has value 10.

```
<1>: (gentran '((($literal) |c| $tab |--This is a FORTRAN Comment--| $cr
                            |c| $cr)
                 (($literal) $tab |data | |n//| (($eval) n) |//| $cr)) ())
c     --This is a FORTRAN Comment--
c
      data n/10/
t

<1>: (setq *gentranlang 'ratfor)
ratfor
```

```
<1>: (gentran '(((mdo) i 1 nil nil n nil
                       ((mprogn)
                        (($literal) $tab |# --This is a RATFOR Comment--| $cr)
                        (($literal) $tab |write(6,10) (m(i,j),j=1,n)| $cr)
                        (($literal) 10 $tab |format(1x,10(i5,3x))| $cr)))) ())
do i=1,n
    {
        # --This is a RATFOR Comment--
        write(6,10) (m(i,j),j=1,n)
10      format(1x,10(i5,3x))
    }
t

<1>: (setq *gentranlang 'c)
c

<1>: (gentran '(((msetq) x 0.0)
                (($literal) $tab |//*| $cr
                            $tab | * This is a C Comment| $cr
                            $tab | *//| $cr)) ())
x=0.0;
/*
 * This is a C Comment
 */
t
```

2.6. The fortran, ratfor, and c Mode Switches

Sections 2.1 - 2.5 have shown how to produce numerical code from derived expressions and algorithmic specifications. First the variable **gentranlang* is set to *fortran*, *ratfor*, or *c*, and then the *gentran* function is called. Any of the special functions *eval*, *lsetq*, *rsetq*, *lrsetq*, *type*, and *literal* can be called from within a call to *gentran*. This section describes an alternative method of code generation.

Three mode switches are provided in GENTRAN to change the default mode of VAXIMA from evaluation to translation. These switches are *fortran*, *ratfor*, and *c*, and can be turned on and off with the special functions *on* and *off*. Each time a fresh VAXIMA session is started up, the system is in evaluation mode. It prints a prompt on the user's terminal screen and waits for an expression or statement to be entered. When it reads in a ; or $, the preceding expression is evaluated, a new prompt is printed, and the system waits for the user to enter another expression or statement. This mode can be changed to translation mode by turning on one of the following switches: *fortran*, *ratfor*, or *c*. After one of these switches is turned on, and until it is turned off, every expression or statement entered by the user is translated into the corresponding language just as if it had been given as an argument to the *gentran* function. Each of the special functions which can be used from within a call to *gentran* can be used at the top level until the switch is turned off. Thus, the *fortran*, *ratfor*, and *c* mode switches are an alternative method of code generation.

User Level Examples:

```
(c1) m : genmatrix(m, 2,2, 1,1)$
```

```
(c2) on(fortran);

     rsetq( minv, m^^(-1) );
      minv(1,1)=m(2,2)/(-m(1,2)*m(2,1)+m(1,1)*m(2,2))
      minv(1,2)=-m(1,2)/(-m(1,2)*m(2,1)+m(1,1)*m(2,2))
      minv(2,1)=-m(2,1)/(-m(1,2)*m(2,1)+m(1,1)*m(2,2))
      minv(2,2)=m(1,1)/(-m(1,2)*m(2,1)+m(1,1)*m(2,2))

     rsetq( d, determinant(m) );
      d=-m(1,2)*m(2,1)+m(1,1)*m(2,2)

     off(fortran);

(d2)                         done
```

LISP Level Examples:

```
<1>: (meval '((msetq) m (($genmatrix) m 2 2 1 1)) )
(($matrix simp) ((mlist simp) ((m simp array) 1 1) ((m simp array
) 1 2)) ((mlist simp) ((m simp array) 2 1) ((m simp array) 2 2)))

<1>: (on '(fortran))

     rsetq( minv, ?m^^(-1) );
      minv(1,1)=m(2,2)/(-m(1,2)*m(2,1)+m(1,1)*m(2,2))
      minv(1,2)=-m(1,2)/(-m(1,2)*m(2,1)+m(1,1)*m(2,2))
      minv(2,1)=-m(2,1)/(-m(1,2)*m(2,1)+m(1,1)*m(2,2))
      minv(2,2)=m(1,1)/(-m(1,2)*m(2,1)+m(1,1)*m(2,2))

     rsetq( d, determinant(?m) );
      d=-m(1,2)*m(2,1)+m(1,1)*m(2,2)

     off(fortran);
$done
```

2.7. Code Optimization

VAXIMA contains an *optimize* command which takes one expression as its argument. It searches for common subexpressions, replaces them by temporary variable names, and returns the resulting expression preceded by a sequence of assignment statements which assign the common subexpressions to the temporary variable names. GENTRAN has been interfaced with this command to make it possible to generate code in which the numerical expressions have been optimized. This interface is activated when the *gentranopt* flag is on. (*gentranopt* is off initially.) The following command will turn the *gentranopt* flag on:

> *on(gentranopt);*

and

> *off(gentranopt);*

will turn it back off again.

Example:

Suppose the following expression, ans, has been derived:

```
(c10) ans;
```

$$
\begin{aligned}
\text{(d10)}\ & - 36\, w^2 x^7 y^4 z^8 + 3\, w^2 x^6 y^3 z^8 - 24\, w^3 x^7 y^4 z^6 + 2\, w^3 x^6 y^3 z^6 \\
& + 96\, w^2 x^8 y^6 z^5 - 168\, w^4 x^7 y^6 z^5 + 12\, w^2 x^7 y^6 z^5 - 216\, w^2 x^{10} y^5 z^5 \\
& - 8\, w^2 x^7 y^5 z^5 + 9\, x^7 y^5 z^5 + 14\, w^4 x^6 y^5 z^5 - w^2 x^6 y^5 z^5 + 18\, w^2 x^9 y^4 z^5 \\
& + 87\, x^7 y^3 z^5 - 3\, w^2 x^6 y^3 z^5 + 6\, w\, x^7 y^5 z^3 + 58\, w\, x^7 y^3 z^3 - 2\, w^3 x^6 y^3 z^3 \\
& - 24\, x^8 y^7 z^2 + 42\, w^2 x^7 y^7 z^2 - 3\, x^7 y^7 z^2 + 54\, x^{10} y^6 z^2 - 232\, x^8 y^5 z^2 \\
& + 414\, w^2 x^7 y^5 z^2 - 29\, x^7 y^5 z^2 - 14\, w^4 x^6 y^5 z^2 + w^2 x^6 y^5 z^2 + 522\, x^{10} y^4 z^2 \\
& - 18\, w^2 x^9 y^4 z^2
\end{aligned}
$$

We wish to generate optimized FORTRAN code to compute this value.

The following VAXIMA session will generate the code:

```
(c11) on(gentranopt);

(d11)                          done

(c12) gentran( rsetq(ans,ans) );

      t0=w**2
      t1=x**9
      t2=y**4
      t3=z**2
      t4=x**10
      t5=x**6
      t6=y**5
      t7=w**4
      t8=x**7
      t9=x**8
      t10=y**6
      t11=y**7
      t12=w**3
      t13=y**3
      t14=z**3
      t15=z**5
      t16=z**6
      t17=z**8
```

```
      ans=-18.0*t0*t1*t2*t3+522.0*t4*t2*t3+t0*t5*t6*t3-14.0*t7*t5*t6*t3
     . -29.0*t8*t6*t3+414.0*t0*t8*t6*t3-232.0*t9*t6*t3+54.0*t4*t10*t3
     . -3.0*t8*t11*t3+42.0*t0*t8*t11*t3-24.0*t9*t11*t3-2.0*t12*t5*t13*
     . t14+58.0*w*t8*t13*t14+6.0*w*t8*t6*t14-3.0*t0*t5*t13*t15+87.0*t8*
     . t13*t15+18.0*t0*t1*t2*t15-t0*t5*t6*t15+14.0*t7*t5*t6*t15+9.0*t8*
     . t6*t15-8.0*t0*t8*t6*t15-216.0*t0*t4*t6*t15+12.0*t0*t8*t10*t15
     . -168.0*t7*t8*t10*t15+96.0*t0*t9*t10*t15+2.0*t12*t5*t13*t16-24.0*
     . t12*t8*t2*t16+3.0*t0*t5*t13*t17-36.0*t0*t8*t2*t17

(d12)                          true
```

Unfortunately, VAXIMA's *optimize* facility is not as powerful as it could be. It does not produce an absolutely optimal sequence of assignments to calculate an expression's value. For example, in the code above,

```
t3=z**2
  .
  .
  .
t14=z**3
t15=z**5
t16=z**6
t17=z**8
```

could be calculated more efficiently with

```
t3=z*z
  .
  .
  .
t14=t3*z
t15=t3*t14
t16=t15*z
t17=t16*t3
```

In addition, the *optimize* facility does not detect all possible common subexpressions. For example, in the above, t2*t3 appears in each of the first two terms of the final assignment statement. That expression should have been replaced by another temporary variable name, and another assignment statement should have been generated for it.

A code optimizer, designed by J. A. van Hulzen /14/ of Twente University of Technology in The Netherlands, has been implemented in RLISP to run under REDUCE. It utilizes an optimization technique in which one or more expressions are mapped onto matrices of coefficients and exponents. These matrices are searched for patterns corresponding to common subexpressions, all of which are replaced by temporary variable names. This optimization technique is much more powerful than the one used in VAXIMA's *optimize* command. A comparison of code generated using each of the two optimization facilities is given in sections 6.3.3 - 6.3.6.

2.8. Expression Segmentation

Symbolic derivations can easily produce formulas that can be anywhere from a few lines to several pages in length. Such formulas can be translated into numerical assignment statements, but unless they are broken into smaller pieces, they may be too long for a compiler to handle. (The maximum number of continuation lines for one statement allowed by most FORTRAN compilers is only 19.) Therefore GENTRAN contains a segmentation facility which automatically "segments", or breaks down, unreasonably large expressions.

The segmentation facility generates a sequence of assignment statements, each of which assigns a subexpression to an automatically generated temporary variable. This sequence is generated in such a way that temporary variables are re-used as soon as possible, thereby keeping the number of automatically generated variables to a minimum. The facility can be turned on or off by setting the mode switch *gentranseg* accordingly (i.e., by calling the function *on* or *off* on it). The user can control the maximum allowable expression size by setting the variable *?maxexpprintlen** to the maximum number of characters allowed in an expression printed in the target language (excluding spaces automatically printed by the formatter). The *gentranseg* switch is on initially, and *?maxexpprintlen** is initialized to 800.

User Level Examples:

```
(c1) on(gentranseg)$

(c2) ?maxexpprintlen\* : 120$

(c3) ( poly : 0,
       for i:0 thru 20 do
           poly : poly + c(i+1)*x^i )$

(c4) gentran( rsetq( poly, poly ) );
      t0=c(1)+c(2)*x+c(3)*x**2+c(4)*x**3+c(5)*x**4+c(6)*x**5+c(7)*x**6+c
     . (8)*x**7+c(9)*x**8+c(10)*x**9
      t0=t0+c(11)*x**10+c(12)*x**11+c(13)*x**12+c(14)*x**13+c(15)*x**14+
     . c(16)*x**15+c(17)*x**16+c(18)*x**17
      poly=t0+c(19)*x**18+c(20)*x**19+c(21)*x**20

(d4)                      true
```

LISP Level Examples:

```
<1>: (on '(gentranseg))
$done

<1>: (setq maxexpprintlen* 120)
120

<1>: (meval '((mprogn)
              ((msetq) poly 0)
              ((mdo) i 0 nil nil 20 nil
                     ((msetq) poly ((mplus) poly
                                           ((mtimes) ((c) ((mplus) i 1))
                                                     ((mexpt) x i)))) ) ) )
$done

<1>: (gentran '((($rsetq) poly poly)) ())
      t0=c(1)+c(2)*x+c(3)*x**2+c(4)*x**3+c(5)*x**4+c(6)*x**5+c(7)*x**6+c
     . (8)*x**7+c(9)*x**8+c(10)*x**9
      t0=t0+c(11)*x**10+c(12)*x**11+c(13)*x**12+c(14)*x**13+c(15)*x**14+
     . c(16)*x**15+c(17)*x**16+c(18)*x**17
      poly=t0+c(19)*x**18+c(20)*x**19+c(21)*x**20
t
```

2.8.1. Implicit Type Declarations

When the segmentation routine generates temporary variables, it places type declarations in the symbol table for those variables if possible. It uses the following rules to determine their type:

1) If the type of the variable to which the large expression is being assigned is already known (i.e., has been declared by the user), then the temporary variables will be declared to be of that same type.

2) If the global variable *?tempvartype** (*$tempvartype* at the LISP level) has a non-nil value, then the temporary variables are declared to be of that type.

3) Otherwise, the variables are not declared.

Example:

```
(c1) on(gentranseg)$

(c2) ?maxexpprintlen\* : 20$

(c3) ?tempvartype\* : real$

(c4) gentran( type(integer, isum),
               .
               .
              isum : i(1)+i(2)+i(3)+i(4)+i(5),
              pprod : p(1)*p(2)*p(3)*p(4)*p(5),
               .
               .
            );
      integer isum,t0
      real t1
       .
       .
      t0=i(1)+i(2)+i(3)+i(4)
      isum=t0+i(5)
      t1=p(1)*p(2)*p(3)*p(4)
      pprod=t1*p(5)
       .
       .

(d4)                       true
```

2.8.2. Controlling Generation of Declarations

In some applications, it may not be convenient to have type declarations generated automatically. The *gendecs* flag has been provided for this reason.

gendecs is on initially. As long as it stays on, all type declarations will be generated whenever possible. However, when *gendecs* is off, type declarations will not be generated. Instead, type information will be stored in GENTRAN's Symbol Table but will not be retrieved in the form of declarations unless and until either the *gendecs* function is called or the *gendecs* flag is turned back on.

The *gendecs* function can be called any time the *gendecs* flag is turned off to retrieve all type declarations from the Symbol Table for the given subprogram name (or the "current" subprogram if *false* or *nil* is given as its argument). The syntax for calls to this function is:

User Level Syntax:

*gendecs(*name);

where name is an atom.

LISP Level Syntax:

(gendecs 'name)

where name is an atom.

The *gendecs* flag can be turned on and off with the *on* and *off* functions:

User Level Syntax:

on(gendecs);

off(gendecs);

LISP Level Syntax:

(on '(gendecs))

(off '(gendecs))

The *gendecs* flag will is discussed in connection with template processing in section 3.2.

2.9. Generation of Temporary Variable Names

As we have just seen, GENTRAN's segmentation module generates temporary variables and places type declarations in the symbol table for them whenever possible. Various other modules also generate variables and corresponding declarations. All of these modules call one special GENTRAN function each time they need a temporary variable name. This function is *tempvar*. There are situations in which it may be convenient for the user to be able to generate temporary variable names directly[3]. Therefore *tempvar* is a user-accessible function.

User Level Syntax:

*tempvar(*type);

where type is either a string which indicates the variable type in the target language (e.g., "integer", "real*8", etc.), or is *false* if the variable type is unknown.

LISP Level Syntax:

(tempvar 'type)

where type is either an atom which indicates the variable type in the target language (e.g., integer, real*8, etc.), or is *nil* if the variable type is unknown.

Side Effects

tempvar creates temporary variable names by repeatedly concatenating the values of the global variables *?tempvarname** (*tempvarname**), which has a default value of '*t*, and *?tempvarnum** (*tempvarnum**), which is initially set to 0, and incrementing *?tempvarnum** until a variable name is created which satisfies one of the following conditions:

1) It had not been generated previously, and it has not been declared by the user.
2) It had previously been generated to hold the same type of value that it must hold this time (e.g., integer, real, etc.), and the value assigned to it previously is no longer needed.

If type is a non-*false* (non-*nil*) argument, or if type is *false* (*nil*) and the global variable *?tempvartype** (*tempvartype**), which is initially *false* (*nil*), has been set to a non-*false* (non-*nil*) value, then a type entry for the generated variable name is placed in the symbol table.

Returned Value:

tempvar returns an atom which can be used as a variable.

Note:

It is the user's responsibility to set *?tempvarname** and *?tempvarnum** (*tempvarnum**) to values such that generated variable names will not clash with variables used elsewhere in the program unless those variables have been

[3] One such example is suppression of the simplification process to generate numerical code which is more efficient. See the example in section 6.3.4.

declared.

2.9.1. Marking Temporary Variables

In section 2.9 we saw that a temporary variable name (of a certain type) can be regenerated when the value previously assigned to it is no longer needed. This section describes a function which "marks" a variable to indicate that it currently holds a significant value, and the next section describes functions which "unmark" variables to indicate that the values they hold are no longer significant.

User Level Syntax:

*markvar(*var);

where var is an atom.

LISP Level Syntax:

(markvar var)

where var is an atom.

Side Effect:

markvar sets a flag on var's property list to indicate that var currently holds a significant value.

Return Value:

markvar returns var.

User Level Example:

The following matrix, m, has been derived symbolically:

```
[ x  (z + y)          0      x  z ]
[                                 ]
[        - x      y + x         0 ]
[                                 ]
[                               2 ]
[        x  z         0        z  ]
```

We wish to replace each non-zero element by a generated variable name to prevent these expressions from being resubstituted into further calculations. (We will also record these substitutions in the numerical program we are constructing by generating assignment statements.)

```
(c1) for i:1 thru 3 do
         for j:1 thru 3 do
             if m[i,j]#0 then
             (
                 var : tempvar(false),
                 markvar(var),
                 gentran( eval(var) : eval(m[i,j]) ),
                 m[i,j] : var
             );

      t0=x*(y+z)
      t1=x*z
      t2=-x
      t3=x+y
      t4=x*z
      t5=z**2
```

```
(d1)                          done
```

Now matrix m contains the following entries:

```
[ t0     0    t1 ]
[                ]
[ t2    t3     0 ]
[                ]
[ t4     0    t5 ]
```

LISP Level Example:

The following matrix, m, has been derived symbolically:

```
[ x (z + y)         0      x z ]
[                              ]
[       - x      y + x       0 ]
[                              ]
[                            2 ]
[       x z         0       z  ]
```

We wish to replace each non-zero element by a generated variable name to prevent these expressions from being resubstituted into further calculations. (We will also record these substitutions in the numerical program we are constructing by generating assignment statements.)

```
<1>: (meval '((mdo) i 1 nil nil 3 nil
                    ((mdo) j 1 nil nil 3 nil
                           ((mcond) ((mnotequal) ((m array) i j) 0)
                                    ((mprogn)
                                     ((msetq) var ((tempvar) nil))
                                     ((markvar) var)
                                     ((gentran)
                                      '(((msetq) (($eval) var)
                                                 (($eval) ((m array) i j)))) ())
                                     ((msetq) ((m array) i j) var))
                                    t
                                    $false))))

      t0=x*(y+z)
      t1=x*z
      t2=-x
      t3=x+y
      t4=x*z
      t5=z**2

$done
```

2.9.2. Unmarking Temporary Variables

After the value assigned to a temporary variable has been used in the numerical program and is no longer needed, the variable name can be "unmarked" with the *unmarkvar* function.

User Level Syntax:

*unmarkvar(*var);

where var is an atom.

LISP Level Syntax:

(unmarkvar var)

where var is an atom.

Side Effect:

unmarkvar resets a flag on var's property list to indicate that var does not hold a significant value.

The *recurunmark* function unmarks all variable names in an expression.

User Level Syntax:

*recurunmark(*exp);

where exp is an expression containing one or more variable names.

LISP Level Syntax:

(recurunmark exp)

where exp is an expression containing one or more variable names.

Side Effect:

recurunmark resets flags on the property lists of all variable names in exp to indicate that they do not hold significant values any longer.

2.9.3. The Marked Variable Predicate

A variable name can be tested to see if it is currently marked with the marked variable predicate.

User Level Syntax:

*markedvarp(*var);

where var is an atom.

LISP Level Syntax:

*(markedvarp '*var)

where var is an atom.

Return Value:

markedvarp returns *true* or *false* (*t* or *nil* at the LISP level), depending on whether or not var is currently marked.

2.10. Generation of Statement Numbers

The *genstmtno* function is called by GENTRAN whenever a new statement number must be generated. The syntax is as follows:

User Level Syntax:

genstmtno();

LISP Level Syntax:

(genstmtno)

Side Effects:

genstmtno increases the value of global variable _?genstmtno_* (_genstmtno*_) by _?genstmtincr_* (_genstmtincr*_) when it is called. (_?genstmtno_* is initialized to 25000, and _?genstmtincr_* is initialized to 1.)

Return Value: _genstmtno_ returns the new value of _?genstmtno_* (_genstmtno*_).

3. TEMPLATE PROCESSING

In some code generation applications, pieces of the target numerical program are known in advance. A "template" file containing a program outline is supplied by the user, and formulas are derived in VAXIMA, converted to numerical code, and inserted in the corresponding places in the program outline to form a complete numerical program. A template processor is provided by GENTRAN for use in these applications.

3.1. The Template Processing Command

User Level Syntax:

*gentranin(*f1,f2,...,fn {,[f1,f2,...,fm]});

where f1,f2,...,fn is an argument list containing one or more f's, each of which is one of the following:

`a string  =  a template (input) file`
true = the terminal

and [f1,f2,...,fm] is an optional argument list which, if supplied, contains one or more f's, each of which is one of the following:

`a string  =  an output file`
true = the terminal
false = the current output file(s)
all = all files currently open for output by GENTRAN
(see section 4)

LISP Level Syntax:

*(gentranin '(*f1 f2 ... fn) '(f1 f2 ... fm))

where f1 f2 ... fn is an argument list containing one or more f's, each of which is one of the following:

`a string  =  a template (input) file`
t = the terminal

and (f1 f2 ... fm) is an argument list which contains zero or more f's, each of which is one of the following:

`a string  =  an output file`
t = the terminal
nil = the current output file(s)
$all = all files currently open for output by GENTRAN
(see section 4)

Side Effects:

gentranin processes each template file f1,f2,...,fn sequentially.

A template file may contain any number of parts, each of which is either an "active" or an "inactive" part. Each active part starts with the delimiter << and ends with >>.

Inactive parts of template files are assumed to contain code in the target language, FORTRAN, RATFOR, or C, depending on the value of the global variable **gentranlang*. Each inactive part is copied to the output. Each comment delimited by the appropriate characters,

```
C  ... <cr>    for FORTRAN (beginning in column 1) or
*  ... <cr>    for FORTRAN (beginning in column 1),
#  ... <cr>    for RATFOR, and
/* ...  */     for C
```

is also copied in its entirety to the output. Thus the character sequences << and >> have no special meanings within comments.

Active parts may contain any number of VAXIMA expressions and statements. They are not copied directly to the output. Instead, they are given to VAXIMA for evaluation *at the user level*. All output generated by each evaluation is sent to the output file(s). Returned values are only printed on the terminal.

Active parts will most likely contain calls to *gentran* to generate code. This means that the result of processing a template file will be the original template file with all active parts replaced by generated code.

If [f1,f2,...,fm] ((f1 f2 ... fn)) is empty, then generated code is simply written to the current output file(s).

However, if it is given, then the current output file is temporarily overridden. Generated code is written to each file represented by f1,f2,...,fn for this command only. Files which were open prior to the call to *gentranin* will remain open after the call, and files which did not exist prior to the call will be created, opened, written to, and closed. The output file stack will be exactly the same both before and after the call.

Return Value:

gentranin returns the names of all files written to by this command.

Diagnostic Messages:

Nonexistent Input File

Template File Already Open for Input

Wrong Type of Arg

Example:

Suppose we wish to generate a FORTRAN subprogram to compute the determinant of a 3 x 3 matrix. We can construct a template file with an outline of the FORTRAN subprogram and VAXIMA and GENTRAN commands to fill it in:

```
Contents of file det.tem:
+----------------------------------------------+
|      real*8 function det(m)                  |
|      real*8 m(3,3)                           |
|<<                                            |
|      m : genmatrix(m, 3,3, 1,1)$             |
|      gentran( rsetq( det, determinant(m) ) )$|
|>>                                            |
|      return                                  |
|      end                                     |
+----------------------------------------------+
```

Now we can generate a FORTRAN subprogram with the following VAXIMA session:

```
(c1) gentranlang(fortran)$

(c2) gentranin("det.tem", ["det.f"]);

(d2)                        det.f
```

```
Contents of file det.f:
+-------------------------------------------------------------------------+
|      real*8 function det(m)                                             |
|      real*8 m(3,3)                                                      |
|      det=m(1,3)*(-m(2,2)*m(3,1)+m(2,1)*m(3,2))-m(1,2)*(-m(2,3)*m(3,1)+m|
|     . (2,1)*m(3,3))+m(1,1)*(-m(2,3)*m(3,2)+m(2,2)*m(3,3))               |
|      return                                                             |
|      end                                                                |
+-------------------------------------------------------------------------+
```

3.1.1. Copying Files into Template Files

Template files can be copied into other template files with recursive calls to *gentranin*; i.e., by calling *gentranin* from the active part of a template file.

For example, suppose we wish to copy the contents of the file det.f, generated in the previous section, into a file containing a main program. The template file for the main program, main.tem, will contain an active part to copy the contents of det.f:

```
Contents of file main.tem:
+-----------------------------------------+
|c                                        |
|c --- Main Program ---                   |
|c                                        |
|      real*8 m(3,3),det                  |
|      write(6,*) 'Enter 3 x 3 Matrix:'   |
|      do 100 i=1,3                       |
|          read(5,*) (m(i,j),j=1,3)       |
|100   continue                           |
|      write(6,*) 'Determinant = ',det(m)|
|      stop                               |
|      end                                |
|c                                        |
|c --- Determinant Calculation ---        |
|c                                        |
|<<                                       |
|      gentranin("det.f")$                |
|>>                                       |
+-----------------------------------------+
```

The following VAXIMA session will create the file main.f:

```
(c1) gentranin("main.tem", ["main.f"]);

(d1)                           main.f
```

```
Contents of file main.f:
+------------------------------------------------------------------------+
|c                                                                       |
|c --- Main Program ---                                                  |
|c                                                                       |
|      real*8 m(3,3),det                                                 |
|      write(6,*) 'Enter 3 x 3 Matrix:'                                  |
|      do 100 i=1,3                                                      |
|          read(5,*) (m(i,j),j=1,3)                                      |
|100   continue                                                          |
|      write(6,*) 'Determinant = ',det(m)                                |
|      stop                                                              |
|      end                                                               |

|c                                                                       |
|c --- Determinant Calculation ---                                       |
|c                                                                       |
|      real*8 function det(m)                                            |
|      real*8 m(3,3)                                                     |
|      det=m(1,3)*(-m(2,2)*m(3,1)+m(2,1)*m(3,2))-m(1,2)*(-m(2,3)*m(3,1)+m|
|     . (2,1)*m(3,3))+m(1,1)*(-m(2,3)*m(3,2)+m(2,2)*m(3,3))              |
|      return                                                            |
|      end                                                               |
+------------------------------------------------------------------------+
```

3.1.2. The Template File Stack

The VAXIMA *batch* command takes one file name as its argument. VAXIMA reads in the file and executes all statements and commands, any of which may be another *batch* command. A stack of input file names is maintained by VAXIMA to allow recursive invocation of the *batch* command. Similarly, a stack of template file names is maintained by GENTRAN to facilitate recursive invocation of the template processor. Section 3.2 showed that the *gentranin* command can be called recursively to copy files into other files. This section shows that template files which are copied into other template files can also contain active parts, and thus, the whole code generation process can be invoked recursively.

We can generalize the example of section 3.2 by generating code recursively. We can extend it to generate code which will compute entries of the inverse matrix, also. Suppose we have created the file init.in, which contains a VAXIMA command to create an n x n matrix, m, and initialize its entries to m(1,1), m(1,2), ..., m(n,n), for some user-entered value n:

```
Contents of file init.in:
+---------------------------+
|m : genmatrix(m, n,n, 1,1)$|
+---------------------------+
```

We have also created template files det.tem and inv.tem which contain outlines of FORTRAN subprograms to compute the determinant and inverse of an n x n matrix, m, respectively:

Contents of file det.tem:

```
+------------------------------------------------------+
|      real*8 function det(m)                          |
|<<                                                    |
|      gentran( type("real*8", m(eval(n),eval(n)) ),   |
|               rsetq( det, determinant(m) )       )$  |
|>>                                                    |
|      return                                          |
|      end                                             |
+------------------------------------------------------+
```

Contents of file inv.tem:

```
+------------------------------------------------------+
|      subroutine inv(m,minv)                          |
|<<                                                    |
|      gentran( type("real*8", m(eval(n),eval(n)),     |
|                              minv(eval(n),eval(n))), |
|               rsetq( minv, m^^(-1) )               )$|
|>>                                                    |
|      return                                          |
|      end                                             |
+------------------------------------------------------+
```

Now we can construct a template file with a generalized version of the main program given in section 3.2 and can place *gentranin* commands in this file to generate code recursively from the template files det.tem and inv.tem:

Contents of file main.tem:

```
+------------------------------------------------------------------+
|c                                                                 |
|c  Main Program                                                   |
|c                                                                 |
|<<                                                                |
|      gentran( type("real*8", m(eval(n),eval(n)),                 |
|                              det,                                |
|                              minv(eval(n),eval(n)) ),            |
|               type("integer", n),                                |
|               literal(tab, "data n/", eval(n), "/", cr) )$       |
|>>                                                                |
|      write(6,*) 'Enter ',n,' x ',n,' Matrix:'                    |
|      do 100 i=1,n                                                |
|          read(5,*) (m(i,j),j=1,n)                                |
|100   continue                                                    |
|      write(6,*) 'Determinant = ',det(m)                          |
|      write(6,*) 'Inverse Matrix:'                                |
|      call inv(m,minv)                                            |
|      do 200 i=1,n                                                |
|          write(6,*) (minv(i,j),j=1,n)                            |
|200   continue                                                    |
|      stop                                                        |
|      end                                                         |
```

```
|c                                                                  |
|c  Determinant Calculation                                         |
|c                                                                  |
|<<                                                                 |
|      gentranin("det.tem")$                                        |
|>>                                                                 |
|c                                                                  |
|c  Inverse Calculation                                             |
|c                                                                  |
|<<                                                                 |
|      gentranin("inv.tem")$                                        |
|>>                                                                 |
+-------------------------------------------------------------------+
```

The following VAXIMA session will create the file main.f:

```
(c1) n : 3$

(c2) batch("init.in");

(c3) m : genmatrix(m, n,n, 1,1)$

(d4)                      BATCH DONE

(c5) gentranlang(fortran)$

(c6) gentranin("main.tem", ["main.f"]);

(d6)                      main.f

Contents of file main.f:
+------------------------------------------------------------------------+
|c                                                                       |
|c  Main Program                                                         |
|c                                                                       |
```

```
      real*8 m(3,3),det,minv(3,3)
      integer n
      data n/3/
      write(6,*) 'Enter ',n,' x ',n,' Matrix:'
      do 100 i=1,n
          read(5,*) (m(i,j),j=1,n)
100   continue
      write(6,*) 'Determinant = ',det(m)
      write(6,*) 'Inverse Matrix:'
      call inv(m,minv)
      do 200 i=1,n
          write(6,*) (minv(i,j),j=1,n)
200   continue
      stop
      end
c
c  Determinant Calculation
c
      real*8 function det(m)
      real*8 m(3,3)
      det=m(1,3)*(-m(2,2)*m(3,1)+m(2,1)*m(3,2))-m(1,2)*(-m(2,3)*m(3,1)+m
     . (2,1)*m(3,3))+m(1,1)*(-m(2,3)*m(3,2)+m(2,2)*m(3,3))
      return
      end
c
c  Inverse Calculation
c
      subroutine inv(m,minv)
      real*8 m(3,3),minv(3,3)
      minv(1,1)=(-m(2,3)*m(3,2)+m(2,2)*m(3,3))/((-m(1,3)*m(2,2)+m(1,2)*m
     . (2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,1
     . )+m(1,1)*m(2,2))*m(3,3))
      minv(1,2)=-(-m(1,3)*m(3,2)+m(1,2)*m(3,3))/((-m(1,3)*m(2,2)+m(1,2)*
     . m(2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,
     . 1)+m(1,1)*m(2,2))*m(3,3))
      minv(1,3)=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))/((-m(1,3)*m(2,2)+m(1,2)*m
     . (2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,1
     . )+m(1,1)*m(2,2))*m(3,3))
      minv(2,1)=-(-m(2,3)*m(3,1)+m(2,1)*m(3,3))/((-m(1,3)*m(2,2)+m(1,2)*
     . m(2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,
     . 1)+m(1,1)*m(2,2))*m(3,3))
      minv(2,2)=(-m(1,3)*m(3,1)+m(1,1)*m(3,3))/((-m(1,3)*m(2,2)+m(1,2)*m
     . (2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,1
     . )+m(1,1)*m(2,2))*m(3,3))
      minv(2,3)=-(-m(1,3)*m(2,1)+m(1,1)*m(2,3))/((-m(1,3)*m(2,2)+m(1,2)*
     . m(2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,
     . 1)+m(1,1)*m(2,2))*m(3,3))
      minv(3,1)=(-m(2,2)*m(3,1)+m(2,1)*m(3,2))/((-m(1,3)*m(2,2)+m(1,2)*m
     . (2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,1
     . )+m(1,1)*m(2,2))*m(3,3))
      minv(3,2)=-(-m(1,2)*m(3,1)+m(1,1)*m(3,2))/((-m(1,3)*m(2,2)+m(1,2)*
     . m(2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,
     . 1)+m(1,1)*m(2,2))*m(3,3))
      minv(3,3)=(-m(1,2)*m(2,1)+m(1,1)*m(2,2))/((-m(1,3)*m(2,2)+m(1,2)*m
```

```
|      . (2,3))*m(3,1)+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)+(-m(1,2)*m(2,1|
|      . )+m(1,1)*m(2,2))*m(3,3))                                         |
|       return                                                            |
|       end                                                               |
+-------------------------------------------------------------------------+
```

This is an example of a modular approach to code generation; separate subprogram templates are given in separate files. Furthermore, the template files are general; they can be used for matrices of any predetermined size. Therefore, we can easily generate different subprograms to handle matrices of different sizes from the same template files simply by assigning different values to n, and reloading the file init.in.

3.2. Insertion of Generated Type Declarations

The *gendecs* flag makes it possible to generate declarations for variables generated from within active parts of template files.

Example:

Suppose we wish to process a template file which will cause GENTRAN to generate one or more temporary variables in an active part. We can tell GENTRAN to generate declarations in the correct location by processing the template file twice as follows:

Construct the template file as usual, but with two additions:

1) Insert an active part at the beginning of the file to turn the *gendecs* flag off:

```
<<  off(gendecs)$  >>
```

2) Insert an active part immediately after the subprogram heading to write an active part to the output file. This active part will cause the *gendecs* flag to be switched back on during the second pass over the template file:

```
<<  gentran( literal("<<  on(gendecs)\$  \>\>", cr) )$  >>
```

For example, suppose the original template file contains the following:

```
Contents of file detinv.tem:
+-------------------------------------------------------------+
|       subroutine detinv(m,det,inv)                          |
|<<                                                           |
|       off(gendecs)$                                         |
|       gentran( literal("<<    on(gendecs)\$    \>\>", cr) )$|
|>>                                                           |
|<<                                                           |
|       m : genmatrix(m, n,n, 1,1)$                           |
|       gentran( type("real*8", m(eval(n),eval(n))) )$        |
|>>                                                           |
|<<                                                           |
|       gentran( type("real*8", det),                         |
|                rsetq( det, determinant(m) ) )$              |
|>>                                                           |
|<<                                                           |
|       gentran( type("real*8", inv(eval(n),eval(n))),        |
|                rsetq( inv, m^^(-1) ) )$                     |
|>>                                                           |
|       return                                                |
|       end                                                   |
+-------------------------------------------------------------+
```

Now we are ready to invoke the template processor. Since it must make two passes over the template file, we will invoke it twice and have it write to a temporary file as follows:

```
(c1) n : 3$

(c2) ?maxexpprintlen: 60$

(c3) gentranin("detinv.tem", ["##detinv"]);

(d3)                        ##detinv

(c4) gentranin("##detinv", ["detinv.f"]);

(d4)                        detinv.f
```

After the first template processing command has been executed, ##detinv contains the following:

```
Contents of file ##detinv:
+------------------------------------------------------------+
|       subroutine detinv(m,det,inv)                         |
|<<     on(gendecs)$    >>                                   |
|       t0=m(1,3)*(-m(2,2)*m(3,1)+m(2,1)*m(3,2))             |
|       t0=t0-m(1,2)*(-m(2,3)*m(3,1)+m(2,1)*m(3,3))          |
|       det=t0+m(1,1)*(-m(2,3)*m(3,2)+m(2,2)*m(3,3))         |
|       t1=-m(2,3)*m(3,2)+m(2,2)*m(3,3)                      |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(1,1)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-(-m(1,3)*m(3,2)+m(1,2)*m(3,3))                   |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(1,2)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-m(1,3)*m(2,2)+m(1,2)*m(2,3)                      |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(1,3)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-(-m(2,3)*m(3,1)+m(2,1)*m(3,3))                   |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(2,1)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-m(1,3)*m(3,1)+m(1,1)*m(3,3)                      |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(2,2)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-(-m(1,3)*m(2,1)+m(1,1)*m(2,3))                   |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(2,3)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-m(2,2)*m(3,1)+m(2,1)*m(3,2)                      |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(3,1)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-(-m(1,2)*m(3,1)+m(1,1)*m(3,2))                   |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(3,2)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-m(1,2)*m(2,1)+m(1,1)*m(2,2)                      |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)             |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)           |
|       inv(3,3)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       return                                               |
|       end                                                  |
+------------------------------------------------------------+
```

This is a temporary file. It can be removed after the second *gentranin* command has been executed. The final output file will contain the following:

Contents of file detinv.f:

```
+-------------------------------------------------------------+
|       subroutine detinv(m,det,inv)                          |
|       real*8 m(3,3),det,inv(3,3),t0,t1                      |
|       t0=m(1,3)*(-m(2,2)*m(3,1)+m(2,1)*m(3,2))              |
|       t0=t0-m(1,2)*(-m(2,3)*m(3,1)+m(2,1)*m(3,3))           |
|       det=t0+m(1,1)*(-m(2,3)*m(3,2)+m(2,2)*m(3,3))          |
|       t1=-m(2,3)*m(3,2)+m(2,2)*m(3,3)                       |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(1,1)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-(-m(1,3)*m(3,2)+m(1,2)*m(3,3))                    |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(1,2)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-m(1,3)*m(2,2)+m(1,2)*m(2,3)                       |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(1,3)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-(-m(2,3)*m(3,1)+m(2,1)*m(3,3))                    |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(2,1)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-m(1,3)*m(3,1)+m(1,1)*m(3,3)                       |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(2,2)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-(-m(1,3)*m(2,1)+m(1,1)*m(2,3))                    |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(2,3)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-m(2,2)*m(3,1)+m(2,1)*m(3,2)                       |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(3,1)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-(-m(1,2)*m(3,1)+m(1,1)*m(3,2))                    |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(3,2)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       t1=-m(1,2)*m(2,1)+m(1,1)*m(2,2)                       |
|       t0=(-m(1,3)*m(2,2)+m(1,2)*m(2,3))*m(3,1)              |
|       t0=t0+(m(1,3)*m(2,1)-m(1,1)*m(2,3))*m(3,2)            |
|       inv(3,3)=t1/(t0+(-m(1,2)*m(2,1)+m(1,1)*m(2,2))*m(3,3))|
|       return                                                |
|       end                                                   |
+-------------------------------------------------------------+
```

3.3. Special Template Variables

** Not Implemented Yet **

4. OUTPUT REDIRECTION

Most examples given thus far in this manual have sent all generated code to the terminal screen. In actual code generation applications, however, code must be sent to a file which will be compiled at a later time. This section explains methods of redirecting code to a file as it is generated. Any number of output files can be open simultaneously, and generated code can be sent to any combination of these open files.

4.1. File Selection Commands

VAXIMA provides the user with two file handling commands for output redirection: *writefile* and *closefile*. The *writefile* command takes a single file name as argument and directs all VAXIMA output to that file from then on, until another *writefile* changes the output file, or *closefile* explicitly closes it. Output can go to only one file at a time, and only one file can be open at any given time. Any existing file is erased before its first use for output in a job.

If an output file is closed with a *closefile* command, and a further *writefile* command is issued for that same file, the contents of the file are erased before the new output is written. When an output file is closed with the *closefile* command, output is switched to the terminal.

These commands are suitable for most applications in which VAXIMA output must be saved. However, they have two deficiencies when considered for use in code generation applications. First, they are inconvenient. *writefile* tells VAXIMA to direct *all* output to the specified file. Thus, in addition to output written as side effects of function calls, returned values are also written to the file. Secondly, the *writefile* command does not allow output to be sent to two or more files without reissuing the *writefile* with another file name. In an effort to remove these deficiencies and make the code generation commands flexible and easy to use, separate file handling commands are provided by GENTRAN which redirect generated code *only*.

The *gentranout* and *gentranshut* commands are identical to the VAXIMA *writefile* and *closefile* commands with the following exceptions:

- *gentranout* and *gentranshut* redirect *only* code which is printed as a side effect of GENTRAN commands.
- *gentranout* allows more than one file name to be given to indicate that generated code is to be sent to two or more files. (It is particularly convenient to be able to have generated code sent to the terminal screen and one or more files simultaneously.)
- *gentranout* does not automatically erase existing files; instead, it appends output onto the end of files.

The next two subsections describe these commands in detail.

4.1.1. gentranout

User Level Syntax:

*gentranout(*f1,f2,...,fn);

where f1,f2,...,fn is a list of one or more f's, and each f is one of:

```
a string  =  an output file
```
true = the terminal
false = the current output file(s)
all = all files currently open for output by GENTRAN

LISP Level Syntax:

*(gentranout '(*f1 f2 ... fn))

where f1 f2 ... fn is a sequence of one or more f's, and each f is one of:

```
an atom  =  an output file
```
t = the terminal
nil = the current output file(s)
$all = all files currently open for output by GENTRAN

Side Effects:

GENTRAN maintains a list of files currently open for output by GENTRAN commands *only*. *gentranout* inserts each file name represented by f1,f2,...,fn into that list and opens each one for output. It also resets the current output file(s) to be all files in f1,f2,...,fn.

Return Value:

gentranout returns the list of files represented by f1,f2,...,fn; i.e., the current output file(s) after the command has been executed.

Diagnostic Messages:

Wrong Type of Arg

Examples:

```
Output file list:

  current-output
  |
  v
+------+
| t    |
+------+

(c1) gentranout("f1");

(d1)                       f1

Output file list:

         current-output
         |
         v
+------+------+
| t    | "f1" |
+------+------+

(c2) gentranout("f2");

(d2)                       f2

Output file list:

                current-output
                |
                v
+------+------+------+
| t    | "f1" | "f2" |
+------+------+------+
```

```
(c3) gentranout(true,"f3");

(d3)                        [true,f3]

Output file list:

  +--------------------+------current-output
  |                    |
  v                    v
+------+------+------+------+
| t    | "f1" | "f2" | "f3" |
+------+------+------+------+

(c4) gentranout("f1");

(d4)                        f1

Output file list:

         current-output
         |
         v
+------+------+------+------+
| t    | "f1" | "f2" | "f3" |
+------+------+------+------+
```

4.1.2. gentranshut

User Level Syntax:

*gentranshut(*f1,f2,...,fn);

where f1,f2,...,fn is a list of one or more f's, and each f is one of:

`a string  =  an output file`
false = the current output file(s)
all = all files currently open for output by GENTRAN

LISP Level Syntax:

*(gentranshut '(*f1 f2 ... fn))

where f1 f2 ... fn is a list of one or more f's, and each f is one of:

`an atom  =  an output file`
nil = the current output file(s)
$all = all files currently open for output by GENTRAN

Side Effects:

gentranshut creates a list of file names from f1,f2,...,fn, deletes each from the output file list, and closes the corresponding files. If (all of) the current output file(s) are closed, then the current output file is reset to the terminal.

Return Value:

gentranshut returns (a list of) the current output file(s) after the command has been executed.

Diagnostic Messages:

File Not Open for Output

Wrong Type of Arg

User Level Examples:

```
Output file list:

  current-output ------+------+-------------------+
                       |      |                   |
                       v      v                   v
+------+------+------+------+------+------+------+------+
| t    | "f1" | "f2" | "f3" | "f4" | "f5" | "f6" | "f7" |
+------+------+------+------+------+------+------+------+

(c1) gentranshut("f1","f2","f7");

(d1)                        [f3,f4]

Output file list:

         +------+----------- current-output
         |      |
         v      v
+------+------+------+------+------+
| t    | "f3" | "f4" | "f5" | "f6" |
+------+------+------+------+------+

(c2) gentranshut(false);

(d2)                        true

Output file list:

  current-output
  |
  v
+------+------+------+
| t    | "f5" | "f6" |
+------+------+------+

(c3) gentranshut(all);

(d3)                        true

Output file list:

  current-output
  |
  v
+------+
| t    |
+------+
```

4.2. Operations on the Output File Stack

Section 4.1 explained the *gentranout* and *gentranshut* commands which are very similar to the VAXIMA *writefile* and *closefile* commands, but redirect only code generated as side effects of GENTRAN commands to files. This section describes another pair of file handling commands provided by GENTRAN.

In some code generation applications it may be convenient to be able to send generated code to one (set of) file(s), then temporarily send code to another (set of) file(s), and later resume sending generated code to the first (set of) file(s). In other words, it is convenient to think of the output files as being arranged in a stack which can be pushed whenever new files are to be written to temporarily, and popped whenever previously written-to files are to be appended onto. *gentranpush* and *gentranpop* enable the user to manipulate a stack of open output files in these ways.

gentranpush simply pushes a (set of) file(s) onto the stack and opens each one that is not already open for output. *gentranpop* deletes the top-most occurrence of the given file(s) from the stack and closes each one that is no longer in the stack. The stack is initialized to one element: the terminal. This element is always on the bottom of the stack, and thus, is the default output file. The current output file is always the file(s) on top of the stack.

4.2.1. gentranpush

User Level Syntax:

*gentranpush(*f1,f2,...,fn);

where f1,f2,...,fn is a list of one or more f's, and each f is one of:

```
a string  =  an output file
```
true = the terminal
false = the current output file(s)
all = all files currently open for output by GENTRAN

LISP Level Syntax:

*(gentranpush '(*f1 f2 ... fn))

where f1 f2 ... fn is a list of one or more f's, and each f is one of:

```
an atom  =  an output file
```
t = the terminal
nil = the current output file(s)
$all = all files currently open for output by GENTRAN

Side Effects:

gentranpush creates a list of file name(s) represented by f1,f2,...,fn and pushes that list onto the output stack. Each file in the list that is not already open for output is opened at this time. The current output file is reset to this new element on the top of the stack.

Return Value:

gentranpush returns the list of files represented by f1,f2,...,fn; i.e., the current output file(s) after the command has been executed.

Diagnostic Messages:

Wrong Type of Arg

User Level Examples:

```
Output stack:

+-------------+
| t           |  <-- current-output
+-------------+

(c1) gentranpush("f1");

(d1)                        f1

Output stack:

+-------------+
| "f1"        |  <-- current-output
| t           |
+-------------+

(c2) gentranpush("f2","f3");

(d2)                        [f2,f3]

Output stack:

+-------------+
| "f2" "f3"   |  <-- current-output
| "f1"        |
| t           |
+-------------+

(c3) gentranpush(false,true);

(d3)                        [f2,f3,true]

Output stack:

+-------------+
| "f2" "f3" t |  <-- current-output
| "f2" "f3"   |
| "f1"        |
| t           |
+-------------+

(c4) gentranpush("f1");

(d4)                        f1
```

```
Output stack:

+-------------+
| "f1"        |  <-- current-output
| "f2" "f3" t |
| "f2" "f3"   |
| "f1"        |
| t           |
+-------------+
```

4.2.2. gentranpop

User Level Syntax:

*gentranpop(*f1,f2,...,fn);

where f1,f2,...,fn is a list of one or more f's, and each f is one of:

`a string  =  an output file`
true = the terminal
false = the current output file(s)
all = all files currently open for output by GENTRAN

LISP Level Syntax:

*(gentranpop '(*f1 f2 ... fn))

where f1 f2 ... fn is a list of one or more f's, and each f is one of:

`an atom  =  an output file`
f = the terminal
nil = the current output file(s)
$all = all files currently open for output by GENTRAN

Side Effects:

gentranpop deletes the top-most occurrence of the single element containing the file name(s) represented by f1,f2,...,fn from the output stack. Files whose names have been completely removed from the output stack are closed. The current output file is reset to the (new) element on the top of the output stack.

Return Value:

gentranpop returns the current output file(s) after this command has been executed.

Diagnostic Messages

File Not Open for Output

Wrong Type of Arg

User Level Examples:

```
Output stack:

+-------------+
| "f4"        |  <-- current-output
| "f4" "f2" t |
| "f4"        |
| t           |
| "f3"        |
| "f2" "f1"   |
| t           |
+-------------+

(c1) gentranpop(false);

(d1)                        [f4,f2,true]

Output stack:

+-------------+
| "f4" "f2" t |  <-- current-output
| "f4"        |
| t           |
| "f3"        |
| "f2" "f1"   |
| t           |
+-------------+

(c2) gentranpop("f2","f1");

(d2)                        [f4,f2,true]

Output stack:

+-------------+
| "f4" "f2" t |  <-- current-output
| "f4"        |
| t           |
| "f3"        |
| t           |
+-------------+

(c3) gentranpop(false);

(d3)                        f4

Output stack:

+-------------+
| "f4"        |  <-- current-output
| t           |
| "f3"        |
| t           |
+-------------+
```

```
(c4) gentranpop(all);

(d4)                            true

Output stack:

+-------------+
| t           |  <-- current-output
+-------------+
```

4.3. Temporary Output Redirection

Sections 2.2 and 3.1 explain how to use the code generation and template processing commands. The syntax for these two commands is:

*gentran(*stmt1,stmt2,...,stmtn {,[f1,f2,...,fn]});

and

*gentranin(*f1,f2,...,fn {,[f1,f2,...,fm]});

The optional parts of these two commands can be used for *temporary* output redirection; they can be used when the current output file is to be temporarily reset, for this command only.

Thus the following two sequences of commands are equivalent:

(c1) *gentranpush*(f1,f2,...,fm)$

(c2) *gentran*(stmt1,stmt2,...,stmtn)$

(c3) *gentranpop*(false)$

and

(c1) *gentran(*stmt1,stmt2,...,stmtn, [f1,f2,...,fm])$

5. MODIFICATION OF THE CODE GENERATION PROCESS

GENTRAN is designed to be flexible enough to be used in a variety of code generation applications. For this reason, several mode switches and variables are provided to enable the user to tailor the code generation process to meet his or her particular needs.

5.1. Mode Switches

Two special functions, *on* and *off*, are provided to enable the user to turn GENTRAN mode switches on and off:

User Level Syntax:

*on(*flag1,flag2,...,flagn);

*off(*flag1,flag2,...,flagn);

where flag1,flag2,...,flagn is a list of one or more flag's, and each flag is one of the mode switches described in this section.

LISP Level Syntax:

*(on '(*flag1 flag2 ... flagn))

(off '(flag1 flag2 ... flagn))

where flag1 flag2 ... flagn is a list of one or more flag's, and each flag is one of the mode switches described in this section.

Side Effects:

When the *on* function is called, each of the LISP variables corresponding to flag1,flag2,...,flagn is set to *t*. Similarly, when the *off* function is called, each of the corresponding LISP variables is set to *nil*.

When either function is called, the property lists of each of the corresponding LISP variables are also checked. Each property list that contains a *simpfg* property and has a function call as part of the property's value also has that function call evaluated.

Return Value:

Each of these functions returns the atom *done* (*$done* at the LISP level).

The following GENTRAN mode switches can be turned on and off with the *on* and *off* functions:

fortran

- when turned on, causes all VAXIMA statements and expressions typed in at the user's terminal to be translated into FORTRAN code until an *off(fortran)$* command is issued
The special GENTRAN functions *eval*, *lsetq*, *rsetq*, *lrsetq*, *type*, and *literal*, can be entered at the user level of VAXIMA while *fortran* is on.
All generated code is written to the current output file(s).
See also section 2.6.
- default setting: off

ratfor

- when turned on, causes all VAXIMA statements and expressions typed in at the user's terminal to be translated into RATFOR code until an *off(ratfor)$* command is issued
The special GENTRAN functions *eval*, *lsetq*, *rsetq*, *lrsetq*, *type*, and *literal*, can be entered at the user level of VAXIMA while *ratfor* is on.
All generated code is written to the current output file(s).
See also section 2.6.
- default setting: off

c

- when turned on, causes all VAXIMA statements and expressions typed in at the user's terminal to be translated into C code until an *off(c)$* command is issued
The special GENTRAN functions *eval*, *lsetq*, *rsetq*, *lrsetq*, *type*, and *literal*, can be entered at the user level of VAXIMA while *c* is on.
All generated code is written to the current output file(s).
See also section 2.6.
- default setting: off

gendecs

- When turned on, enables type declarations to be generated automatically.
As long as *gendecs* is on, declaration information is extracted from the Symbol Table and converted into declarations as soon as possible.
However, when *gendecs* is turned off, declarations are not generated; information is only stored in the Symbol Table.
If *gendecs* is turned back on after being off, all type information is extracted from the Symbol Table, and declarations are generated immediately (i.e., *gendecs(false);* is evaluated).
See also sections 2.8.2 and 3.2.
- default setting: on

float

- when turned on, causes integers in generated numerical code to be converted to floating point numbers except in the following places: array subscripts, powers to which expressions are raised (in exponentiation), and initial, final, and step values in do loops.
- default setting: on

gentranseg

- when turned on, checks the print length of expressions and breaks those expressions that are longer than *?maxexpprintlen** (*maxexpprintlen**) down into subexpressions and assigns them to temporary variables
See also section 2.8.
- default setting: on

gentranopt

- when turned on, replaces each block of straightline code by an optimized sequence of assignments
The *optimize* facility takes an assignment and replaces common subexpressions by temporary variables. It returns the resulting assignment statements with common-subexpression-to-temporary-variable assignment statements preceding them
- default setting: off

gentranparser

- when turned off, improves translation speed by disabling the GENTRAN parser module.
Note: when this flag is off, error messages for statements which cannot be translated will not be printed.
- default setting: off

5.2. Variables

Several global variables are provided in GENTRAN to enable the user to

- select the target language
- control expression optimization and segmentation
- change automatically generated variable names and statement numbers
- modify the code formatter

The following four subsections describe these variables. Each variable is given in its user level syntax, followed by its LISP level syntax.

5.2.1. Target Language Selection

?gentranlang* **gentranlang***

- target language (FORTRAN, RATFOR, or C)
See also section 2.1.
- value type: atom
- default value: fortran

?gentranopt* **gentranopt*** ** kentvax only **

- system under which code is to be optimized (VAXIMA or REDUCE)
See also section 2.7.
- value type: atom
- default value: vaxima

5.2.2. Expression Optimization & Segmentation Control

?maxexpprintlen* **maxexpprintlen***

- value used to determine whether or not an expression should be segmented; maximum number of characters permitted in an expression in the target language (excluding spaces printed for formatting)
See also section 2.8.
- value type: integer
- default value: 800

5.2.3. Temporary Variable Names & Statement Numbers

?tempvarname* **tempvarname***

- name used as prefix in generating temporary variable names
See also section 2.9.
- value type: atom
- default value: t

?tempvarnum* **tempvarnum***

- number appended to *?tempvarname** to create a temporary variable name
If the temporary variable name resulting from appending *?tempvarnum** onto *?tempvarname** has already been generated and still holds a useful value, then *?tempvarnum** is incremented and temporary variable names are compressed until one is found which was not previously generated or does not still hold a significant value.
See also section 2.9.
- value type: integer
- default value: 0

?tempvartype* **tempvartype***

- target language variable type (e.g., integer, real*8, float, etc) used as a default for automatically generated variables whose type cannot be determined otherwise
If *?tempvartype** is false (nil), then generated temporary variables whose type cannot be determined are not automatically declared.
See also section 2.8.
- value type: atom
- default value: false (nil)

?genstmtno*   genstmtno*

- number used when a statement number must be generated
- value type: integer
- default value: 25000

?genstmtincr*   genstmtincr*

- number by which *?genstmtno** is increased each time a new statement number is generated
- value type: integer
- default value: 1

5.2.4. Code Formatting

?fortcurrind*   fortcurrind*

- number of blank spaces printed at the beginning of each line of generated FORTRAN code
- value type: integer
- default value: 6

?ratcurrind*   ratcurrind*

- number of blank spaces printed at the beginning of each line of generated RATFOR code
- value type: integer
- default value: 0

?ccurrind*   ccurrind*

- number of blank spaces printed at the beginning of each line of generated C code
- value type: integer
- default value: 0

?tablen*   tablen*

- number of blank spaces printed for each new level of indentation
Automatic indentation can be turned off by setting this variable to 0.
- value type: integer
- default value: 4

?fortlinelen*   fortlinelen*

- maximum number of characters printed on each line of generated FORTRAN code
- value type: integer
- default value: 72

?ratlinelen*   ratlinelen*

- maximum number of characters printed on each line of generated RATFOR code
- value type: integer
- default value: 80

?clinelen*   clinelen*

- maximum number of characters printed on each line of generated C code
- value type: integer
- default value: 80

6. EXAMPLES

Short examples have been given throughout this manual to illustrate basic usage of the GENTRAN commands. This section contains some complete examples.

6.1. Example 1: Interactive Code Generation

Suppose we wish to generate a FORTRAN subprogram which can be used for computing the roots of a polynomial by Graeffe's Root-Squaring Method[4]. This method states that the roots

$$x_i$$

of a polynomial

$$P_n(x) = \sum a_i x^{n-i}$$

can be found by constructing the polynomial

$$P^*_n(x^2) = (a_0 x^n + a_2 x^{n-2} + \ldots)^2 - (a_1 x^{n-1} + a_3 x^{n-3} + \ldots)^2$$

with roots

$$x_i^2$$

When *batch*'d into VAXIMA, the following file of VAXIMA statements will place the coefficients of

$$P^*_n$$

into array b for some user-entered value of n greater than zero:

```
Contents of file graeffe.in:
+-----------------------------+
|q : 0$                       |
|for i:0 step 2 thru n do     |
|    q : q + a(i+1)*x^(n-i)$  |   [5]
|r : 0$                       |
|for i:1 step 2 thru n-1 do   |
|    r : r + a(i+1)*x^(n-i)$  |
|p : q^2 - r^2$               |
|p : ratsubst(y, x^2, p)$     |
|for i:0 thru n do            |
|    b[i] : ratcoeff(p, y, i)$|
+-----------------------------+
```

Now a numerical subprogram can be generated with assignment statements for the coefficients of

$$P^*_n$$

[4] This is for instance convenient for ill-conditioned polynomials. More details are given in *Introduction to Numerical Analysis* by C. E. Froberg, Addison-Wesley Publishing Company, 1966.

[5] In accordance with section 2.4, the subscripts of a are i+1 instead of i.

(now stored in array b in VAXIMA). Since these coefficients are given in terms of the coefficients of

```
P
 n
```

(i.e., subscripted variable a in VAXIMA), the subprogram will need two parameters: a and b, each of which must be arrays of size n+1.

The following VAXIMA session will create subroutine graeff for a polynomial of degree n=10 and write it to file graeffe.f:

```
(c1) n : 10$

(c2) batch("graeffe.in");

(c3) q : 0$

(c4) for i:0 step 2 thru n do
    q : q + a(i+1)*x^(n-i)$

(c5) r : 0$

(c6) for i:1 step 2 thru n-1 do
    r : r + a(i+1)*x^(n-i)$

(c7) p : q^2 - r^2$

(c8) p : ratsubst(y, x^2, p)$

(c9) for i:0 thru n do
    b[i] : ratcoeff(p, y, i)$

(d10)                      BATCH DONE

(c11) gentranlang(fortran)$

(c12) gentranpush("graeffe.f");

(d12)                      graeffe.f

(c13) gentran( subroutine( graeff(a,b) ),
               type("real*8", a(eval(n+1)), b(eval(n+1))),
               literal("c",cr),
               literal("c",tab,"--  Graeffe Root-Squaring Method   --",cr),
               literal("c",tab,"--  to Find Roots of a Polynomial  --",cr),
               literal("c",cr) );

(d13)                      graeffe.f

(c14) for i:1 thru n+1 do
          gentran( lrsetq( b[i], b[i-1] ) );

(d14)                      done
```

```
(c15) gentran( return(), end() );

(d15)                     graeffe.f

(c16) gentranpop(false);

(d16)                     true
```

Contents of file graeffe.f:

```
+------------------------------------------------------------------------+
|      subroutine graeff(a,b)                                            |
|      real*8 a(11),b(11)                                                |
|c                                                                       |
|c     --  Graeffe Root-Squaring Method   --                             |
|c     --  to Find Roots of a Polynomial  --                             |
|c                                                                       |
|      b(1)=a(11)**2                                                     |
|      b(2)=-a(10)**2+2.0*a(9)*a(11)                                     |
|      b(3)=a(9)**2-2.0*a(8)*a(10)+2.0*a(7)*a(11)                        |
|      b(4)=-a(8)**2+2.0*a(7)*a(9)-2.0*a(6)*a(10)+2.0*a(5)*a(11)         |
|      b(5)=a(7)**2-2.0*a(6)*a(8)+2.0*a(5)*a(9)-2.0*a(4)*a(10)+2.0*a(3)*a|
|     . (11)                                                             |
|      b(6)=-a(6)**2+2.0*a(5)*a(7)-2.0*a(4)*a(8)+2.0*a(3)*a(9)-2.0*a(2)*a|
|     . (10)+2.0*a(1)*a(11)                                              |
|      b(7)=a(5)**2-2.0*a(4)*a(6)+2.0*a(3)*a(7)-2.0*a(2)*a(8)+2.0*a(1)*a(|
|     . 9)                                                               |
|      b(8)=-a(4)**2+2.0*a(3)*a(5)-2.0*a(2)*a(6)+2.0*a(1)*a(7)           |
|      b(9)=a(3)**2-2.0*a(2)*a(4)+2.0*a(1)*a(5)                          |
|      b(10)=-a(2)**2+2.0*a(1)*a(3)                                      |
|      b(11)=a(1)**2                                                     |
|      return                                                            |
|      end                                                               |
+------------------------------------------------------------------------+
```

6.2. Example 2: Template Processing

Circuit simulation plays a vital role in computer hardware development. A recent paper[6] describes the design of an Automatic Circuitry Code Generator (ACCG), which generates circuit simulation programs based on user-supplied circuit specifications. The actual code generator runs under the REDUCE computer algebra system and consists of a series of *write* statements, each of which writes one line of FORTRAN code.

This section presents an alternative implementation for the ACCG in VAXIMA which uses GENTRAN's template processor to generate code. Template processing is a much more natural method of code generation than the *write* statement method.

First we will put all VAXIMA calculations into two files: rk.in and ham.in.

```
Contents of file rk.in:
+-----------------------------------------------------------------+
|/*  Runge-Kutta Method  */                                        |
|                                                                 |
|rungekutta(p1, p2, p, q, tt) :=                                  |
|  block( [k11, k12, k21, k22, k31, k32, k41, k42],               |
|         k11 : hh*p1,                                            |
|         k12 : hh*p2,                                            |
|         k21 : hh*ratsubst(q+k12/2, q,                           |
|                           ratsubst(p+k11/2, p,                  |
|                                    ratsubst(tt+hh/2, tt, p1))),|
|         k22 : hh*ratsubst(q+k12/2, q,                           |
|                           ratsubst(p+k11/2, p,                  |
|                                    ratsubst(tt+hh/2, tt, p2))),|
|         k31 : hh*ratsubst(q+k22/2, q,                           |
|                           ratsubst(p+k21/2, p,                  |
|                                    ratsubst(tt+hh/2, tt, p1))),|
|         k32 : hh*ratsubst(q+k22/2, q,                           |
|                           ratsubst(p+k21/2, p,                  |
|                                    ratsubst(tt+hh/2, tt, p2))),|
|         k41 : hh*ratsubst(q+k32, q,                             |
|                           ratsubst(p+k31, p,                    |
|                                    ratsubst(tt+hh, tt, p1))),  |
|         k42 : hh*ratsubst(q+k32, q,                             |  [7]
|                           ratsubst(p+k31, p,                    |
|                                    ratsubst(tt+hh, tt, p2))),  |
|         pn : ratsimp(p + (k11 + 2*k21 + 2*k31 + k41)/6),        |
|         qn : ratsimp(q + (k12 + 2*k22 + 2*k32 + k42)/6)         |
|       )$                                                        |
+-----------------------------------------------------------------+
```

```
Contents of file ham.in:
+---------------------------------------------------------+
|/*  Hamiltonian Calculation  */                          |
|                                                         |
|difq : diff(h, p)$                                       |
|difp : -diff(h, q) - ratsubst(p/m, qdot, diff(d, qdot))$|
|rungekutta(difp, difq, p, q, tt)$                        |
+---------------------------------------------------------+
```

---

[6] Loe, K. F., N. Ohsawa, and E. Goto. "Design of an Automatic Circuitry Code Generator (ACCG)", *RSYMSAC Proceedings*, Riken, Wako-shi, Saitama, Japan. August 21-22, 1984.

[7] Line 20 of procedure rungekutta was changed from an assignment to k41, as given in [6], to an assignment to k42.

Next we will create a template file with an outline of the target FORTRAN program and GENTRAN commands.

```
Contents of file runge.tem:
+----------------------------------------------------------------------+
|      program runge                                                   |
|      implicit real (k,m)                                             |
|c                                                                     |
|c  Input                                                              |
|c                                                                     |
|      write(6,*) 'Initial Value of p'                                  |
|      read(5,*) p                                                      |
|      write(6,*) ' p = ', p                                            |
|      write(6,*) 'Initial Value of q'                                  |
|      read(5,*) q                                                      |
|      write(6,*) ' q = ', q                                            |
|      write(6,*) 'Value of m'                                          |
|      read(5,*) m                                                      |
|      write(6,*) ' m = ', m                                            |
|      write(6,*) 'Value of k0'                                         |
|      read(5,*) k0                                                     |
|      write(6,*) ' k0 = ', k0                                          |
|      write(6,*) 'Value of b'                                          |
|      read(5,*) b                                                      |
|      write(6,*) ' b = ', b                                            |
|      write(6,*) 'Step Size of t'                                      |
|      read(5,*) hh                                                     |
|      write(6,*) ' Step Size of t = ', hh                              |
|      write(6,*) 'Final Value of t'                                    |
|      read(5,*) tp                                                     |
|      write(6,*) ' Final Value of t = ', tp                            |
```

```
|c                                                                      |
|c  Initialization                                                      |
|c                                                                      |
|      tt=0.0                                                           |
|<<                                                                     |
|      gentran( literal(tab, "write(9,*) ' h = ", eval(h), "'", cr),    |
|               literal(tab, "write(9,*) ' d = ", eval(d), "'", cr) )$ |
|>>                                                                     |
|      write(9,901) c                                                   |
|901   format(' c= ',e20.10)                                            |
|      write(9,910) tt, q, p                                            |
|910   format(' ',3e20.10)                                              |
|c                                                                      |
|c  Loop                                                                |
|c                                                                      |
|<<                                                                     |
|      gentran( unless tt >= tf do                                      |
|               (                                                       |
|                    rsetq(pn, ev(pn,expand)),                          |
|                    rsetq(q, ev(qn,expand)),                           |
|                    p : pn,                                            |
|                    tt : tt + hh,                                      |
|                    literal(tab, "write(9,910) tt, q, p", cr)          |
|               ) )$                                                    |
|>>                                                                     |
|      stop                                                             |
|      end                                                              |
+-----------------------------------------------------------------------+
```

Now we can generate a circuit simulation program simply by starting a VAXIMA session and following three steps:

```
    (1)  enter circuit specifications
    (2)  perform calculations
    (3)  call the GENTRAN template processor
```

For example, the following VAXIMA session will write a simulation program to the file runge.f:

```
/*  Enter Circuit Specifications  */

(c1) k : 1/(2*m)*p^2$

(c2) u : k0/2*q^2$

(c3) d : b/2*qdot$

(c4) h : k + u$
```

```
/*  Perform Calculations  */

(c5) batch("rk.in");

(c6) /*  Runge-Kutta Method  */

rungekutta(p1, p2, p, q, tt) :=
  block( [k11, k12, k21, k22, k31, k32, k41, k42],
         k11 : hh*p1,
         k12 : hh*p2,
         k21 : hh*ratsubst(q+k12/2, q,
                           ratsubst(p+k11/2, p,
                                    ratsubst(tt+hh/2, tt, p1))),
         k22 : hh*ratsubst(q+k12/2, q,
                           ratsubst(p+k11/2, p,
                                    ratsubst(tt+hh/2, tt, p2))),
         k31 : hh*ratsubst(q+k22/2, q,
                           ratsubst(p+k21/2, p,
                                    ratsubst(tt+hh/2, tt, p1))),
         k32 : hh*ratsubst(q+k22/2, q,
                           ratsubst(p+k21/2, p,
                                    ratsubst(tt+hh/2, tt, p2))),
         k41 : hh*ratsubst(q+k32, q,
                           ratsubst(p+k31, p,
                                    ratsubst(tt+hh, tt, p1))),
         k42 : hh*ratsubst(q+k32, q,
                           ratsubst(p+k31, p,
                                    ratsubst(tt+hh, tt, p2))),
         pn : ratsimp(p + (k11 + 2*k21 + 2*k31 + k41)/6),
         qn : ratsimp(q + (k12 + 2*k22 + 2*k32 + k42)/6)
       )$

(d7)                         BATCH DONE

(c8) batch("ham.in");

(c9) /*  Hamiltonian Calculation  */

difq : diff(h, p)$

(c10) difp : -diff(h, q) - ratsubst(p/m, qdot, diff(d, qdot))$

(c11) rungekutta(difp, difq, p, q, tt)$

(d12)                        BATCH DONE
```

/* Call Template Processor */

```
(c13) gentranlang(fortran)$

(c14) on(gentranfloat)$
```

```
(c15) gentranin("runge.tem", ["runge.f"]);

(d15)                      runge.f

Contents of file runge.f:
+--------------------------------------------------------------------------+
|      program runge                                                       |
|      implicit real (k,m)                                                 |
|c                                                                         |
|c  Input                                                                  |
|c                                                                         |
|      write(6,*) 'Initial Value of p'                                     |
|      read(5,*) p                                                         |
|      write(6,*) ' p = ', p                                               |
|      write(6,*) 'Initial Value of q'                                     |
|      read(5,*) q                                                         |
|      write(6,*) ' q = ', q                                               |
|      write(6,*) 'Value of m'                                             |
|      read(5,*) m                                                         |
|      write(6,*) ' m = ', m                                               |
|      write(6,*) 'Value of k0'                                            |
|      read(5,*) k0                                                        |
|      write(6,*) ' k0 = ', k0                                             |
|      write(6,*) 'Value of b'                                             |
|      read(5,*) b                                                         |
|      write(6,*) ' b = ', b                                               |
|      write(6,*) 'Step Size of t'                                         |
|      read(5,*) hh                                                        |
|      write(6,*) ' Step Size of t = ', hh                                 |
|      write(6,*) 'Final Value of t'                                       |
|      read(5,*) tp                                                        |
|      write(6,*) ' Final Value of t = ', tp                               |
|c                                                                         |
|c  Initialization                                                         |
|c                                                                         |
|      tt=0.0                                                              |
|      write(9,*) ' h = p**2/2.0/m+k0*q**2/2.0'                            |
|      write(9,*) ' d = b*qdot/2.0'                                        |
|      write(9,901) c                                                      |
|901   format(' c= ',e20.10)                                               |
|      write(9,910) tt, q, p                                               |
|910   format(' ',3e20.10)                                                 |
```

```
|c                                                                         |
|c  Loop                                                                   |
|c                                                                         |
|25001 if (tt.ge.tf) goto 25002                                            |
|          pn=-1.0/2.0*b*hh-1.0/96.0*b*hh**5*k0**2/m**2+b*hh**3*k0/12.0/m|
|     .     +p-1.0/96.0*hh**6*k0**3*p/m**3+hh**4*k0**2*p/8.0/m**2-1.0/2.0|
|     .     *hh**2*k0*p/m-hh*k0*q-1.0/48.0*hh**5*k0**3*q/m**2+hh**3*k0**2|
|     .     *q/6.0/m                                                       |
|          q=b*hh**8*k0**3/384.0/m**4-1.0/48.0*b*hh**6*k0**2/m**3+b*hh**4|
|     .     *k0/12.0/m**2-1.0/4.0*b*hh**2/m+hh**9*k0**4*p/384.0/m**5-1.0/|
|     .     32.0*hh**7*k0**3*p/m**4+hh**5*k0**2*p/6.0/m**3-1.0/2.0*hh**3*|
|     .     k0*p/m**2+hh*p/m+q+hh**8*k0**4*q/192.0/m**4-1.0/24.0*hh**6*k0|
|     .     **3*q/m**3+hh**4*k0**2*q/6.0/m**2-1.0/2.0*hh**2*k0*q/m       |
|          p=pn                                                           |
|          tt=tt+hh                                                       |
|          write(9,910) tt, q, p                                          |
|          goto 25001                                                     |
|25002 continue                                                            |
|      stop                                                               |
|      end                                                                |
+--------------------------------------------------------------------------+
```

6.3. Example 3: Segmentation & Optimization Techniques

The following 3 x 3 inertia matrix, mat, was derived in the course of research reported in [8]:

```
mat[1,1] :
             2    2               2               2
    - 9 m30 p  sin (q3) + j30z sin (q3) - j30y sin (q3)
                2                            2        2
      + 18 m30 p  cos(q2) cos(q3) + 18 m30 p  + m10 p  + j30y

      + j10y

mat[1,2] :
             2    2               2               2
    - 9 m30 p  sin (q3) + j30z sin (q3) - j30y sin (q3)
               2                           2
      + 9 m30 p  cos(q2) cos(q3) + 9 m30 p  + j30y

mat[1,3] :
             2
    - 9 m30 p  sin(q2) sin(q3)

mat[2,1] :
             2    2               2               2
    - 9 m30 p  sin (q3) + j30z sin (q3) - j30y sin (q3)
               2                           2
      + 9 m30 p  cos(q2) cos(q3) + 9 m30 p  + j30y

mat[2,2] :
             2    2               2               2               2
    - 9 m30 p  sin (q3) + j30z sin (q3) - j30y sin (q3) + 9 m30 p

      + j30y

mat[2,3] :

    0

mat[3,1] :
            2
    - 9 m30 p  sin(q2) sin(q3)

mat[3,2] :

    0

mat[3,3] :
           2
    9 m30 p  + j30x
```

We know mat is symmetric. We wish to generate numerical code to compute values for mat and its inverse matrix, inv.

[8] Bos, A. M. and M. J. L. Tiernego. "Formula Manipulation in the Bond Graph Modelling and Simulation of Large Mechanical Systems", *Journal of the Franklin Institute*, Pergamon Press Ltd., Vol. 319, No. 1/2, pp. 51-65, January/February 1985.

6.3.1. Code Generation

Generating code for matrix mat and its inverse matrix is straightforward. We can simply generate an assignment statement for each element of mat, compute the inverse matrix, inv, and generate an assignment statement for each element of inv. Since we know mat is symmetric across the main diagonal, we know that inv will also be symmetric. To avoid duplicate computations, we will not generate assignments for elements below the diagonals of these matrices. Instead, we will generate nested loops to copy elements across the main diagonals.

The following VAXIMA session will write to the file matrix.f:

```
(c1) batch("matrix.v");

(c2) mat : genmatrix(mat, 3,3, 1,1)$

(c3) mat[1,1] :
     18*cos(q3)*cos(q2)*m30*p^2 - 9*sin(q3)^2*p^2*m30
     - sin(q3)^2*j30y + sin(q3)^2*j30z + p^2*m10 + 18*p^2*m30
     + j10y + j30y$

(c4) mat[2,1] :
     mat[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$

(c5) mat[3,1] :
     mat[1,3] :
     -9*sin(q3)*sin(q2)*m30*p^2$

(c6) mat[2,2] :
     -sin(q3)^2*j30y + sin(q3)^2*j30z - 9*sin(q3)^2 *m30*p^2
     + j30y + 9*m30*p^2$

(c7) mat[3,2] :
     mat[2,3] :
     0$

(c8) mat[3,3] :
     9*m30*p^2 + j30x$

(d9)                        BATCH DONE

(c10) gentranout("matrix.f")$

(c11) gentran( literal("c", cr,
                       "c --- Calculate Matrix Values ---", cr,
                       "c", cr) )$

(c12) for i:1 thru 3 do
          for j:i thru 3 do
              gentran( lrsetq(mat[i,j], mat[i,j]) )$

(c13) inv : mat^^(-1)$
```

```
(c14) gentran( literal("c", cr,
                       "c --- Calculate Inverse Matrix Values ---",
                       cr, "c", cr) )$

(c15) for i:1 thru 3 do
          for j:i thru 3 do
              gentran( lrsetq( inv[i,j], inv[i,j]) )$

(c16) gentran( literal("c", cr,
                       "c --- Copy Entries Across Main Diagonals ---",
                       cr, "c", cr),
               for i:1 thru 3 do
                   for j:i+1 thru 3 do
                   (   mat[j,i] : mat[i,j],  inv[j,i] : inv[i,j]   )
             )$

(c17) gentranshut("matrix.f")$
```

Contents of file matrix.f:

```
c
c --- Calculate Matrix Values ---
c
      mat(1,1)=j10y+j30y+m10*p**2+18.0*m30*p**2+18.0*m30*p**2*cos(q2)*
     . cos(q3)-j30y*sin(q3)**2+j30z*sin(q3)**2-9.0*m30*p**2*sin(q3)**2
      mat(1,2)=j30y+9.0*m30*p**2+9.0*m30*p**2*cos(q2)*cos(q3)-j30y*sin(
     . q3)**2+j30z*sin(q3)**2-9.0*m30*p**2*sin(q3)**2
      mat(1,3)=-9.0*m30*p**2*sin(q2)*sin(q3)
      mat(2,2)=j30y+9.0*m30*p**2-j30y*sin(q3)**2+j30z*sin(q3)**2-9.0*m30
     . *p**2*sin(q3)**2
      mat(2,3)=0.0
      mat(3,3)=j30x+9.0*m30*p**2
c
c --- Calculate Inverse Matrix Values ---
c
      inv(1,1)=-(-j30x*j30y+(-9.0*j30x-9.0*j30y)*m30*p**2-81.0*m30**2*p
     . **4+(j30x*j30y-j30x*j30z+(9.0*j30x+9.0*j30y-9.0*j30z)*m30*p**2+
     . 81.0*m30**2*p**4)*sin(q3)**2)/(j10y*j30x*j30y+(j30x*j30y*m10+(9.0
     . *j10y*j30x+(9.0*j10y+9.0*j30x)*j30y)*m30)*p**2+((9.0*j30x+9.0*
     . j30y)*m10*m30+(81.0*j10y+81.0*j30x+81.0*j30y)*m30**2)*p**4+(81.0*
     . m10*m30**2+729.0*m30**3)*p**6+(-81.0*j30x*m30**2*p**4-729.0*m30**
     . 3*p**6)*cos(q2)**2*cos(q3)**2+(-j10y*j30x*j30y+j10y*j30x*j30z+((-
     . j30x*j30y+j30x*j30z)*m10+(-9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*
     . j30y+(9.0*j10y+9.0*j30x)*j30z)*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0
     . *j30z)*m10*m30+(-81.0*j10y-81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)
     . *p**4+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*j30y*m30**2*p**
     . 4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0*j30y-81.0*j30z
     . )*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3)**4)
      inv(1,2)=(-j30x*j30y+(-9.0*j30x-9.0*j30y)*m30*p**2-81.0*m30**2*p**
     . 4+(-9.0*j30x*m30*p**2-81.0*m30**2*p**4)*cos(q2)*cos(q3)+(j30x*
     . j30y-j30x*j30z+(9.0*j30x+9.0*j30y-9.0*j30z)*m30*p**2+81.0*m30**2*
     . p**4)*sin(q3)**2)/(j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(
     . 9.0*j10y+9.0*j30x)*j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(
     . 81.0*j10y+81.0*j30x+81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+
     . 729.0*m30**3)*p**6+(-81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos
     . (q2)**2*cos(q3)**2+(-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+
     . j30x*j30z)*m10+(-9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*
     . j10y+9.0*j30x)*j30z)*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10
     . *m30+(-81.0*j10y-81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4+(
     . -81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*j30y*m30**2*p**4-729.0
     . *m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0*j30y-81.0*j30z)*m30**
     . 2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3)**4)
      inv(1,3)=-((-9.0*j30y*m30*p**2-81.0*m30**2*p**4)*sin(q2)*sin(q3)+(
     . (9.0*j30y-9.0*j30z)*m30*p**2+81.0*m30**2*p**4)*sin(q2)*sin(q3)**3
     . )/(j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*
     . j30x)*j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+
     . 81.0*j30x+81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*
     . p**6+(-81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos(q2)**2*cos(q3
     . )**2+(-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+
     . (-9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*
     . j30z)*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*
     . j10y-81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4+(-81.0*m10*m30**
```

```
     . 2-729.0*m30**3)*p**6+(-81.0*j30y*m30**2*p**4-729.0*m30**3*p**6)*
     . sin(q2)**2)*sin(q3)**2+((81.0*j30y-81.0*j30z)*m30**2*p**4+729.0*
     . m30**3*p**6)*sin(q2)**2*sin(q3)**4)
      inv(2,2)=-(-j10y*j30x-j30x*j30y+(-j30x*m10+(-9.0*j10y-18.0*j30x
     . -9.0*j30y)*m30)*p**2+(-9.0*m10*m30-162.0*m30**2)*p**4+(-18.0*j30x
     . *m30*p**2-162.0*m30**2*p**4)*cos(q2)*cos(q3)+(j30x*j30y-j30x*j30z
     . +(9.0*j30x+9.0*j30y-9.0*j30z)*m30*p**2+81.0*m30**2*p**4+81.0*m30
     . **2*p**4*sin(q2)**2)*sin(q3)**2)/(j10y*j30x*j30y+(j30x*j30y*m10+(
     . 9.0*j10y*j30x+(9.0*j10y+9.0*j30x)*j30y)*m30)*p**2+((9.0*j30x+9.0*
     . j30y)*m10*m30+(81.0*j10y+81.0*j30x+81.0*j30y)*m30**2)*p**4+(81.0*
     . m10*m30**2+729.0*m30**3)*p**6+(-81.0*j30x*m30**2*p**4-729.0*m30**
     . 3*p**6)*cos(q2)**2*cos(q3)**2+(-j10y*j30x*j30y+j10y*j30x*j30z+((-
     . j30x*j30y+j30x*j30z)*m10+(-9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*
     . j30y+(9.0*j10y+9.0*j30x)*j30z)*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0
     . *j30z)*m10*m30+(-81.0*j10y-81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)
     . *p**4+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*j30y*m30**2*p**
     . 4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0*j30y-81.0*j30z
     . )*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3)**4)
      inv(2,3)=(((-9.0*j30y*m30*p**2-81.0*m30**2*p**4)*sin(q2)-81.0*m30
     . **2*p**4*cos(q2)*sin(q2)*cos(q3))*sin(q3)+((9.0*j30y-9.0*j30z)*
     . m30*p**2+81.0*m30**2*p**4)*sin(q2)*sin(q3)**3)/(j10y*j30x*j30y+(
     . j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*j30x)*j30y)*m30)*p**2+
     . ((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+81.0*j30x+81.0*j30y)*m30
     . **2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*p**6+(-81.0*j30x*m30**2*
     . p**4-729.0*m30**3*p**6)*cos(q2)**2*cos(q3)**2+(-j10y*j30x*j30y+
     . j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+(-9.0*j10y*j30x+(-9.0*
     . j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*j30z)*m30)*p**2+((-9.0*
     . j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y-81.0*j30x-81.0*j30y+
     . 81.0*j30z)*m30**2)*p**4+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(
     . -81.0*j30y*m30**2*p**4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+
     . ((81.0*j30y-81.0*j30z)*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*
     . sin(q3)**4)
      inv(3,3)=-(-j10y*j30y+(-j30y*m10+(-9.0*j10y-9.0*j30y)*m30)*p**2+(
     . -9.0*m10*m30-81.0*m30**2)*p**4+81.0*m30**2*p**4*cos(q2)**2*cos(q3
     . )**2+(j10y*j30y-j10y*j30z+((j30y-j30z)*m10+(9.0*j10y+9.0*j30y-9.0
     . *j30z)*m30)*p**2+(9.0*m10*m30+81.0*m30**2)*p**4)*sin(q3)**2)/(
     . j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*j30x)*
     . j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+81.0*j30x
     . +81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*p**6+(
     . -81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos(q2)**2*cos(q3)**2+(
     . -j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+(-9.0*
     . j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*j30z)*m30
     . )*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y-81.0*
     . j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4+(-81.0*m10*m30**2-729.0*
     . m30**3)*p**6+(-81.0*j30y*m30**2*p**4-729.0*m30**3*p**6)*sin(q2)**
     . 2)*sin(q3)**2+((81.0*j30y-81.0*j30z)*m30**2*p**4+729.0*m30**3*p**
     . 6)*sin(q2)**2*sin(q3)**4)
c
c --- Copy Entries Across Main Diagonals ---
c
      do 25001 i=1,3
          do 25002 j=i+1,3
              mat(j,i)=mat(i,j)
              inv(j,i)=inv(i,j)
```

```
|25002     continue                                                         |
|25001 continue                                                             |
+---------------------------------------------------------------------------+
```

6.3.2. Segmentation

The assignment statements generated in the previous subsection were several lines long. Although none of the statements generated in that example exceeded the 19 continuation line limit imposed by most FORTRAN compilers, symbolic computations can easily yield expressions which are much longer. For this reason, the segmentation facility automatically breaks long expressions into subexpressions of manageable size.

For example, we could have generated segmented code by setting the special variable *?maxexpprintlen** to a smaller value. By preceding the VAXIMA session given in the previous subsection by

```
(c1) ?maxexpprintlen\* : 300$
```

we would generate the following:

```
Contents of file matrix.f:
+---------------------------------------------------------------------------+
|c                                                                          |
|c --- Calculate Matrix Values ---                                          |
|c                                                                          |
|      mat(1,1)=j10y+j30y+m10*p**2+18.0*m30*p**2+18.0*m30*p**2*cos(q2)*     |
|     . cos(q3)-j30y*sin(q3)**2+j30z*sin(q3)**2-9.0*m30*p**2*sin(q3)**2      |
|      mat(1,2)=j30y+9.0*m30*p**2+9.0*m30*p**2*cos(q2)*cos(q3)-j30y*sin(     |
|     . q3)**2+j30z*sin(q3)**2-9.0*m30*p**2*sin(q3)**2                       |
|      mat(1,3)=-9.0*m30*p**2*sin(q2)*sin(q3)                                |
|      mat(2,2)=j30y+9.0*m30*p**2-j30y*sin(q3)**2+j30z*sin(q3)**2-9.0*m30|
|     . *p**2*sin(q3)**2                                                     |
|      mat(2,3)=0.0                                                          |
|      mat(3,3)=j30x+9.0*m30*p**2                                            |
|c                                                                          |
|c --- Calculate Inverse Matrix Values ---                                  |
|c                                                                          |
```

```
      t2=-(-j30x*j30y+(-9.0*j30x-9.0*j30y)*m30*p**2-81.0*m30**2*p**4+(
     . j30x*j30y-j30x*j30z+(9.0*j30x+9.0*j30y-9.0*j30z)*m30*p**2+81.0*
     . m30**2*p**4)*sin(q3)**2)
      t0=j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*j30x
     . )*j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+81.0*
     . j30x+81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*p**6
      t1=-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+(
     . -9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*j30z
     . )*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y
     . -81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4
      inv(1,1)=t2/(t0+(-81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos(q2)
     . **2*cos(q3)**2+(t1+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*
     . j30y*m30**2*p**4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0
     . *j30y-81.0*j30z)*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3
     . )**4)
      t2=-j30x*j30y+(-9.0*j30x-9.0*j30y)*m30*p**2-81.0*m30**2*p**4+(-9.0
     . *j30x*m30*p**2-81.0*m30**2*p**4)*cos(q2)*cos(q3)+(j30x*j30y-j30x*
     . j30z+(9.0*j30x+9.0*j30y-9.0*j30z)*m30*p**2+81.0*m30**2*p**4)*sin(
     . q3)**2
      t0=j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*j30x
     . )*j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+81.0*
     . j30x+81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*p**6
      t1=-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+(
     . -9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*j30z
     . )*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y
     . -81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4
      inv(1,2)=t2/(t0+(-81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos(q2)
     . **2*cos(q3)**2+(t1+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*
     . j30y*m30**2*p**4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0
     . *j30y-81.0*j30z)*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3
     . )**4)
      t2=-((-9.0*j30y*m30*p**2-81.0*m30**2*p**4)*sin(q2)*sin(q3)+((9.0*
     . j30y-9.0*j30z)*m30*p**2+81.0*m30**2*p**4)*sin(q2)*sin(q3)**3)
      t0=j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*j30x
     . )*j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+81.0*
     . j30x+81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*p**6
      t1=-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+(
     . -9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*j30z
     . )*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y
     . -81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4
      inv(1,3)=t2/(t0+(-81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos(q2)
     . **2*cos(q3)**2+(t1+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*
     . j30y*m30**2*p**4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0
     . *j30y-81.0*j30z)*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3
     . )**4)
      t0=-j10y*j30x-j30x*j30y+(-j30x*m10+(-9.0*j10y-18.0*j30x-9.0*j30y)*
     . m30)*p**2+(-9.0*m10*m30-162.0*m30**2)*p**4+(-18.0*j30x*m30*p**2
     . -162.0*m30**2*p**4)*cos(q2)*cos(q3)
      t0=-(t0+(j30x*j30y-j30x*j30z+(9.0*j30x+9.0*j30y-9.0*j30z)*m30*p**2
     . +81.0*m30**2*p**4+81.0*m30**2*p**4*sin(q2)**2)*sin(q3)**2)
      t1=j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*j30x
     . )*j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+81.0*
     . j30x+81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*p**6
      t2=-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+(
```

```
|     . -9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*j30z|
|     . )*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y   |
|     . -81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4                     |
|       inv(2,2)=t0/(t1+(-81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos(q2)|
|     . **2*cos(q3)**2+(t2+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*  |
|     . j30y*m30**2*p**4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0|
|     . *j30y-81.0*j30z)*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3|
|     . )**4)                                                            |
|       t2=((-9.0*j30y*m30*p**2-81.0*m30**2*p**4)*sin(q2)-81.0*m30**2*p**4|
|     . *cos(q2)*sin(q2)*cos(q3))*sin(q3)+((9.0*j30y-9.0*j30z)*m30*p**2+ |
|     . 81.0*m30**2*p**4)*sin(q2)*sin(q3)**3                             |
|       t0=j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*j30x|
|     . )*j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+81.0*  |
|     . j30x+81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*p**6 |
|       t1=-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+(  |
|     . -9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*j30z|
|     . )*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y   |
|     . -81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4                     |
|       inv(2,3)=t2/(t0+(-81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos(q2)|
|     . **2*cos(q3)**2+(t1+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*  |
|     . j30y*m30**2*p**4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0|
|     . *j30y-81.0*j30z)*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3|
|     . )**4)                                                            |
|       t0=-(-j10y*j30y+(-j30y*m10+(-9.0*j10y-9.0*j30y)*m30)*p**2+(-9.0* |
|     . m10*m30-81.0*m30**2)*p**4+81.0*m30**2*p**4*cos(q2)**2*cos(q3)**2+|
|     . (j10y*j30y-j10y*j30z+((j30y-j30z)*m10+(9.0*j10y+9.0*j30y-9.0*j30z|
|     . )*m30)*p**2+(9.0*m10*m30+81.0*m30**2)*p**4)*sin(q3)**2)          |
|       t1=j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+(9.0*j10y+9.0*j30x|
|     . )*j30y)*m30)*p**2+((9.0*j30x+9.0*j30y)*m10*m30+(81.0*j10y+81.0*  |
|     . j30x+81.0*j30y)*m30**2)*p**4+(81.0*m10*m30**2+729.0*m30**3)*p**6 |
|       t2=-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*j30z)*m10+(  |
|     . -9.0*j10y*j30x+(-9.0*j10y-9.0*j30x)*j30y+(9.0*j10y+9.0*j30x)*j30z|
|     . )*m30)*p**2+((-9.0*j30x-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y   |
|     . -81.0*j30x-81.0*j30y+81.0*j30z)*m30**2)*p**4                     |
|       inv(3,3)=t0/(t1+(-81.0*j30x*m30**2*p**4-729.0*m30**3*p**6)*cos(q2)|
|     . **2*cos(q3)**2+(t2+(-81.0*m10*m30**2-729.0*m30**3)*p**6+(-81.0*  |
|     . j30y*m30**2*p**4-729.0*m30**3*p**6)*sin(q2)**2)*sin(q3)**2+((81.0|
|     . *j30y-81.0*j30z)*m30**2*p**4+729.0*m30**3*p**6)*sin(q2)**2*sin(q3|
|     . )**4)                                                            |
|c                                                                       |
|c --- Copy Entries Across Main Diagonals ---                             |
|c                                                                       |
|       do 25001 i=1,3                                                    |
|           do 25002 j=i+1,3                                              |
|               mat(j,i)=mat(i,j)                                         |
|               inv(j,i)=inv(i,j)                                         |
|25002      continue                                                     |
|25001 continue                                                          |
+------------------------------------------------------------------------+
```

6.3.3. Expression Optimization

VAXIMA contains an *optimize* command which takes an expression as argument and returns a sequence of zero or more assignments followed by another expression. The returned expression will compute the same value as the original expression after the assignments have been made, but it will do so more efficiently by avoiding recomputation of some common subexpressions. This command can be applied to expressions generated by GENTRAN by turning the *gentranopt* switch on. Thus, if we issue the command

```
(c1) on(gentranopt)$
```

followed by the commands given in subsection 6.3.1, the following will be generated:

```
Contents of file matrix.f:
+--------------------------------------------------------------------------+
|c                                                                         |
|c --- Calculate Matrix Values ---                                         |
|c                                                                         |
|      t0=p**2                                                             |
|      t1=sin(q3)**2                                                       |
|      mat(1,1)=j10y+j30y+m10*t0+18.0*m30*t0+18.0*m30*t0*cos(q2)*cos(q3)-|
|     . j30y*t1+j30z*t1-9.0*m30*t0*t1                                      |
|      t0=p**2                                                             |
|      t1=sin(q3)**2                                                       |
|      mat(1,2)=j30y+9.0*m30*t0+9.0*m30*t0*cos(q2)*cos(q3)-j30y*t1+j30z* |
|     . t1-9.0*m30*t0*t1                                                   |
|      mat(1,3)=-9.0*m30*p**2*sin(q2)*sin(q3)                              |
|      t0=p**2                                                             |
|      t1=sin(q3)**2                                                       |
|      mat(2,2)=j30y+9.0*m30*t0-j30y*t1+j30z*t1-9.0*m30*t0*t1              |
|      mat(2,3)=0.0                                                        |
|      mat(3,3)=j30x+9.0*m30*p**2                                          |
|c                                                                         |
|c --- Calculate Inverse Matrix Values ---                                 |
|c                                                                         |
|      t0=-j30x*j30y                                                       |
|      t1=-9.0*j30x                                                        |
|      t2=-9.0*j30y                                                        |
|      t3=p**2                                                             |
|      t4=m30**2                                                           |
|      t5=p**4                                                             |
|      t6=9.0*j30x                                                         |
|      t7=9.0*j30y                                                         |
|      t8=sin(q3)                                                          |
|      t9=t8**2                                                            |
|      t10=9.0*j10y+t6                                                     |
|      t11=81.0*j30y                                                       |
|      t12=m30**3                                                          |
|      t13=p**6                                                            |
|      t14=-729.0*t12*t13                                                  |
|      t15=sin(q2)**2                                                      |
```

```
      inv(1,1)=-(t0+(t1+t2)*m30*t3-81.0*t4*t5+(j30x*j30y-j30x*j30z+(t6+
     . t7-9.0*j30z)*m30*t3+81.0*t4*t5)*t9)/(j10y*j30x*j30y+(j30x*j30y*
     . m10+(9.0*j10y*j30x+t10*j30y)*m30)*t3+((t6+t7)*m10*m30+(81.0*j10y+
     . 81.0*j30x+t11)*t4)*t5+(81.0*m10*t4+729.0*t12)*t13+(-81.0*j30x*t4*
     . t5+t14)*cos(q2)**2*cos(q3)**2+(-j10y*j30x*j30y+j10y*j30x*j30z+((
     . t0+j30x*j30z)*m10+(-9.0*j10y*j30x+(-9.0*j10y+t1)*j30y+t10*j30z)*
     . m30)*t3+((t1+t2+9.0*j30z)*m10*m30+(-81.0*j10y-81.0*j30x-81.0*j30y
     . +81.0*j30z)*t4)*t5+(-81.0*m10*t4-729.0*t12)*t13+(-81.0*j30y*t4*t5
     . +t14)*t15)*t9+((t11-81.0*j30z)*t4*t5+729.0*t12*t13)*t15*t8**4)
      t0=-j30x*j30y
      t1=-9.0*j30x
      t2=-9.0*j30y
      t3=p**2
      t4=m30**2
      t5=p**4
      t6=-81.0*t4*t5
      t7=cos(q2)
      t8=cos(q3)
      t9=9.0*j30x
      t10=9.0*j30y
      t11=sin(q3)
      t12=t11**2
      t13=9.0*j10y+t9
      t14=81.0*j30y
      t15=m30**3
      t16=p**6
      t17=-729.0*t15*t16
      t18=sin(q2)**2
      inv(1,2)=(t0+(t1+t2)*m30*t3+t6+(-9.0*j30x*m30*t3+t6)*t7*t8+(j30x*
     . j30y-j30x*j30z+(t9+t10-9.0*j30z)*m30*t3+81.0*t4*t5)*t12)/(j10y*
     . j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+t13*j30y)*m30)*t3+((t9+
     . t10)*m10*m30+(81.0*j10y+81.0*j30x+t14)*t4)*t5+(81.0*m10*t4+729.0*
     . t15)*t16+(-81.0*j30x*t4*t5+t17)*t7**2*t8**2+(-j10y*j30x*j30y+j10y
     . *j30x*j30z+((t0+j30x*j30z)*m10+(-9.0*j10y*j30x+(-9.0*j10y+t1)*
     . j30y+t13*j30z)*m30)*t3+((t1+t2+9.0*j30z)*m10*m30+(-81.0*j10y-81.0
     . *j30x-81.0*j30y+81.0*j30z)*t4)*t5+(-81.0*m10*t4-729.0*t15)*t16+(
     . -81.0*j30y*t4*t5+t17)*t18)*t12+((t14-81.0*j30z)*t4*t5+729.0*t15*
     . t16)*t18*t11**4)
      t0=p**2
      t1=m30**2
      t2=p**4
      t3=sin(q2)
      t4=sin(q3)
      t5=9.0*j30y
      t6=9.0*j30x
      t7=9.0*j10y+t6
      t8=81.0*j30y
      t9=m30**3
      t10=p**6
      t11=-729.0*t9*t10
      t12=-9.0*j30x
      t13=t3**2
      inv(1,3)=-((-9.0*j30y*m30*t0-81.0*t1*t2)*t3*t4+((t5-9.0*j30z)*m30*
     . t0+81.0*t1*t2)*t3*t4**3)/(j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y
```

```
     . *j30x+t7*j30y)*m30)*t0+((t6+t5)*m10*m30+(81.0*j10y+81.0*j30x+t8)*
     . t1)*t2+(81.0*m10*t1+729.0*t9)*t10+(-81.0*j30x*t1*t2+t11)*cos(q2)
     . **2*cos(q3)**2+(-j10y*j30x*j30y+j10y*j30x*j30z+((-j30x*j30y+j30x*
     . j30z)*m10+(-9.0*j10y*j30x+(-9.0*j10y+t12)*j30y+t7*j30z)*m30)*t0+(
     . (t12-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y-81.0*j30x-81.0*j30y+
     . 81.0*j30z)*t1)*t2+(-81.0*m10*t1-729.0*t9)*t10+(-81.0*j30y*t1*t2+
     . t11)*t13)*t4**2+((t8-81.0*j30z)*t1*t2+729.0*t9*t10)*t13*t4**4)
      t0=-j30x*j30y
      t1=-9.0*j10y
      t2=-9.0*j30y
      t3=p**2
      t4=m30**2
      t5=p**4
      t6=cos(q2)
      t7=cos(q3)
      t8=9.0*j30x
      t9=9.0*j30y
      t10=sin(q2)**2
      t11=sin(q3)
      t12=t11**2
      t13=9.0*j10y+t8
      t14=81.0*j30y
      t15=m30**3
      t16=p**6
      t17=-729.0*t15*t16
      t18=-9.0*j30x
      inv(2,2)=-(-j10y*j30x+t0+(-j30x*m10+(t1-18.0*j30x+t2)*m30)*t3+(
     . -9.0*m10*m30-162.0*t4)*t5+(-18.0*j30x*m30*t3-162.0*t4*t5)*t6*t7+(
     . j30x*j30y-j30x*j30z+(t8+t9-9.0*j30z)*m30*t3+81.0*t4*t5+81.0*t4*t5
     . *t10)*t12)/(j10y*j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+t13*j30y
     . )*m30)*t3+((t8+t9)*m10*m30+(81.0*j10y+81.0*j30x+t14)*t4)*t5+(81.0
     . *m10*t4+729.0*t15)*t16+(-81.0*j30x*t4*t5+t17)*t6**2*t7**2+(-j10y*
     . j30x*j30y+j10y*j30x*j30z+((t0+j30x*j30z)*m10+(-9.0*j10y*j30x+(t1+
     . t18)*j30y+t13*j30z)*m30)*t3+((t18+t2+9.0*j30z)*m10*m30+(-81.0*
     . j10y-81.0*j30x-81.0*j30y+81.0*j30z)*t4)*t5+(-81.0*m10*t4-729.0*
     . t15)*t16+(-81.0*j30y*t4*t5+t17)*t10)*t12+((t14-81.0*j30z)*t4*t5+
     . 729.0*t15*t16)*t10*t11**4)
      t0=p**2
      t1=m30**2
      t2=p**4
      t3=sin(q2)
      t4=cos(q2)
      t5=cos(q3)
      t6=sin(q3)
      t7=9.0*j30y
      t8=9.0*j30x
      t9=9.0*j10y+t8
      t10=81.0*j30y
      t11=m30**3
      t12=p**6
      t13=-729.0*t11*t12
      t14=-9.0*j30x
      t15=t3**2
      inv(2,3)=(((-9.0*j30y*m30*t0-81.0*t1*t2)*t3-81.0*t1*t2*t4*t3*t5)*
```

```
     . t6+((t7-9.0*j30z)*m30*t0+81.0*t1*t2)*t3*t6**3)/(j10y*j30x*j30y+(
     . j30x*j30y*m10+(9.0*j10y*j30x+t9*j30y)*m30)*t0+((t8+t7)*m10*m30+(
     . 81.0*j10y+81.0*j30x+t10)*t1)*t2+(81.0*m10*t1+729.0*t11)*t12+(
     . -81.0*j30x*t1*t2+t13)*t4**2*t5**2+(-j10y*j30x*j30y+j10y*j30x*j30z
     . +((-j30x*j30y+j30x*j30z)*m10+(-9.0*j10y*j30x+(-9.0*j10y+t14)*j30y
     . +t9*j30z)*m30)*t0+((t14-9.0*j30y+9.0*j30z)*m10*m30+(-81.0*j10y
     . -81.0*j30x-81.0*j30y+81.0*j30z)*t1)*t2+(-81.0*m10*t1-729.0*t11)*
     . t12+(-81.0*j30y*t1*t2+t13)*t15)*t6**2+((t10-81.0*j30z)*t1*t2+
     . 729.0*t11*t12)*t15*t6**4)
      t0=-9.0*j10y
      t1=-9.0*j30y
      t2=p**2
      t3=m30**2
      t4=p**4
      t5=cos(q2)**2
      t6=cos(q3)**2
      t7=9.0*j10y
      t8=9.0*j30y
      t9=sin(q3)
      t10=t9**2
      t11=9.0*j30x
      t12=t7+t11
      t13=81.0*j30y
      t14=m30**3
      t15=p**6
      t16=-729.0*t14*t15
      t17=-9.0*j30x
      t18=sin(q2)**2
      inv(3,3)=-(-j10y*j30y+(-j30y*m10+(t0+t1)*m30)*t2+(-9.0*m10*m30
     . -81.0*t3)*t4+81.0*t3*t4*t5*t6+(j10y*j30y-j10y*j30z+((j30y-j30z)*
     . m10+(t7+t8-9.0*j30z)*m30)*t2+(9.0*m10*m30+81.0*t3)*t4)*t10)/(j10y
     . *j30x*j30y+(j30x*j30y*m10+(9.0*j10y*j30x+t12*j30y)*m30)*t2+((t11+
     . t8)*m10*m30+(81.0*j10y+81.0*j30x+t13)*t3)*t4+(81.0*m10*t3+729.0*
     . t14)*t15+(-81.0*j30x*t3*t4+t16)*t5*t6+(-j10y*j30x*j30y+j10y*j30x*
     . j30z+((-j30x*j30y+j30x*j30z)*m10+(-9.0*j10y*j30x+(t0+t17)*j30y+
     . t12*j30z)*m30)*t2+((t17+t1+9.0*j30z)*m10*m30+(-81.0*j10y-81.0*
     . j30x-81.0*j30y+81.0*j30z)*t3)*t4+(-81.0*m10*t3-729.0*t14)*t15+(
     . -81.0*j30y*t3*t4+t16)*t18)*t10+((t13-81.0*j30z)*t3*t4+729.0*t14*
     . t15)*t18*t9**4)
c
c --- Copy Entries Across Main Diagonals ---
c
      do 25001 i=1,3
          do 25002 j=i+1,3
              mat(j,i)=mat(i,j)
              inv(j,i)=inv(i,j)
25002     continue
25001 continue
```

VAXIMA's optimize facility is limited in that it can only optimize one expression at a time. Therefore, as can be seen in this example, subexpressions which are common to two or more assignments are needlessly regenerated.

6.3.4. Generation of Temporary Variables to Suppress Simplification

We can dramatically improve upon the efficiency of the code generated in the previous subsections by replacing expressions by temporary variables *before* computing the inverse matrix. This effectively suppresses simplification; these expressions will not be substituted into later computations. We will replace each non-zero element of VAXIMA matrix mat by a generated variable name and generate a FORTRAN assignment statement to reflect that substitution in the numerical program being generated.

The following VAXIMA session will write generated code to file matrix.f:

```
(c1) batch("matrix.v");

(c2) mat : genmatrix(mat, 3,3, 1,1)$

(c3) mat[1,1] :
     18*cos(q3)*cos(q2)*m30*p^2 - 9*sin(q3)^2*p^2*m30
     - sin(q3)^2*j30y + sin(q3)^2*j30z + p^2*m10 + 18*p^2*m30
     + j10y + j30y$

(c4) mat[2,1] :
     mat[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$

(c5) mat[3,1] :
     mat[1,3] :
     -9*sin(q3)*sin(q2)*m30*p^2$

(c6) mat[2,2] :
     -sin(q3)^2*j30y + sin(q3)^2*j30z - 9*sin(q3)^2 *m30*p^2
     + j30y + 9*m30*p^2$

(c7) mat[3,2] :
     mat[2,3] :
     0$

(c8) mat[3,3] :
     9*m30*p^2 + j30x$

(d9)                          BATCH DONE

(c10) gentranout("matrix.f")$

(c11) gentran( literal("c", cr,
                       "c --- Calculate Matrix Values ---", cr,
                       "c", cr) )$

(c12) for i:1 thru 3 do
          for j:i thru 3 do
              gentran( lrsetq(mat[i,j], mat[i,j]) )$

(c13) gentran( literal("c", cr,
                       "c --- Assign Non-Zero Matrix Values to ",
                       "Temporary Variables ---", cr, "c", cr) )$
```

```
(c14) for i:1 thru 3 do
          for j:i thru 3 do
              if mat[i,j]#0 then
              (
                  markvar( var : tempvar(false) ),
                  mat[i,j] : mat[j,i] : var,
                  gentran( eval(var) : mat[eval(i),eval(j)] )
              )$

(c15) mat;         /***  Contents of Matrix mat:  ***/

                            [ t0   t1   t2 ]
                            [              ]
(d15)                       [ t1   t3    0 ]
                            [              ]
                            [ t2    0   t4 ]

(c16) inv : mat^^(-1)$

(c17) gentran( literal("c", cr,
                       "c --- Calculate Inverse Matrix Values ---",
                       cr, "c", cr) )$

(c18) for i:1 thru 3 do
          for j:i thru 3 do
              gentran( lrsetq(inv[i,j], inv[i,j]) )$

(c19) gentran( literal("c", cr,
                       "c --- Copy Entries Across Main Diagonals ---",
                       cr, "c", cr),
               for i:1 thru 3 do
                   for j:i+1 thru 3 do
                   (   mat[j,i] : mat[i,j],  inv[j,i] : inv[i,j]   )
             )$

(c20) gentranshut("matrix.f")$
```

Contents of file matrix.f:

```
+--------------------------------------------------------------------------+
|c                                                                         |
|c --- Calculate Matrix Values ---                                         |
|c                                                                         |
|      mat(1,1)=j10y+j30y+m10*p**2+18.0*m30*p**2+18.0*m30*p**2*cos(q2)*    |
|     . cos(q3)-j30y*sin(q3)**2+j30z*sin(q3)**2-9.0*m30*p**2*sin(q3)**2    |
|      mat(1,2)=j30y+9.0*m30*p**2+9.0*m30*p**2*cos(q2)*cos(q3)-j30y*sin(   |
|     . q3)**2+j30z*sin(q3)**2-9.0*m30*p**2*sin(q3)**2                     |
|      mat(1,3)=-9.0*m30*p**2*sin(q2)*sin(q3)                              |
|      mat(2,2)=j30y+9.0*m30*p**2-j30y*sin(q3)**2+j30z*sin(q3)**2-9.0*m30|
|     . *p**2*sin(q3)**2                                                   |
|      mat(2,3)=0.0                                                        |
|      mat(3,3)=j30x+9.0*m30*p**2                                          |
|c                                                                         |
|c --- Assign Non-Zero Matrix Values to Temporary Variables ---            |
|c                                                                         |
|      t0=mat(1,1)                                                         |
|      t1=mat(1,2)                                                         |
|      t2=mat(1,3)                                                         |
|      t3=mat(2,2)                                                         |
|      t4=mat(3,3)                                                         |
|c                                                                         |
|c --- Calculate Inverse Matrix Values ---                                 |
|c                                                                         |
|      inv(1,1)=t3*t4/(-t2**2*t3+(-t1**2+t0*t3)*t4)                        |
|      inv(1,2)=-t1*t4/(-t2**2*t3+(-t1**2+t0*t3)*t4)                       |
|      inv(1,3)=-t2*t3/(-t2**2*t3+(-t1**2+t0*t3)*t4)                       |
|      inv(2,2)=(-t2**2+t0*t4)/(-t2**2*t3+(-t1**2+t0*t3)*t4)               |
|      inv(2,3)=t1*t2/(-t2**2*t3+(-t1**2+t0*t3)*t4)                        |
|      inv(3,3)=(-t1**2+t0*t3)/(-t2**2*t3+(-t1**2+t0*t3)*t4)               |
|c                                                                         |
|c --- Copy Entries Across Main Diagonals ---                              |
|c                                                                         |
|      do 25001 i=1,3                                                      |
|          do 25002 j=i+1,3                                                |
|              mat(j,i)=mat(i,j)                                           |
|              inv(j,i)=inv(i,j)                                           |
|25002     continue                                                        |
|25001 continue                                                            |
+--------------------------------------------------------------------------+
```

6.3.5. Expression Optimization & Generation of Temporary Variables

By combining expression optimization (section 6.3.3) with generation of temporary variables (section 6.3.4), we can generate code which is even more efficient than any generated thus far in this section.

By turning on the *gentranopt* flag before issuing the VAXIMA commands given in the previous subsection, the following code will be generated:

```
Contents of file matrix.f:
+--------------------------------------------------------------------------+
|c                                                                         |
|c --- Calculate Matrix Values ---                                         |
|c                                                                         |
|      t0=p**2                                                             |
|      t1=sin(q3)**2                                                       |
|      mat(1,1)=j10y+j30y+m10*t0+18.0*m30*t0+18.0*m30*t0*cos(q2)*cos(q3)-|
|     . j30y*t1+j30z*t1-9.0*m30*t0*t1                                      |
|      t0=p**2                                                             |
|      t1=sin(q3)**2                                                       |
|      mat(1,2)=j30y+9.0*m30*t0+9.0*m30*t0*cos(q2)*cos(q3)-j30y*t1+j30z* |
|     . t1-9.0*m30*t0*t1                                                   |
|      mat(1,3)=-9.0*m30*p**2*sin(q2)*sin(q3)                              |
|      t0=p**2                                                             |
|      t1=sin(q3)**2                                                       |
|      mat(2,2)=j30y+9.0*m30*t0-j30y*t1+j30z*t1-9.0*m30*t0*t1              |
|      mat(2,3)=0.0                                                        |
|      mat(3,3)=j30x+9.0*m30*p**2                                          |
|c                                                                         |
|c --- Assign Non-Zero Matrix Values to Temporary Variables ---            |
|c                                                                         |
|      t0=mat(1,1)                                                         |
|      t1=mat(1,2)                                                         |
|      t2=mat(1,3)                                                         |
|      t3=mat(2,2)                                                         |
|      t4=mat(3,3)                                                         |
|c                                                                         |
|c --- Calculate Inverse Matrix Values ---                                 |
|c                                                                         |
|      inv(1,1)=t3*t4/(-t2**2*t3+(-t1**2+t0*t3)*t4)                        |
|      inv(1,2)=-t1*t4/(-t2**2*t3+(-t1**2+t0*t3)*t4)                       |
|      inv(1,3)=-t2*t3/(-t2**2*t3+(-t1**2+t0*t3)*t4)                       |
|      t5=t2**2                                                            |
|      inv(2,2)=(-t5+t0*t4)/(-t5*t3+(-t1**2+t0*t3)*t4)                     |
|      inv(2,3)=t1*t2/(-t2**2*t3+(-t1**2+t0*t3)*t4)                        |
|      t5=-t1**2+t0*t3                                                     |
|      inv(3,3)=t5/(-t2**2*t3+t5*t4)                                       |

|c                                                                         |
|c --- Copy Entries Across Main Diagonals ---                              |
|c                                                                         |
|      do 25001 i=1,3                                                      |
|          do 25002 j=i+1,3                                                |
|              mat(j,i)=mat(i,j)                                           |
|              inv(j,i)=inv(i,j)                                           |
|25002     continue                                                        |
|25001 continue                                                            |
+--------------------------------------------------------------------------+
```

6.3.6. Comparison of Code Optimization Techniques

As previously mentioned, the VAXIMA *optimize* facility can only optimize one expression at a time, and thus, subexpressions common to two or more assignment statements are needlessly regenerated. A code optimizer /11/ designed by J. A. van Hulzen, which runs under REDUCE, is much more powerful. If we generate code with

GENTRAN under the REDUCE system in much the same way as we did in the previous subsection under VAXIMA, but call van Hulzen's code optimizer to optimize the assignments, the following code will be generated:

```
Contents of file matrix.f:
+---------------------------------------------------------------+
|c                                                              |
|c --- Calculate Matrix Values ---                              |
|c                                                              |
|      t0=cos(q3)                                               |
|      t1=cos(q2)                                               |
|      t2=sin(q3)                                               |
|      t8=t2*t2                                                 |
|      t7=p*p                                                   |
|      t5=t7*m30                                                |
|      t4=t5*t0*t1                                              |
|      t6=-j30y+j30z                                            |
|      t14=9.0*t5                                               |
|      t10=-t14+t6                                              |
|      t11=t10*t8                                               |
|      t13=t5+t4                                                |
|      mat(1,1)=18.0*t13+t11+j30y+j10y+t7*m10                   |
|      t9=t14+j30y                                              |
|      mat(2,2)=t11+t9                                          |
```

```
|      mat(1,2)=mat(2,2)+9.0*t4                                |
|      t3=sin(q2)                                              |
|      mat(1,3)=-t14*t2*t3                                     |
|      mat(2,3)=0.0                                            |
|      mat(3,3)=t14+j30x                                       |
|c                                                             |
|c --- Assign Non-Zero Matrix Values to Temporary Variables ---|
|c                                                             |
|      t0=mat(1,1)                                             |
|      t1=mat(1,2)                                             |
|      t2=mat(1,3)                                             |
|      t3=mat(2,2)                                             |
|      t4=mat(3,3)                                             |
|c                                                             |
|c --- Calculate Inverse Matrix Values ---                     |
|c                                                             |
|      t5=t3*t4                                                |
|      t6=t4*t1                                                |
|      t12=t6*t1                                               |
|      t13=t4*t0                                               |
|      t14=t2*t2                                               |
|      t8=-t14+t13                                             |
|      t15=t8*t3                                               |
|      t11=t15-t12                                             |
|      inv(1,1)=t5/t11                                         |
|      inv(1,2)=t6/t11                                         |
|      t7=-t3*t2                                               |
|      inv(1,3)=t7/t11                                         |
|      inv(2,2)=t8/t11                                         |
|      t9=t2*t1                                                |
|      inv(2,3)=t9/t11                                         |
|      t16=t1*t1                                               |
|      t10=-t16+t3*t0                                          |
|      inv(3,3)=t10/t11                                        |
|c                                                             |
|c --- Copy Entries Across Main Diagonals ---                  |
|c                                                             |
|      do 25001 i=1,3                                          |
|         do 25002 j=i+1,3                                     |
|             mat(j,i)=mat(i,j)                                |
|             inv(j,i)=inv(i,j)                                |
|25002     continue                                            |
|25001 continue                                                |
+--------------------------------------------------------------+
```

Not only was van Hulzen's Code Optimizer able to detect and consistently replace subexpressions common to more than one expression, it also improved the efficiency of the resultant code by replacing all exponentiations by optimal sequences of products.

The following table compares the efficiency of code generated in each of the previous subsections:

	+,-	*,/	**	sin,cos
6.3.1. code generation	369	645	257	73
6.3.2. segmentation	369	645	257	73
6.3.3. optimization (VAXIMA)	368	580	70	33
6.3.4. temporary variables	49	66	34	15
6.3.5. temp. vars. & opt. (VAXIMA)	47	65	20	9
6.3.6. temp. vars. & opt. (REDUCE)	21	28	0	4

REFERENCES

/1/ American National Standards Institute. "American National Standard Programming Language FORTRAN", ANS X3.9, New York. 1978.

/2/ Foderaro, J. K. and R. J. Fateman. "Characterization of VAX Macsyma", *SYMSAC '81 Proceedings*, P. S. Wang (ed.) Snowbird, Utah. August 5-7, 1981.

/3/ Foderaro, J. K. and K. L. Sklower. *The Franz LISP Manual*, University of California, Berkeley, California. September 1981.

/4/ Gates, B. L. "GENTRAN: An Automatic Code Generation Facility for REDUCE", *ACM SIGSAM Bulletin*. To Appear.

/5/ Gates, B. L. and P. S. Wang. "A LISP-Based RATFOR Code Generator", *Proceedings of the 1984 MACSYMA User's Conference*, V. E. Golden (ed.) G. E. Research & Development, Schenectady, New York. July 23-25, 1984.

/6/ Hearn, A. C. (ed.) *REDUCE User's Manual*, Version 3.0. The Rand Corporation, Santa Monica, California. April 1983.

/7/ Kernighan, B. W. "RATFOR -- A Preprocessor for a Rational Fortran", *UNIX Programmer's Manual*, Volume 2B, Seventh Edition. Bell Telephone Laboratories, Inc., Murray Hill, New Jersey. January 1979.

/8/ Kernighan, B. W. and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey. 1978.

/9/ The MATHLAB Group. *MACSYMA Reference Manual*, Version Ten. Laboratory for Computer Science, M. I. T., Cambridge, Massachusetts. December 1983.

/10/ Pavelle, R. and P. S. Wang. "MACSYMA from F to G", *Journal of Symbolic Computation*, Volume 1, Number 1. Academic Press, New York, New York. March 1985.

/11/ van Hulzen, J. A. "Code Optimization of Multivariate Polynomial Schemes: A Pragmatic Approach", *EUROCAL '83 Proceedings*, J. A. van Hulzen (ed.) Springer-Verlag LNCS Series Nr. 162. 1983.

/12/ Wang, P. S. "MACSYMA-Aided Finite Element Analysis", *Proceedings of the 1984 MACSYMA User's Conference*, V. E. Golden (ed.) G. E. Research & Development, Schenectady, New York. July 23-25, 1984.

/13/ Wang, P. S., T. Y. P. Chang, and J. A. van Hulzen. "Code Generation and Optimization for Finite Element Analysis", *EUROSAM '84 Proceedings*, J. P. Fitch (ed.) Springer-Verlag LNCS Series. July 9-11, 1984.

/14/ Wilensky, R. *LISPcraft*, W. W. Norton & Company, New York, New York. 1984.

A. TRANSLATABLE VAXIMA STATEMENTS & EXPRESSIONS

A substantial subset of all VAXIMA statements and expressions can be translated by GENTRAN into semantically equivalent code in the target numerical language[9]. This appendix is divided into two sections. The first section contains a formal definition of all translatable VAXIMA statements and expressions. The second section contains examples of several statement types translated into FORTRAN, RATFOR and C code.

A.1. Formal Definition

This section contains a formal definition of all VAXIMA statements and expressions that can be translated by GENTRAN into numerical code. First the VAXIMA user level syntax is given, and then the LISP level syntax is given.

Note: Terminals are *underlined*.

eps represents the empty string.

User Level

Preliminary Definitions

id - an identifier (i.e., atom).
string - any number of characters (excluding double quotes) enclosed in double quotes.

Expressions

Arithmetic Expressions:

exp ::= *number* | var | funcall | *abs(* exp) |
- exp | exp + exp | exp - exp |
exp * exp | exp / exp | exp ** exp |
exp ^ exp | (exp)

exp’ ::= , exp exp’ | *eps*

exp’’ ::= exp exp’ | *eps*

var ::= *id* | *id [* exp exp’]

funcall ::= *id(* arg’)

arg ::= exp | logexp | *string*

arg’ ::= arg arg’ | *eps*

arg0 ::= arg | *eps*

string0 ::= *string* | *eps*

id’ ::= *id* id’’ | *eps*

id’’ ::= , *id* id’’ | *eps*

Logical Expressions:

logexp ::= *true* | *false* | var | funcall |
exp > exp | exp >= exp | exp = exp |
exp # exp | exp < exp | exp <= exp |
not logexp | logexp *and* logexp |
logexp *or* logexp | (logexp)

Operator Precedence

Parentheses are automatically generated when expressions are translated which contain operators whose precedence in VAXIMA differs from that in the target language. Thus the meaning of the original expression is preserved.

[9] It should be noted that call-by-value parameter passing is used in VAXIMA, while call-by-address parameter passing is normally used in FORTRAN and RATFOR. GENTRAN does not attempt to simulate call-by-value passing when generating code.

For example, in VAXIMA,

```
not a = b    and    not (a = b)
```

are equivalent, whereas in C,

```
! a == b     and    (!a) == b
```

are equivalent. Therefore,

```
not a = b
```

is translated into C code which forces the VAXIMA precedence rules:

```
!(a == b)
```

Statements

stmt ::= assign | loop | cond | go | call |
return | iostmt | stmtgp

stmt' ::= , stmt stmt' | *eps*

assign ::= var : exp | *id : matrix(* row row')

row ::= [exp exp']

row' ::= , row row' | *eps*

loop ::= *for* var initval nextexp thruexp loopcond *do*
stmt

initvar ::= : exp | *eps*

nextexp ::= *step* exp | *next* exp | *eps*

thruexp ::= *thru* exp | *eps*

loopcond ::= *while* logexp | *unless* logexp | *eps*

cond ::= *if* logexp *then* stmt elsestmt

elsestmt ::= *else* stmt | *eps*

go ::= *go(* id)

call ::= *id(* exp'')

return ::= *return(* arg0)

iostmt[10] ::= var : **readonly**(string0) |
print(arg arg')

stmtgp ::= (stmt stmt')[11] |
block(stmt stmt')[12]

Subprograms

subprog ::= *id(* id') := body ;

body ::= stmt | exp | logexp

LISP Level

[10] I/O statements cannot be translated into C code.

[11] VAXIMA compound statements are translated into (undelimited) statement *sequences*.

[12] VAXIMA statement blocks are translated into statement *groups* (delimited by { and }) when RATFOR or C code is being generated. However, they are translated into (undelimited) statement *sequences* when the target language is FORTRAN.

Preliminary Definitions

id - an identifier (i.e., atom whose first character is not &).
string - an atom whose first character is &.

Expressions

Arithmetic Expressions:

exp ::= *number* | var | funcall |
((mabs) exp) | *((mminus)* exp) |
((mplus) exp exp') | *((mtimes)* exp exp') |
((mquotient) exp exp) | *((mexpt)* exp exp)

exp' ::= exp exp' | *eps*

var ::= *id* | *((id)* exp exp')

funcall ::= *((id)* arg')

arg ::= exp | logexp | *string*

arg' ::= arg arg' | *eps*

arg0 ::= arg | *eps*

string0 ::= *string* | *eps*

id' ::= *id* id' | *eps*

Logical Expressions:

logexp ::= *t* | *nil* | var | funcall |
((mgreaterp) exp exp) |
((mgeqp) exp exp) |
((mequal) exp exp) |
((mnotequal) exp exp) |
((mlessp) exp exp) |
((mleqp) exp exp) | *((mnot)* logexp) |
((mand) logexp logexp logexp') |
((mor) logexp logexp logexp')

logexp' ::= logexp logexp' | *eps*

Statements

stmt ::= assign | loop | condR | go | call |
return | iostmt | stmtgp

stmt' ::= stmt stmt' | *eps*

assign ::= *((msetq)* var exp) |
((msetq) id (($matrix) row row'))

row ::= *((mlist)* exp exp')

row' ::= row row' | *eps*

loop ::= *((mdo)* var initexp stepexp nextexp thruexp
loopcond stmt)

initexp ::= exp | *nil*

stepexp ::= exp | *nil*

nextexp ::= exp | *nil*

thruexp ::= exp | *nil*

loopcond ::= logexp | *nil*

cond ::= *((mcond)* logexp stmt *t* elsestmt)

elsestmt ::= stmt | *$false*

go ::= *((mgo) id)*

call ::= *((id)* exp')

return ::= *((mreturn)* arg0)

iostmt[13] ::= *((msetq)* var *(($readonly)* string0)) |
(($print) arg arg')

stmtgp ::= *((mprogn)* stmt stmt')[14] |
((mprog) stmt stmt')[15]

Subprograms

subprog ::= *((mdefine) ((id)* id') body)

body ::= stmt | exp | logexp

A.2. Examples

The following table contains a list of VAXIMA statement types that can be translated by GENTRAN. Examples of each statement type, along with equivalent FORTRAN, RATFOR and C code, are given.

[13] I/O statements cannot be translated into C code.

[14] VAXIMA compound statements are translated into (undelimited) statement *sequences*.

[15] VAXIMA statement blocks are translated into statement groups (delimited by { and }) when RATFOR or C code is being generated. However, they are translated into (undelimited) statement *sequences* when the target language is FORTRAN.

```
  STATEMENT TYPE    |   USER LEVEL EXAMPLE    |         LISP LEVEL EXAMPLE
--------------------+-------------------------+-----------------------------------
                    |                         |
Assignments         |                         |
                    |                         |
  - simple          |p : a*x^2 + b*x + c$     |((msetq) p
                    |                         |         ((mplus)
                    |                         |          ((mtimes) a
                    |                         |                    ((mexpt) x 2))
                    |                         |          ((mtimes) b x)
                    |                         |          c))
                    |                         |
  - matrix          |m : matrix( [u, v],      |((msetq) m
                    |            [w, x] )$    |         (($matrix) ((mlist) u v)
                    |                         |                    ((mlist) w x)))
                    |                         |
Control Structures  |                         |
                    |                         |
  Loops             |                         |
                    |                         |
  - for-step-thru   |for i:1 step 2 thru 9    |((mdo) i 1 2 nil 9 nil
                    |    do p : p + a[i]$     |       ((msetq) p ((mplus) p
                    |                         |                           ((a) i))
                    |                         |       ))
                    |                         |
  - for-next-thru   |for n:2 next n*2         |((mdo) n 2 nil ((mtimes) n 2) 500
                    |        thru 500         |       nil
                    |    do s : s + n$        |       ((msetq) s ((mplus) s n)))
                    |                         |
  - while           |while f(x)>=0            |((mdo) nil nil nil nil nil
                    |      do x : x + 0.25$   |       ((mnot) ((mgeqp) ((f) x) 0))
                    |                         |       ((msetq) x ((mplus) x 0.25))
                    |                         |)
                    |                         |
  - unless          |unless f(x)>=0           |((mdo) nil nil nil nil nil
                    |       do x : x + 0.25$  |       ((mgeqp) ((f) x) 0)
                    |                         |       ((msetq) x ((mplus) x 0.25))
                    |                         |)
                    |                         |
  Conditional       |                         |
  Transfer of       |                         |
  Control           |                         |
                    |                         |
  - if-then         |if tot>=0                |((mcond) ((mgeqp) tot 0)
                    |   then flag : true$     |         ((msetq) flag t)
                    |                         |         t
                    |                         |         $false)
                    |                         |
  - if-then-else    |if tot>=0                |((mcond) ((mgeqp) tot 0)
                    |   then flag : true      |         ((msetq) flag t)
                    |   else flag : false$    |         t
                    |                         |         ((msetq) flag nil))
                    |                         |
  Unconditional     |                         |
```

```
  Transfer of       |                        |
  Control           |                        |
                    |                        |
  - goto            |go(loop)$               |((mgo) loop)
                    |                        |
  - call            |calcz(a,b,c,z)$         |((calcz) a b c z)
                    |                        |
  - return          |return(x^2)$            |((mreturn) ((mexpt) x 2))
                    |                        |
Statement Sequences |                        |
& Groups            |                        |
                    |                        |
  - sequence        |( u : x^2, v : y^2 )$   |((mprogn) ((msetq) u ((mexpt) x 2))
                    |                        |          ((msetq) v ((mexpt) y 2))
                    |                        |)
                    |                        |
  - group           |block( u : x^2,         |((mprog) ((msetq) u ((mexpt) x 2))
                    |       v : y^2 )$       |         ((msetq) v ((mexpt) y 2)))
                    |                        |
I/O Statements      |                        |
                    |                        |
  - input           |x : readonly("Enter x")$|((msetq) x (($readonly) |&Enter x|)
                    |                        |)
                    |                        |
  - output          |print("x = ", x)$       |(($print) |&x = | x)
                    |                        |
Subprograms         |                        |
                    |                        |
  - function        |f(a,b,c,x) :=           |((mdefine) ((f) a b c x)
                    |  block( z : a*b*c + x, | ((mprog)
                    |         return(z) )$   |  ((msetq) z ((mplus)
                    |                        |               ((mtimes) a b c)
                    |                        |              x))
                    |                        |  ((mreturn) z)))
                    |                        |
  - subroutine      |f(a,b,c,x) :=           |((mdefine) ((f) a b c x)
                    |  block( z : a*b*c + x, | ((mprog)
                    |         print(z) )$    |  ((msetq) z ((mplus)
                    |                        |               ((mtimes) a b c)
                    |                        |              x))
                    |                        |  (($print) z)))
--------------------+------------------------+----------------------------------
```

```
              FORTRAN CODE                   |         RATFOR CODE
---------------------------------------+-----------------------------
                                       |
      p=a*x**2+b*x+c                   |p=a*x**2+b*x+c
                                       |
      m(1,1)=u                         |m(1,1)=u
      m(1,2)=v                         |m(1,2)=v
      m(2,1)=w                         |m(2,1)=w
      m(2,2)=x                         |m(2,2)=x
                                       |
      do 25001 i=1,9,2                 |do i=1,9,2
          p=p+a(i)                     |    p=p+a(i)
25001 continue                         |
                                       |
      n=2.0                            |for (n=2.0;!n>500.0;n=n*2.0)
25002 if (n.gt.500.0) goto 25003       |    s=s+n
          s=s+n                        |
          n=n*2.0                      |
          goto 25002                   |
25003 continue                         |
                                       |
25004 if (.not.f(x).ge.0.0) goto 25005|while (f(x)>=0.0)
          x=x+0.25                     |    x=x+0.25
          goto 25004                   |
25005 continue                         |
                                       |
25006 if (f(x).ge.0.0) goto 25007      |while (!f(x)>=0.0)
          x=x+0.25                     |    x=x+0.25
          goto 25006                   |
25007 continue                         |
                                       |
      if (.not.tot.ge.0.0) goto 25008 |if (tot>=0.0)
          flag=.true.                  |    flag=.true.
25008 continue                         |
                                       |
      if (.not.tot.ge.0.0) goto 25009 |if (tot>=0.0)
          flag=.true.                  |    flag=.true.
          goto 25010                   |else
25009 continue                         |    flag=.false.
          flag=.false.                 |
25010 continue                         |
                                       |
      goto 25011                       |goto 25011
                                       |
      functionname=x**2                |return(x**2)
      return                           |
                                       |
      u=x**2                           |u=x**2
      v=y**2                           |v=y**2
                                       |
      u=x**2                           |{
      v=y**2                           |  u=x**2
                                       |  v=y**2
                                       |}
```

```
                                   |
   write(*,*) "Enter x"            |write(*,*) "Enter x"
   read(*,*) x                     |read(*,*) x
                                   |
   write(*,*) "x = ", x            |write(*,*) "x = ", x
                                   |
   function f(a,b,c,x)             |function f(a,b,c,x)
   z=a*b*c+x                       |z=a*b*c+x
   f=z                             |return(z)
   return                          |end
   end                             |
                                   |
   subroutine f(a,b,c,x)           |subroutine f(a,b,c,x)
   z=a*b*c+x                       |z=a*b*c+x
   write(*,*) z                    |write(*,*) z
   return                          |return
   end                             |end
-----------------------------------+-------------------------------
```

```
            C CODE
-------------------------------

p=a*power(x,2)+b*x+c;

m[1][1]=u;
m[1][2]=v;
m[2][1]=w;
m[2][2]=x;

for (i=1;!(i>9);i=i+2)
    p=p+a[i];

for (n=2.0;!(n>500.0);n=n*2.0)
    s=s+n;

while (f(x)>=0.0)
    x=x+0.25;

while (!(f(x)>=0.0))
    x=x+0.25;

if (tot>=0.0)
    flag=1;

if (tot>=0.0)
    flag=1;
else
    flag=0;

goto loop;

return(power(x,2));

u=power(x,2);
v=power(y,2);

{
    u=power(x,2);
    v=power(y,2);
}
-------------------------------
```

B. LIST OF GENTRAN FUNCTIONS, SWITCHES AND VARIABLES

```
User Level Syntax                | LISP Level Syntax
---------------------------------+---------------------------------------------

Functions:                       | Functions:
                                 |
```

User Level Syntax	LISP Level Syntax
*gentran(*stmt1,stmt2,...,stmtn {,[f1,f2,...,fm]}); [16]	*(gentran '(*stmt1 stmt2 ... stmtn) '(f1 f2 ... fm))[16]
*gentranin(*f1,f2,...,fn {,[f1,f2,...,fm]});	*(gentranin '(*f1 f2 ... fn) '(f1 f2 ... fm))
*gentranout(*f1,f2,...,fn);	*(gentranout '(*f1 f2 ... fn))
*gentranpop(*f1,f2,...,fn);	*(gentranpop '(*f1 f2 ... fn))
*gentranpush(*f1,f2,...,fn);	*(gentranpush '(*f1 f2 ... fn))
*gentranshut(*f1,f2,...,fn);	*(gentranshut '(*f1 f2 ... fn))
*off(*flag1,flag2,...,flagn);	*(off '(*flag1 flag2 ... flagn))
*on(*flag1,flag2,...,flagn);	*(on '(*flag1 flag2 ... flagn))
*gendecs(*name);	*(gendecs* 'name)
genstmtno();	*(genstmtno)*
*markedvarp(*var);	*(markedvarp* 'var)
*markvar(*var);	*(markvar* 'var)
*recurunmark(*exp);	*(recurunmark* 'exp)
*tempvar(*type);	*(tempvar* 'type)
*unmarkvar(*var);	*(unmarkvar* 'var)

[16] In addition to the subset of translatable VAXIMA statements given in Appendix A, the following special functions and arguments can also be given as arguments to the *gentran* function:

begin_group	*$begin_group*
*body(*stmt1,stmt2,...,stmtn)	*(($body)* stmt1 stmt2 ... stmtn)
break()	*(($break))*
*cprocedure({*type,} name(p1,p2,...,pn))	*(($cprocedure) {*type} ((name) p1 p2 ... pn))
end()	*(($end))*
end_group	*$end_group*
*eval(*exp)	*(($eval)* exp)
*function({*type,} name(p1,p2,...,pn))	*(($function) {*type} ((name) p1 p2 ... pn))
*literal(*arg1,arg2,...,argn)	*(($literal)* arg1 arg2 ... argn)
*lrsetq(*var,exp)	*(($lrsetq)* var exp)
*lsetq(*var,exp)	*(($lsetq)* var exp)
*rsetq(*var,exp)	*(($rsetq)* var exp)
stop()	*(($stop))*
*subroutine(*name(p1,p2,...,pn))	*(($subroutine) ((*name) p1 p2 ... pn))
*type(*type, v1,v2,...,vn)	*(($type)* type v1 v2 ... vn)

Flags & Mode Switches:	Flags & Mode Switches:
c	*c*
float	*float*
fortran	*fortran*
gendecs	*gendecs*
gentranopt	*gentranopt*
gentranparser	*gentranparser*
gentranseg	*gentranseg*
ratfor	*ratfor*

Global Variables:	Global Variables:
*?ccurrind**	*ccurrind**
*?clinelen**	*clinelen**
*?fortcurrind**	*fortcurrind**
*?fortlinelen**	*fortlinelen**
*?genstmtincr**	*genstmtincr**
*?genstmtno**	*genstmtno**
*?gentranlang**	*gentranlang**
*?gentranopt**	*gentranopt**
*?maxexpprintlen**	*maxexpprintlen**
*?ratcurrind**	*ratcurrind**
*?ratlinelen**	*ratlinelen**
*?tablen**	*tablen**
*?tempvarname**	*tempvarname**
*?tempvarnum**	*tempvarnum**
*?tempvartype**	*tempvartype**