Technical Report 97-05, Department of Computer Science, Leiden University, 1997

A High-Level Language and Interpreter for the Use of Mathematical Vector Notation in PDE-Problem Specifications

*Robert van Engelen** & *Lex Wolters*
High Performance Computing Division
Dept. of Computer Science, Leiden University
PB. 9512, 2300 RA Leiden, The Netherlands
robert@cs.leidenuniv.nl

Abstract

Mathematical vector notation provides a convenient shorthand for describing scientific PDE-based models. By using several notational conventions, an expert model developer can exploit the expressive power of vector notation to formulate a PDE problem in a mathematically concise way. In this paper, we present a high-level language for describing PDEs in vector notation and an interpreter for symbolically transforming the vector equations into sets of scalar equations with respect to any selected coordinate system. The presented high-level language interpreter provides a basic set of primitive list operations and implements a recursive, implicit mapping algorithm for symbolically mapping scalar, vector, or multi-dimensional operators on vectors, matrices, and multi-dimensional objects in general. Together they form a powerful means for defining PDE vector operators and matrix/vector operations for the adoption of full vector notation in PDE specifications.

1 Introduction

This paper focuses on the parsing and interpretation of expressions in so-called mathematical vector notation. We present a high-level language for describing Partial Differential Equations (PDEs) in vector notation and describe the implementation of an interpreter for the language. The language elements are based on mathematical concepts. Mathematical vector notation provides a convenient shorthand for writing mathematical models consisting of PDEs. A high-level computer language for the specification of PDEs in vector notation requires a special form of semantics in which functions and operators are implicitly mapped on the elements of vectors and matrices. The presented high-level language and interpreter implement this form of semantics which allows for a more natural notation for specifying PDEs. The interpreter of the presented high-level language is currently being used as a preprocessor in the CTADEL code generation system for the interpretation and translation of mathematical vector notation. CTADEL is a symbolic transformation system for generating efficient architecture-dependent codes for PDE problems [7, 8, 9].

*Support was provided by the Foundation for Computer Science (SION) of the Netherlands Organization for Scientific Research (NWO) under Project No. 612-17-120.

The aim of this paper is not to provide a new parsing technique nor to describe a sophisticated user-interface for mathematical expressions for Symbolic and Algebraic Computing systems (SACs). Many efforts have been made to develop knowledge-based techniques for parsing expressions in mathematical notation. One of the main problems is that mathematical notation is an inherently ambiguous form of notation due to the frequent omission of parenthesis and the overloading of symbols with different interpretations. Furthermore, mathematical constructs like matrices have a two-dimensional layout that cannot be parsed by parsers that require a linearized form. See [12] for an extensive overview with a bibliography containing about 150 references on this subject.

Despite the efforts, a general parsing technique does not exist for interpreting mathematical notation and modern existing commercial and experimental SACs adopt their own custom computer language for mathematical expressions. Older SACs only employ a 'linear' text editor. One of the disadvantages of these type of editors is that two-dimensional mathematical expressions have to be collapsed in an unnatural way into one dimension and possibly spread over several lines of text. More advanced SACs include a front-end mathematics editor for editing mathematical expressions and a back-end visualization tool to output the results in a graphically acceptable mathematical form. These SAC interfaces often integrate a special-purpose template-based mathematics editor. Template-based editors require the programmer to fill in boxes containing mathematical expressions by selecting operators from lists or menus. Editing of expressions is performed using a pointer to pick up, move, and drop subexpressions. Example systems of this kind are Theorist [3] and CAS/PI [11] that allow both line editing and template-based editing of mathematical expressions.

For applications based on PDEs, an important requirement for symbolic expression parsing is that the parser and interpreter should accept expressions that closely resemble well-known mathematical constructs. One of the reasons is that expert PDE model developers are hesitated to learn about special computer language constructs that they should use in place of the in general more concise mathematical notation. As a typical example of a PDE problem posed in vector notation, consider the equation for time-independent flows

$$(\mathbf{V}\cdot\nabla)\,\mathbf{V} = -\frac{1}{\rho}\,\nabla p + \nu\,\nabla^2\,\mathbf{V} \tag{1}$$

which is specified in our high-level language as[1]

```
(V .* nabla) * V = - 1/rho * nabla * p + nu * nabla^2 * V
```

For the interpretation and translation to scalar form of the specification of Eq. (1), any selected coordinate system, e.g. Cartesian, polar, or cylindrical, etc., can be taken into account. By using a polar (r,θ) coordinate system and $\mathbf{V} = (\,u, v\,)^T$, Eq. (1) is transformed and presented in LaTeX typesetting as

$$\left(\begin{array}{l} u\,\frac{\partial u}{\partial r} + \frac{v}{r}\,\frac{\partial u}{\partial \theta} \\ u\,\frac{\partial v}{\partial r} + \frac{v}{r}\,\frac{\partial v}{\partial \theta} \end{array}\right) = \left(\begin{array}{l} \frac{\nu}{r}\left(\frac{\partial\left(r\,\frac{\partial u}{\partial r}\right)}{\partial r} + \frac{\partial\left(\frac{1}{r}\,\frac{\partial u}{\partial \theta}\right)}{\partial \theta}\right) - \frac{1}{\rho}\,\frac{\partial p}{\partial r} \\ \frac{\nu}{r}\left(\frac{\partial\left(r\,\frac{\partial v}{\partial r}\right)}{\partial r} + \frac{\partial\left(\frac{1}{r}\,\frac{\partial v}{\partial \theta}\right)}{\partial \theta}\right) - \frac{1}{r\,\rho}\,\frac{\partial p}{\partial \theta} \end{array}\right) \tag{2}$$

The LaTeX output is obtained by using the LaTeX package of the CTADEL system[2].

[1] When possible we prefer to use asymmetric symbols for denoting non-commuting operators, an example of which is the infix dot-product operator '.*'. This can be easily changed by the user when desired.

[2] In this way, all the examples were compiled into LaTeX for this paper.

The implementation of vector algebra and PDE operators in our language requires just a few declarations while other vector algebra packages of existing SACs would require extensive programming efforts to obtain similar operators with comparable functionality. This is due to the fact the interpreter implements a recursive, implicit mapping algorithm for implicitly mapping scalar and multi-dimensional operators on multi-dimensional objects such as vectors and matrices. As an example application, a Jacobian matrix of a vector with respect to a coordinate system is obtained in a natural way by taking the gradient of the vector. This requires first mapping the gradient operator on the vector elements and then expanding the gradient operator definition. Both are performed implicitly by the interpreter. This order of mapping and expanding is a form of execution control for symbolic evaluation. In contrast, in existing SACs PDE operators are always immediately expanded and the operator definitions contain alternatives using if-then-else constructs for checking combinations of scalar, vector, and matrix arguments. This makes the implementation hard to read and hard to adapt. Existing SACs are general-purpose systems for algebraic and symbolic computations and the provided language constructs are not always suitable for the specification of PDEs, especially when using vector notation. The possibility exists that unexpected results are obtained in the evaluation of an operator when no alternative exists for some combination of scalar/vector/matrix arguments.

Because vector notation is a very concise form of notation, notational mistakes can be easily made just by small and subtle changes in syntax. To avoid such notational mistakes, we want to impose the following constraints on the language and interpreter:

- the declarations of the language constructs should be completely *transparent* to a user so he/she can understand and easily change the definitions of the operators;
- to this end, the language should be *self-contained*, that is, it should be possible to define PDE operators and matrix/vector operations in the language itself;
- this should avoid occurrences of '*surprise experiences*' by a user for obtaining unexpected results when using PDE and matrix/vector operators;
- furthermore, the language should be *extensible* and the interpreter *open-ended*;
- hence, any *ad-hoc and fixed implementation* of the language constructs should be avoided.

Using the design concepts above, two separate packages have been written in the high-level language that implement PDE operators and matrix/vector operations:

- the '`vecalg`' package provides PDE operators and special programming constructs for adopting full vector notation for the specification of PDEs (Section 4, Appendix A), and
- the '`linalg`' package provides basic operations on matrices and vectors (Section 5, Appendix B).

For the matrix/vector syntax and language constructs these packages provide, we made an attempt to incorporate the 'most convenient' language constructs among four of the best-known commercial SACs. The implementation of the PDE operators, vector notation constructs, and matrix/vector operators make extensively use of the interpreter's implicit mapping algorithm.

The remainder of this paper is organized as follows. In Section 2, we explain the design motivations for the syntax and semantics of the high-level language and interpreter and we compare the most basic language syntax between four existing SACs. Section 3 discusses the implementation of the language interpreter and illustrates the implementation with several PDE examples. Section 4 introduces the '`vecalg`' package written in the high-level language that provides language constructs for vector notation and Section 5 introduces the '`linalg`' package for basic matrix/vector operations. In Section 6, we describe declarations for prefix, postfix, infix operators, symbolic constants, and special descriptions for alternative LaTeX output. Finally, Section 7 presents advanced features of the interpreter including imperative programming constructs.

2 Language Design

In this section we explain the design motivations for the syntax and semantics of the high-level language.

2.1 Language Syntax

We have made an attempt to combine the 'best' features among the language constructs present in the SACs MAPLE [5], MATHEMATICA [18], REDUCE [6], and MATLAB [15]. The MAPLE, MATHEMATICA, and REDUCE systems are general-purpose symbolic and algebraic computing systems (SACs) while MATLAB is a matrix-based system for scientific and engineering calculations. Each of these systems adopts a kind of mathematics-oriented language with a different syntax for representing matrices and vectors.

2.1.1 Comparing MAPLE, MATHEMATICA, REDUCE, and MATLAB Language Syntax

Figure 1 depicts example matrix and vector operations within each of the four SACs investigated. The figure illustrates matrix assignments, matrix/vector arithmetic, mapping of a scalar function on vector and matrix elements, and an attempt to map a 'divergence' operator denoted as '`div`' on the rows of a matrix. We will briefly discuss the differences between the language syntax of the four SACs.

MAPLE. Matrices and vectors are represented in MAPLE by two and one-dimensional arrays respectively, where the array contents are nested lists. A list is written using '`[`' and '`]`' brackets. For a vector, the list of the array contains the elements of the vector. For a matrix, the outermost list constains the rows of the matrix; the row elements are stored in the innermost lists. Each line of input is terminated by a '`;`' or by a '`:`'. In the latter case the output is suppressed.

Matrix and vector expressions are evaluated using the matrix evaluation function '`evalm`'. In the matrix expressions supplied to '`evalm`', scalar functions are implicitly mapped on the elements of matrices and vectors. Without '`evalm`' a dyadic '`map`' function can be used to explicitly map scalar functions onto matrices and vectors. The matrix and vector operations shown in Figure 1 are defined in the '`linalg`' package of MAPLE. With this package, '`transpose(A)`' transposes a matrix '`A`' and '`multiply(A, B)`' multiplies two matrices or vectors together which can also be written with the '`evalm`' function as '`A &* B`'. Arguments to the infix '`&*`'-operator that are neither matrices nor vectors will be considered as symbolic

MAPLE

```
> alias(Id = &*()):
> S := array([[1, 2], [3, 4]]):
> T := array([[1, 1], [2, -1]]):
> evalm(S^2 + 2 * T);
```

$$\begin{bmatrix} 9 & 12 \\ 19 & 20 \end{bmatrix}$$

```
> evalm(sin(S) &* T);
```

$$\begin{bmatrix} \sin(1)+2\sin(2) & \sin(1)-\sin(2) \\ \sin(3)+2\sin(4) & \sin(3)-\sin(4) \end{bmatrix}$$

```
> evalm((S - Id) &* array([a,b]) &* U);
```

$$[2\,b \quad 3\,a+3\,b]\,\&*\,U$$

```
> evalm(S^0);
```

$$1$$

```
> evalm(transpose(array([1, 2])) &* array([3, 4]));
```

$$11$$

```
> evalm(S + array([1, 2]));

Error, (in linalg[add]) matrix dimensions incompatible

> map(diverge, [[a(x), b(y)], [c(x), d(y)]], [x, y]);
```

$$\left[\left(\frac{\partial}{\partial x}a(x)\right)+\left(\frac{\partial}{\partial y}b(y)\right),\left(\frac{\partial}{\partial x}c(x)\right)+\left(\frac{\partial}{\partial y}d(y)\right)\right]$$

MATHEMATICA

```
In[1]:= S := {{1, 2}, {3, 4}}
In[2]:= T := {{1, 1}, {2, -1}}
In[3]:= MatrixPower[S, 2] + 2 T

Out[3]= {{9, 12}, {19, 20}}

In[4]:= Sin[S] . T

Out[4]= {{Sin[1] + 2 Sin[2], Sin[1] - Sin[2]},
>        {Sin[3] + 2 Sin[4], Sin[3] - Sin[4]}}

In[5]:= (S - IdentityMatrix[2]) . {a, b} . U

Out[5]= {2 b, 3 a + 3 b} . U

In[6]:= MatrixPower[S, 0]

Out[6]= {{1, 0}, {0, 1}}

In[7]:= Transpose[{1, 2}] . {3, 4}

Out[7]= 11

In[8]:= S + {1, 2}

Out[8]= {{2, 3}, {5, 6}}

In[9]:= Map[Div, {{a, b}, {c, d}}]

Out[9]= {Div[{a, b}], Div[{c, d}]}
```

REDUCE

```
1: Id2 := mat((1, 0), (0, 1))$
2: S := mat((1, 2), (3, 4))$
3: T := mat((1, 1), (2, -1))$
4: S^2 + 2 T;
[9   12]
[      ]
[19  20]
5: sin(S) * T;
*****
[1  2]
[    ]
[3  4]
  invalid as scalar
6: map(sin, S) * T;
[2*sin(2) + sin(1)   - sin(2) + sin(1)]
[                                     ]
[2*sin(4) + sin(3)   - sin(4) + sin(3)]
7: (S - Id2) * mat((a), (b)) * U;
***** u invalid as scalar
8: S^0;
[1  0]
[    ]
[0  1]
9: tp(mat((1), (2))) * mat((3), (4));
[11]
10: S + mat((1), (2));
***** Matrix mismatch
11: map(div, mat((a, b), (c, d)));
[div(a)  div(b)]
[               ]
[div(c)  div(d)]
```

MATLAB

```
>> S = [1 2; 3 4];
>> T = [1 1; 2 -1];
>> S^2 + 2 .* T
ans =
     9    12
    19    20
>> sin(S) * T
ans =
    2.6601   -0.0678
   -1.3725    0.8979
>> (S - eye(2)) * [a; b] * U
??? Undefined function or variable a.
>> (S - eye(2)) * ['a'; 'b'] * 'U'
ans =
        8330
       41480
>> S^0
ans =
     1     0
     0     1
>> [1; 2]' * [3; 4]
ans =
     11
>> S + [1; 2]
??? Error using ==> +
Matrix dimensions must agree.
>> div(['a' 'b'; 'c' 'd'])
ans =
     'diff(a, x) + diff(b, y)'
     'diff(c, x) + diff(d, y)'
```

Figure 1: Matrix and Vector Operations in MAPLE, MATHEMATICA, REDUCE, and MATLAB.

matrices and the '`&*`'-operation will not be evaluated. A special symbol '`&*()`' denotes the identity matrix for which the alias '`Id`' was introduced in the example. The '`linalg`' package of MAPLE further defines gradient. divergence, Laplacian, and curl operators. In the example, the explicit mapping of the '`diverge`' operator on the rows of a matrix is shown.

On evaluating the matrix expressions in '`evalm`', simplifications are performed before passing the arguments to '`evalm`', and these simplifications may not be valid for matrices! For example, '`S^0`' returns '`1`' and not the identity matrix '`&*()`'. Appropriate error messages are given when the matrices and vectors are incompatible with respect to matrix/vector operations.

MATHEMATICA. Like in the MAPLE system, matrices and vectors are represented by double lists in MATHEMATICA. However, in MATHEMATICA a list is written using the '`{`' and '`}`' brackets and the use of a matrix creation function such as '`array`' as in MAPLE is not necessary. Scalar functions are implicitly mapped on the elements of a matrix or vector. Using the '`Map`' function, a function can be explicitly mapped on the rows of a matrix. Scalar multiplication is performed by the infix '`*`'-operator which is optional in case a constant numeric operand is used. Matrix product and dot product operations are denoted both by the infix '`.`'-operator. The '`.`'-operator is not defined for scalars. The functions '`Transpose`' and '`MatrixPower`' can be used for transposing a matrix and for obtaining the power of a matrix, respectively.

Note that the addition of a matrix to a vector is allowed and results in the addition of each element of the vector to the elements of the 'corresponding' row of the matrix!

REDUCE. In REDUCE, matrices are created using a k-ary '`mat`'-operator with the k rows of the matrix as operands. Each row-operand is a comma-separated list of row elements. A vector is just a matrix with singleton rows. Since REDUCE is implemented in Rlisp, an imperative version of lisp for REDUCE, lists are formed using the '`(`' and '`)`' brackets. Each line of input is terminated by a '`;`' or by a '`$`'. In the latter case the output is suppressed.

The element-wise application of a scalar function on matrix/vector elements can only be performed explicitly using the '`map`'-operator. Like in MATHEMATICA, scalar multiplication is denoted by the infix '`*`'-operator which is optional in case a constant numeric operand is used. Besides scalar multiplication, the '`*`'-operator is overloaded to perform matrix multiplication as well. A matrix transpose is performed using the monadic '`tp`'-function.

Computing the inner product between two vectors using a matrix transpose and matrix-vector product erroneously results in a singleton matrix! Appropriate error messages are given when the matrices and vectors are incompatible for a matrix/vector operation.

The FIDE [14] package for REDUCE provides PDE-based operators for finite difference methods (not shown in Figure 1). With this package vectors can be written using '`[`' and '`]`'-brackets like in MAPLE.

Also, a vector algebra and calculus package AVECTOR [10] for REDUCE exists that provides vector algebra and a more convenient syntax and declaration style for vectors.

MATLAB. Although MATLAB is not a symbolic algebra system in the true sense, we include this system in the comparison because of its natural syntax for matrix expressions. Furthermore, MATLAB incorporates a link to the MAPLE system for symbolic manipulation such as symbolic differentiation. The symbolic expressions are supplied as MATLAB strings.

In MATLAB, like in REDUCE, scalar and matrix/vector operations are not strictly separated. The '`*`'-operator denotes both scalar multiplication and matrix product. Element-wise

operations on matrices and vectors (and therefore scalar operations as well) are denoted by prefixing the infix operator with a dot, for example '`.*`' denotes element-wise multiplication.

In the example, strings are used to represent symbolic expressions. However, the evaluation of matrix product with symbolic matrix entries results in garbage since the strings are interpreted as numeric ASCII values! For the '`div`'-operator we have shown an example evaluation using symbolic differentiation which was obtained by using the MAPLE interface of MATLAB after some string manipulations implemented in a '`div.m`' file. Finally, a line of input can be terminated with a '`;`' to suppress output.

2.1.2 Notational Trade-offs Made for the Adopted Language Syntax

The selection of syntactical constructs for the high-level language presented in this paper has been made by avoiding some notational pitfalls. One of the pitfalls to be avoided is overloading the '`*`'-operator for performing both scalar and matrix/vector products. The most important reasons for not overloading the meaning of the '`*`'-operator are that

- the algebraic properties of the scalar product and matrix-matrix or matrix-vector products are different. In the former case the product operation is commutative while in the latter case it is not. A similar fact holds for the dot product operator which is neither associative nor commutative. Making this distinction is essential for using the interpreter as a preprocessor for the CTADEL code generating system or any computer algebra system in general. Since the '`*`'-operator is associative and commutative, the operands of the '`*`'-operator may be interchanged during symbolic simplification;

- for PDEs in vector notation, a dot product operator and a scalar product operator denote recognizably distinct operations, e.g. the divergence of a vector $\mathbf{V}$ is generally denoted as $\nabla \cdot \mathbf{V}$ while the gradient of $\mathbf{V}$ is denoted as $\nabla \mathbf{V}$. Without this distinction, the meaning of the PDEs is obscure and the formulation is error prone;

- the LaTeX output of the '`*`'-operator is always the same while different LaTeX output formats are desirable for scalar, matrix-vector, and dot product operations. The type of operation performed cannot always be inferred from the context since the operands are often symbolic and untyped. Therefore the resulting LaTeX output may be incorrectly interpreted by a user.

The issue on whether operator overloading should be used in SACs in general is also discussed in [12]. Furthermore, all four SACs investigated adopt different constructs for representing vectors and matrices. In our high-level language, we have decided that vectors are written using '`[`' and '`]`' enclosing a comma-separated list comprising the elements of the vector. For example, the vector $\begin{pmatrix} 1 \\ 2 \end{pmatrix}$ is written as

```
[ 1,
  2 ]
```

Vectors can also be written 'horizontally' as a syntactical shorthand e.g. '`[ 1, 2 ]`' by omitting the line-breaks after commas. However, the meaning is the same and the LaTeX typesetting of the vector is still vertically oriented.

In accordance with the MAPLE and MATHEMATICA systems, we consider a matrix as a vector of vectors where the innermost vectors of the matrix comprise the rows of the matrix. So, for example

```
[ [ a11, a12, a13 ],
  [ a21, a22, a23 ] ]
```

represents a 2×3 matrix with symbolic elements $\mathtt{a}_{ij}$, $i = 1, 2$, $j = 1, 2, 3$. A true mathematical horizontal vector is written as a matrix of one row. For example

```
[ [ 1, 2, 3 ] ]
```

represents the vector $\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$.

2.1.3 On the Mathematical Notation using LaTeX

It should be mentioned that although the system adopts LaTeX as a means for mathematical output of the results, it will be cumbersome to use LaTeX for mathematical input as well. Although the LaTeX language [13] and more specifically $\mathcal{AMS}$-LaTeX [1] can be used for defining a uniform style for writing mathematical expressions, it is not generally possible to unambiguously interpret LaTeX input for its mathematical contents. Too often, the same LaTeX constructs are used representing different types of mathematical expressions. For example, two consecutively written symbols like fx may either denote the product of f and x, or the application of a unary function f on x when f is a known function, or may even denote just one symbol comprised by the characters f and x. Furthermore, many LaTeX variants for the same expression can be given depending on the personal flavor, backgrounds, and habits of a user. To avoid different interpretations, we must be very pedantic in the way the LaTeX input is given thereby making the specification of PDEs a very tedious task for a user to perform.

2.2 Language Semantics

The order in which scalar and vector operators are expanded and the mapping of the operators on the elements of vectors and matrices is of fundamental importance to obtain correct scalar results for PDEs specified in vector notation. For some typical compositions of scalar and vector PDE-operators such as in $\nabla \cdot \nabla(u, v)^T$, vector operators have to be implicitly mapped, for example the divergence $\nabla \cdot$ on the rows of a Jacobian matrix constructed from $\nabla(u, v)^T$, in order to have consistent results. To illustrate this, we will make use of a small example. This example clearly motivates the rationale behind the semantics of the proposed language for describing PDE-based problems.

For the example assume that a dyadic function '`diff`' represents partial differentiation and suppose that a user[3] has defined a gradient operator ∇ with respect to a Cartesian (x, y) coordinate system with the definition

```
grad(a) := [ diff(a, x),
             diff(a, y) ].
```

The '`grad`' operator takes a scalar expression and returns its gradient vector with respect to (x, y). Now consider the application of the '`grad`' operator on a vector instead of a scalar. By mathematical definition, the result should be the Jacobian matrix of the elements of the vector with respect to (x, y)

$$\nabla \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{pmatrix}$$

The application of the '`grad`' operator on vector '`[u, v]`' is specified as

[3] With 'users' we also mean those who implement packages for providing PDE operators.

```
grad([ u,
       v ])
```

Since the gradient operator only takes a scalar expression as an argument, the language semantics dictate that the gradient operator should be mapped on the elements of the vector *before* the definition of the gradient operator can be expanded. That is, the gradient operator is implicitly mapped on the vector elements which results in the intermediate form

```
[ grad(u),
  grad(v) ]
```

Thereafter, the gradient operator is applied on the scalar vector elements which results in a double list representing the Jacobian matrix:

```
[ [ diff(u, x), diff(u, y) ],
  [ diff(v, x), diff(v, y) ] ]
```

Recall that we have adopted the same conventions as in the MAPLE and MATHEMATICA systems for representing matrices as double lists where the innermost lists of the matrix comprise the rows of the matrix.

Now let us elaborate on the example with some more advanced operations. Assume that the user introduces a divergence operator operating on arbitrary two-dimensional vectors. The divergence with respect to a two-dimensional (x, y) Cartesian coordinate system can be defined as[4]

```
div([ a, b ]) := diff(a, x) + diff(b, y).
```

An example vector definition is:

```
C := [ c_1, c_2 ].
```

With these definitions, a two-dimensional diffusion problem for two chemical compounds c_1 and c_2 is specified as

```
diff(C, t) = div(nu * grad(C))
```

where '`nu`' is assumed to be a scalar (the diffusion constant). According to the language semantics, operators can only be applied to arguments that are *exactly conforming* to the definition of the operator, i.e. accepting only scalars (e.g. gradient) or accepting only vectors (e.g. divergence). The left-hand side of the PDE evaluates directly by implicit mapping of the '`diff`' function into '`[diff(c_1, t), diff(c_2, t)]`' while the right-hand side of the PDE evaluates via a combination of operator definition expansions and implicit mappings to

```
                 div(nu * grad(C))
expand C ⇒       div(nu * grad([c_1, c_2]))
map grad ⇒       div(nu * [grad(c_1), grad(c_2)])
expand grad ⇒    div(nu * [ [diff(c_1, x), diff(c_2, y)],
                            [diff(c_1, x), diff(c_2, y)] ])
map * ⇒          div([ [nu * diff(c_1, x), nu * diff(c_1, y)],
                       [nu * diff(c_2, x), nu * diff(c_2, y)] ])
map div ⇒        [ div([nu * diff(c_1, x), nu * diff(c_1, y)]),
                   div([nu * diff(c_2, x), nu * diff(c_2, y)]) ]
expand div ⇒     [ diff(nu * diff(c_1, x), x) + diff(nu * diff(c_1, y), y),
                   diff(nu * diff(c_2, x), x) + diff(nu * diff(c_2, y), y) ]
```

[4]For sake of notational convenience we will further write vectors horizontally.

These transformations correspond to the algebraic translation

$$\nabla \cdot (\nu \begin{pmatrix} \frac{\partial c_1}{\partial x} & \frac{\partial c_1}{\partial y} \\ \frac{\partial c_2}{\partial x} & \frac{\partial c_2}{\partial y} \end{pmatrix}) \overset{\text{mapping}}{\Rightarrow} \begin{pmatrix} \nabla \cdot (\nu \frac{\partial c_1}{\partial x}, \nu \frac{\partial c_1}{\partial y})^T \\ \nabla \cdot (\nu \frac{\partial c_2}{\partial x}, \nu \frac{\partial c_2}{\partial y})^T \end{pmatrix} \overset{\text{expand } \nabla\cdot}{\Rightarrow} \begin{pmatrix} \frac{\partial(\nu \frac{\partial c_1}{\partial x})}{\partial x} + \frac{\partial(\nu \frac{\partial c_1}{\partial y})}{\partial y} \\ \frac{\partial(\nu \frac{\partial c_2}{\partial x})}{\partial x} + \frac{\partial(\nu \frac{\partial c_2}{\partial y})}{\partial y} \end{pmatrix}$$

Hence, the mapping and expansion of operator definitions is performed deterministically based on the type of operator definitions given, i.e. whether the operators accept expressions of type scalar, vector, or matrix.

A similar algebraic translation for the divergence operator has been implemented in an ad-hoc way in the FIDE "FInite difference method for partial Differential Equation solving" system [14], a REDUCE package.

3 The Interpreter

The interpreter is written in SWI-Prolog [17], a public-domain Prolog implementation compatible to standard Edinburgh Prolog [4, 16]. In the next sections, we will discuss the implementation of the interpreter. Examples of PDE problems are included to illustrate the presented techniques.

In Figure 2 we have depicted an example dialogue with the interpreter for the same sample matrix and vector operations as were depicted in Figure 1 for the four SACs. In the example, input to the interpreter is denoted by a prompt '`>`' followed by definitions, assignments, and expressions to be evaluated which are possibly spread over multiple lines of input and terminated by a '.'.

In the example, the '`div`'-operator is a predefined operator for symbolically computing the divergence of a vector. To this end, the '`coordinate_system("Polar")`' command is needed for obtaining a (r, θ) polar coordinate system. The application of the '`div`'-operator on a matrix results in the application of the operator on each separate row of the matrix after which the definition of the divergence operator is expanded.

3.1 Lexical Analysis

For lexical analysis and parsing of expressions for the interpreter, the Prolog operator precedence grammar is used. The precedence and associativity of operators is also used for obtaining the correctly parenthesized LaTeX output using the LaTeX package of CTADEL. On the one hand, the advantage of using operator precedence grammars is that the precedence and associativity of new operators can be defined by a user and all built-in operators can be redefined when desired, thus allowing for maximal syntactical freedom. This is a reason why many SACs adopt operator precedence grammars. On the other hand, however, operator precedence grammars are restrictive and such grammars cannot accept syntactical constructs that more general LR-parsers are able to accept.

The grammar productions for expressions are:

```
> S := [[1, 2], [3,  4]].
> T := [[1, 1], [2, -1]].
> S^2 + 2 * T.
```

$$\begin{pmatrix} 9 & 12 \\ 19 & 20 \end{pmatrix}$$

```
> sin(S) &* T.
```

$$\begin{pmatrix} 2.66006583845926 & -0.0678264420177852 \\ -1.37248498255599 & 0.897922503367795 \end{pmatrix}$$

```
> (S - ident(2)) &* [a, b] &* U.
ERROR: matrix product with scalar
  signaled on evaluating
  (S - ident(2)) &* [a, b] &* U
> (S - ident(2)) &* [a, b].
```

$$\begin{pmatrix} 2\,b \\ 3\,a + 3\,b \end{pmatrix}$$

```
> S^0.
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
> [1, 2]‘ &* [3, 4].
```

$$11$$

```
> S + [1, 2].
ERROR: matrix + vector
  signaled on evaluating
  S + [1, 2]
> coordinate_system("Polar").
> div([[a, b], [c, d]]).
```

$$\begin{pmatrix} \frac{1}{r}\left(\frac{\partial(a\,r)}{\partial r} + \frac{\partial b}{\partial \theta}\right) \\ \frac{1}{r}\left(\frac{\partial(c\,r)}{\partial r} + \frac{\partial d}{\partial \theta}\right) \end{pmatrix}$$

Figure 2: Matrix and Vector Operations.

expr ::= *constant*
| *identifier*
| *tuple*
| *list*
| *operator*
| '(' *expr* ')'
| *deferred-op*
| *protected-expr*
| '`interpret`(' *stmt-block* ')'
| '`if`' '(' *expr* ')' '`then`' '(' *expr* ')' '`else`' '(' *expr* ')'
| *expr* '`where`' (*identifier* | *arg-list*) '`=`' *expr*

tuple ::= *expr*
| '(' *expr* ',' *tuple* ')'

operator ::= *identifier*'('*expr* [',' *expr*]* ')'
| *identifier expr*
| *expr identifier*
| *expr identifier expr*

identifier ::= '`a`'...'`z`' [*alphanum*]*
| '`A`'...'`Z`' [*alphanum*]*
| *special* [*special*]*
| ',' | ';' | '|'

wildcard ::= '_' [*alphanum*]*

alphanum ::= '`a`'...'`z`' | '`A`'...'`Z`' | '`0`'...'`9`' | '_'

special ::= '`!`' | '`#`' | '`$`' | '`&`' | '`*`' | '`+`' | '`-`' | '`.`' | '`/`' | '`:`'
| '`<`' | '`=`' | '`>`' | '`?`' | '`@`' | '`\`' | '`^`' | '`‘`' | '`~`'

The last three productions of *operator* require *identifier* to be a prefix, postfix, or infix operator, respectively. The '`where`' construct can be used to introduce a value for an identifier in the local scope of an expression. For operators, the left bracket is actually part of the operator identifier, so it is not allowed to put any spaces between the name and the left bracket.

Expressions comprising *deferred-op*, *protected-expr*, and *stmt-block* will be discussed in Section 7.

3.2 Lists

Lists provide the primary data structure for the representation of matrices and vectors. Lists are formed using '`[`' and '`]`' to surround a comma-separated list of expressions. More formally, the syntax of a list is

list ::= '`[]`'
| '`[`' *expr* [',' *expr*]* ['|' *list*] '`]`'

where '`[]`' denotes the empty list. The list construct '`[`$expr_1$`,` $expr_2$`,` ..., $expr_n$ `|` *list*`]`' denotes a list with $expr_1, \ldots, expr_n$ the first n elements ($n \geq 1$) of the list and *list* denotes a list containing the remaining elements.

3.3 Symbolic Evaluation

For the symbolic evaluation of algebraic expressions, the interpreter invokes the GPAS General-Purpose Algebraic Simplifier of the CTADEL code generation system [7]. Table 1 shows the predefined arithmetic operators and functions that are symbolically evaluated.

In the table, the precedence of operators are shown. Operator precedences are between

prec.	operator	description
200	*expr* `^` *expr*	power
200	*expr* `**` *expr*	alias for `^` power
400	*expr* `//` *expr*	string concatenation
400	*expr* `*` *expr*	multiplication
400	*expr* `/` *expr*	division
400	*expr* `div` *expr*	integer division
400	*expr* `mod` *expr*	integer modulus
400	*expr* `rem` *expr*	remainder of integer division
450	`+` *expr*	unary plus
450	`-` *expr*	unary minus
500	*expr* `+` *expr*	addition
500	*expr* `-` *expr*	subtraction
590	*expr* `max` *expr*	maximum value
590	*expr* `min` *expr*	minimum value
700	*expr* `<` *expr*	less than (numeric, logical, string)
700	*expr* `<=` *expr*	less or equal than (numeric, logical, string)
700	*expr* `<>` *expr*	not equal to (numeric, logical, string)
700	*expr* `==` *expr*	equal to (numeric, logical, string)
700	*expr* `>` *expr*	greater than (numeric, logical, string)
700	*expr* `>=` *expr*	greater or equal than (numeric, logical, string)
910	*expr* `and` *expr*	logical conjunction
912	*expr* `imp` *expr*	logical implication
914	*expr* `or` *expr*	logical disjunction
916	*expr* `eqv` *expr*	logical equivalence
916	*expr* `xor` *expr*	logical exclusive-or
900	`not` *expr*	logical negation
	function	description
	`abs(`*expr*`)`	absolute value and complex modulus
	`acos(`*expr*`)`	arccosine in radians
	`acosh(`*expr*`)`	hyperbolic arccosine in radians
	`asin(`*expr*`)`	arcsin in radians
	`asinh(`*expr*`)`	hyperbolic arcsine in radians
	`atan(`*expr*`)`	arctangent in radians
	`atan(`$expr_1$`,` $expr_2$`)`	arctangent of $expr_1/expr_2$
	`atanh(`*expr*`)`	hyperbolic arctangent in radians
	`complex(`*expr*`)`	*expr* as complex number
	`complex(`$expr_1$`,` $expr_2$`)`	complex number $expr_1 + i\,expr_2$
	`conj(`*expr*`)`	complex conjugate
	`cos(`*expr*`)`	cosine in radians
	`cosh(`*expr*`)`	hyperbolic cosine in radians
	`ceil(`*expr*`)`	ceiling
	`exp(`*expr*`)`	exponent
	`fac(`*expr*`)`	faculty
	`floor(`*expr*`)`	floor
	`frac(`*expr*`)`	fractional part
	`gcd(`*expr*`,` *expr*`)`	greatest common divisor
	`im(`*expr*`)`	imaginary part of complex *expr*
	`lcm(`*expr*`,` *expr*`)`	least common multiple
	`ln(`*expr*`)`	natural logarithm, alias for `log`
	`log(`*expr*`)`	natural logarithm
	`log10(`*expr*`)`	base 10 logarithm
	`re(`*expr*`)`	real part of complex *expr*
	`real(`*expr*`)`	real number
	`round(`*expr*`)`	round to nearest integer
	`sign(`*expr*`)`	sign
	`sin(`*expr*`)`	sine in radians
	`sinh(`*expr*`)`	hyperbolic sine in radians
	`sqrt(`*expr*`)`	square root
	`tan(`*expr*`)`	tangent in radians
	`tanh(`*expr*`)`	hyperbolic tangent in radians
	`trunc(`*expr*`)`	truncation to integer

Table 1: Predefined Operators and Functions.

1 (high precedence) and 1200 (low precedence) as defined by standard Prolog conventions[5]. Operators can also be written using functional syntax, for example, '`max(a,a+1)`' and '`not(a and b)`'.

3.4 Some Remarks on Using Prefix, Postfix, and Infix Operators

Some parsing problems may occur for specific combinations of prefix, postfix, and infix operators. The padding with spaces and the use of brackets may be necessary to separate prefix, postfix, and infix operators[6] or using brackets to avoid ambiguity. These problems are typical for SACs adopting operator precedence grammars, see e.g. [12].

Three types of parsing problems can be encountered that are to be avoided by padding with spaces or by using brackets. With respect to mathematical expressions, an unfortunate situation exists in the use of unary minus and plus with all other infix operators, like in '`2*-3`', '`x<-y`', or '`x^-1`'. In all of these cases we have to use spaces, i.e. write '`2* -3`','`x< -y`', and '`x^ -1`' instead or use brackets like in '`x^(-1)`'. On the other hand, this forces a user to be more precise in formulation: the CTADEL simplifier internally uses an infix '`<-`' operator for its term-rewrite system, hence it would not be wise to type infix '`<`' and prefix '`-`' next to each other in a mathematical expression as the expression will be accepted by the parser, however with a totally different meaning.

Parsing problems may be encountered when

1. using an infix operator $\oplus$ that is immediately followed by a prefix operator $\ominus$: in $expr \oplus \ominus expr$ the $\oplus$ and $\ominus$ should be separated;

2. likewise, using an infix operator $\oplus$ that is immediately preceded by a postfix operator $\ominus$: in $expr \ominus \oplus expr$ the $\ominus$ and $\oplus$ should be separated;

3. using an infix operator $\oplus$, a prefix operator f and a symbol of the same name f: the problem is that $f \oplus expr$ may either denote $(f) \oplus expr$ or $f(\oplus(expr))$ depending on the precedence of f. In this case spaces and/or brackets should be used.

3.5 The Built-in Basic Set of List Operators

Table 2 shows the built-in list operators that are implemented in the interpreter. Their arguments may be scalar algebraic expressions or lists representing vectors and matrices.

The precedence of the infix '`:`' operator is 100 which means that brackets should be used for both arguments when they consists of a prefix, postfix, or infix operator. Other examples are:

- '`10-2*(0 to 4))`' generates '`[10, 8, 6, 4, 2]`';
- '`[1, 2, 3, 4]:3`' gives '`3`';
- '`[[a,b],[c,d]]:2`' gives '`[c,d]`', the second row of the matrix;
- '`[[a,b],[c,d]]:(2,1)`' gives '`c`' (second row, first column);

[5] For more details on syntactical conventions for the operator precedence grammar used in Prolog systems, the reader is referred to a textbook, e.g. [4, 16].

[6] Exceptions to this rule are the infix '`,`', '`;`', and '`|`' operators due to the Prolog implementation.

(*n_1* `to` *n_2*)	generate a list containing integers n_1 up to and including n_2
`append(`*$list_1$*`,` *$list_2$*`)`	concatenate lists
`apply(`*deferred-op*`)`	apply deferred *op*, see Section 7.4
`apply(`*deferred-op*`,` *tuple*`)`	apply deferred *op* on arguments in *tuple*, see Section 7.4
`apply(`*tuple*`,` *deferred-op*`)`	apply deferred *op* on arguments in *tuple*, see Section 7.4
`drop(`*n*`,` *list*`)`	drop first *n* elements from *list*
`eval(`*expr*`)`	evaluate a {}-protected expression, see Section 7.5
`fill(`*n*`,` *expr*`)`	generate a list containing *n* duplicates of *expr*
`flatten(`*expr*`)`	flatten all nested lists to one list
`foldl(`*identifier*`,` *list*`)`	left-reduce *list* with binary operator *identifier*
`foldr(`*identifier*`,` *list*`)`	right-reduce *list* with binary operator *identifier*
`length(`*list*`)`	length of *list*
`map(`*expr*`)`	explicit mapping of *expr*'s outer operator on list-arguments
`nops(`*expr*`)`	count number of operands of *expr*'s outer operator
`op(`*n*`,` *expr*`)`	$n = 0$: return name of *expr*'s outer operator as a string $n > 0$: return n^{th} argument of *expr*'s outer operator
`reduce(`*identifier*`,` *list*`)`	left/right reduce *list* with left/right associative binary operator *identifier*
`reverse(`*list*`)`	reverse *list*
`subscript(`*$identifier_1$*`,` *$identifier_2$*`)`	create identifier composed of $identifier_1$ subscripted with $identifier_2$
`take(`*n*`,` *list*`)`	take first *n* elements from *list*
list`:`*n*	list indexing: return n^{th} element of *list*
list`:`*tuple*	list indexing: return element of *list* indexed by *tuple*
$list_1$`:`*$list_2$*	subset of $list_1$ as indexed by $list_2$

Table 2: Built-in Operations.

- '`map([[a,b],[c,d]]:1)`' gives '`[a,c]`', the first column of the matrix;
- '`A:(2 to 4)`' gives rows 2 to 4 of a matrix '`A`';
- '`V:reverse(2 to 4)`' gives elements 2 to 4 of a vector in reverse order;
- '`A:reverse(2 to 4)`' gives rows 2 to 4 of a matrix '`A`' in reverse order;
- '`op(0, f(a,b,c))`' gives string '`"f"`';
- '`op(2, f(a,b,c))`' gives '`b`';
- '`map(drop(2, A))`' drops the first two columns of a matrix '`A`';
- '`reduce(+, fill(10, 1))`' simply gives '`10`' (sum of ten one's);
- '`subscript([u, v], t)`' gives '`[u_t, v_t]`' since '`subscript`' is mapped on '`[u, v]`'.

Note that the '`:`' index operator can be composed from left to right, hence '`[[a,b],[c,d]]:2:1`' yields '`c`' (second row, first column).

3.6 The Interpreter's Implicit Mapping Mechanism

The most basic form of vector notation denotes the implicit mapping of a scalar function or operator on the elements of vectors as a shorthand. More specifically, we want to exploit the following notational conventions for the application of scalar functions and operators:

- a scalar function applied to a vector is identical to the application of the function to each of the elements of the vector. That is, a monadic scalar function f is implicitly mapped onto the elements of the vector:

$$f\left(\begin{array}{c}\alpha_1\\ \vdots\\ \alpha_n\end{array}\right) \overset{\text{map}}{\Rightarrow} \left(\begin{array}{c}f(\alpha_1)\\ \vdots\\ f(\alpha_n)\end{array}\right)$$

where α_i, $i = 1, \ldots, n$, $n \geq 1$, are scalar algebraic expressions;

- if one of the arguments of a function is scalar and the other vector, the scalar argument is replicated along the vector upon mapping the function on the elements of the vector:

$$g\left(\begin{array}{c}\alpha_1\\ \vdots\\ \alpha_n\end{array}\right), \beta) \overset{\text{map}}{\Rightarrow} \left(\begin{array}{c}g(\alpha_1, \beta)\\ \vdots\\ g(\alpha_n, \beta)\end{array}\right)$$

for any dyadic scalar function g, and where α_i, $i = 1, \ldots, n$, $n \geq 1$, and β are scalar algebraic expressions;

- if two of the arguments to a scalar function are vectors, the function is mapped onto the elements of both vectors in parallel:

$$h\left(\begin{array}{c}\alpha_1\\ \vdots\\ \alpha_n\end{array}\right), \left(\begin{array}{c}\beta_1\\ \vdots\\ \beta_n\end{array}\right)) \overset{\text{map}}{\Rightarrow} \left(\begin{array}{c}h(\alpha_1, \beta_1)\\ \vdots\\ h(\alpha_n, \beta_n)\end{array}\right)$$

for any dyadic scalar function h, and where α_i and β_i, $i = 1, \ldots, n$, $n \geq 1$, are scalar algebraic expressions. This mapping is valid if the vectors are of the same length. Otherwise, a conformability violation is detected and an error message is given.

By definition, mapping scalar functions on 'empty' vectors ($n = 0$ in the above) is prohibited and is signaled as an error. Furthermore, this implicit mapping applies to any k-ary scalar function by using the same conventions as listed above. Thus, for example, $f(0, -\sin((1, 2, 3)^T), (4, 5, 6)^T)$ maps to $(f(0, -\sin(1), 4), f(0, -\sin(2), 5), f(0, -\sin(3), 6))^T$ by application of multiple mappings.

Since matrices are represented as vectors of vectors, the mapping of scalar functions on matrices is performed by mapping the function on the outer vector first and then *recursively* on the innermost vectors until only scalar arguments remain. In principle, all functions and operators are treated as scalar except for functions and operators that are defined with formal matrix/vector arguments.

3.7 Example 1

This example PDE problem illustrates a simple application of the implicit mapping algorithm. Consider the time-dependent Euler equation for an inviscid, compressible flow in a two-dimensional geometry, which can be written in conservation law form (flux form) in an (x, y) Cartesian coordinate system as

$$\frac{\partial}{\partial t}\left(\begin{array}{c}\rho\\ \rho u\\ \rho v\\ \rho E\end{array}\right) + \frac{\partial}{\partial x}\left(\begin{array}{c}\rho u\\ \rho u^2\\ \rho u v\\ u(\rho E + p)\end{array}\right) + \frac{\partial}{\partial y}\left(\begin{array}{c}\rho v\\ \rho u v\\ \rho v^2\\ v(\rho E + p)\end{array}\right) = 0 \tag{3}$$

where ρ is the mass density, p the pressure, E the total energy, and (u, v) are the (x, y) components of the flow velocity, respectively. The above system of equations is closed via the

equation of state

$$p = \rho(\gamma - 1)(E - \frac{1}{2}(u^2 + v^2)) \tag{4}$$

where γ is the ratio of specific heats of the medium. Eq. (3) can be specified as

```
  diff(rho * [ 1, u, v, E ], t)
+ diff([ rho * u, rho * u ^ 2, rho * u * v, u * (rho * E + p) ], x)
+ diff([ rho * v, rho * u * v, rho * v ^ 2, v * (rho * E + p) ], y) = 0.
```

The evaluation of the example equation proceeds by implicit mapping resulting in a vector of four scalar equations:

```
[ diff(rho,     t) + diff(rho * u,           x) + diff(rho * v,           y) = 0,
  diff(rho * u, t) + diff(rho * u ^ 2,       x) + diff(rho * u * v,       y) = 0,
  diff(rho * v, t) + diff(rho * u * v,       x) + diff(rho * v ^ 2,       y) = 0,
  diff(rho * E, t) + diff(u * (rho * E + p), x) + diff(v * (rho * E + p), y) = 0 ]
```

Note that the '`diff`' operator is scalar and therefore mapped onto the elements of the vectors. The result represents the scalar form of Eq. (3).

3.8 Definitions

Operators that act on scalars, vectors, or matrices can be defined by using the appropriate list syntax for the formal arguments of the operator. More specifically, the syntax of a definition is

def	::=	*head* '`:=`' *body*
head	::=	*identifier*
	|	*identifier* '`:`' *integer*
	|	*identifier* '`:`' '`(`' *expr* '`)`'
	|	*identifier*'`(`'*arg* ['`,`' *arg*]* '`)`'
	|	*identifier arg*
	|	*arg identifier*
	|	*arg identifier arg*
	|	*head-list*
head-list	::=	'`[`' *head* ['`,`' *head*]* '`]`'
body	::=	*expr*
	|	*expr* '`when`' *expr*
	|	'`procedure(`' ['`local`' *identifier* ['`,`' *identifier*]* '`;`'] *stmt-block* '`)`'
arg	::=	*identifier*
	|	*constant*
	|	*wildcard*
	|	*arg-list*
arg-list	::=	'`[]`'
	|	'`[`' *arg* ['`,`' *arg*]* ['`|`' *arg*] '`]`'

where the last three productions for *head* require *identifier* to be a prefix, postfix, or infix operator, respectively.

The left-hand side of an assignment can be a matrix or vector indexed using the infix '`:`' operator upon which the indexed element will be changed only.

Expressions assigned to (indexed) identifiers are always evaluated first before being assigned to an identifier, while the body of an operator is only evaluated after the operator definition has been expanded.

The '`when`' keyword can be used for conditionally expanding operator definitions. Only when the condition given by the right argument of '`when`' evaluates to '`true`', the expression given by the left argument of '`when`' is returned. For example,

```
f(u) := u when u>0.
```

Here, '`f(1)`' gives '`1`' while '`f(0)`' and '`f("hello")`' remain unexpanded.

A '`procedure`' keyword identifies the declaration as a procedure instead of a function. Procedures are explained in Section 7.1.

3.9 Example 1 (Revisited)

This example illustrates the use of vector definitions and vector operators. Reconsider the time-dependent Euler equation Eq. (3). By introducing three vector fields

$$\mathbf{w} = \begin{pmatrix} \rho \\ \rho\, u \\ \rho\, v \\ \rho\, E \end{pmatrix}; \quad \mathbf{f} = \begin{pmatrix} \rho\, u \\ \rho\, u^2 \\ \rho\, u\, v \\ u\,(\rho\, E + p) \end{pmatrix}; \quad \mathbf{g} = \begin{pmatrix} \rho\, v \\ \rho\, u\, v \\ \rho\, v^2 \\ v\,(\rho\, E + p) \end{pmatrix}$$

Eq. (3) can be rewritten as

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{f}}{\partial x} + \frac{\partial \mathbf{g}}{\partial y} = 0. \tag{5}$$

The three vectors are defined by

```
w := rho * [ 1, u, v, E ].
f := [ rho * u, rho * u ^ 2, rho * u * v, u * (rho * E + p) ].
g := [ rho * v, rho * u * v, rho * v ^ 2, v * (rho * E + p) ].
```

Note that '`w`' evaluates to the vector '`[rho, rho * u, rho * v, rho * E]`' by implicit mapping of the scalar product operator '`*`'. With these vector definitions, Eq. (5) can be specified as

```
diff(w, t) + diff(f, x) + diff(g, y) = 0
```

Alternatively, we can specify Eq. (5) with '`f`' and '`g`' vector operators acting on vector '`w`':

```
f([ w1, w2, w3, w4 ]) := [ w2, w2^2 / w1, w2 * w3 / w1, w2 / w1 * (w4 + p) ].
g([ w1, w2, w3, w4 ]) := [ w3, w2 * w3 / w1, w3^2 / w1, w3 / w1 * (w4 + p) ].
```

Then, for

```
diff(w, t) + diff(f(w), x) + diff(g(w), y) = 0
```

the same result is obtained.

3.10 Example 2

This example illustrates the definition and use of a divergence operator for a PDE problem. Consider the two-dimensional advection of an entity ψ in an incompressible medium, i.e. $\nabla \cdot \mathbf{V} = 0$ for $\mathbf{V} = (\,u, v\,)^T$ which is the velocity vector field of the fluid flow at each point in space. This PDE problem is generally written in flux form as

$$\frac{\partial \psi}{\partial t} = \nabla \cdot (\psi\, \mathbf{V}) \tag{6}$$

As usual, $\nabla = (\,\frac{\partial}{\partial x}, \frac{\partial}{\partial y}\,)^T$ assuming a Cartesian (x, y) coordinate system.

First, we introduce a vector-operator '`div`' representing the divergence $(\nabla\cdot)$ operation, i.e. we write Eq. (6) as $\frac{\partial \psi}{\partial t} = \mathrm{div}(\psi\, \mathbf{V})$. The definition of the divergence operator within a (x, y) Cartesian coordinate system is

```
div([ a, b ]) := diff(a, x) + diff(b, y).
```

Note that the divergence operator '`div`' takes a two-dimensional vector and returns a scalar expression. Now, Eq. (6) is given by

```
V := [ u, v ].
diff(psi, t) = div(psi * V).
```

By implicit mapping of '`psi * V`' and expansion of '`div`' this results in

```
diff(psi, t) = diff(psi * u, x) + diff(psi * v, y)
```

In LaTeX typesetting this is

$$\frac{\partial \psi}{\partial t} = \frac{\partial (\psi\, u)}{\partial x} + \frac{\partial (\psi\, v)}{\partial y}$$

3.11 The Implicit Mapping Mechanism for Vector and Matrix Operators

The implicit mapping mechanism handles vector-operators like the divergence operator used in Example 2 in a similar way as scalar operators. The difference, however, is that the arguments of an operator are checked for conformability between the *formal* arguments, given by the definition of the operator, and the *actual* arguments supplied. The formal and actual arguments are conformable if they are of the same type, i.e. scalar, vector, or matrix. The conformability check ensures that an operator that takes a vector as an argument, e.g. a divergence operator, will only be applied to a vector. Otherwise, if the argument is a matrix, the operator is implicitly mapped onto the rows of the matrix first, and then applied. If the argument is scalar, the operator is not applied and the expression with the operator is returned as is.

Using the implicit mapping mechanism, the translation of a vector equation or expression to a set of scalar equations or expressions proceeds in a combination of expanding vector and matrix objects, mapping of operators on vectors and matrices if necessary, and expansion of operator definitions. More specifically, for the implicit mapping mechanism with vector-operators we use the following notational conventions:

- for a monadic operator $\mathbf{f}$ that takes a vector as a formal argument while supplied with a matrix as an actual argument, $\mathbf{f}$ is applied to each row of the matrix:

  $$\mathbf{f}\left(\begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1m} \\ \vdots & \ddots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nm} \end{pmatrix}\right) \overset{\text{map}}{\Rightarrow} \begin{pmatrix} \mathbf{f}((\alpha_{11}, \cdots, \alpha_{1m})^T) \\ \vdots \\ \mathbf{f}((\alpha_{n1}, \cdots, \alpha_{nm})^T) \end{pmatrix}$$

  where α_{ij}, $i = 1, \ldots, n$, $n \geq 1$, $j = 1, \ldots, m$, are scalar algebraic expressions;

- for a dyadic operator $\mathbf{g}$ that takes a vector of length m as the first formal argument and a scalar as the second formal argument while the actual arguments constitute a matrix and a vector, respectively, $\mathbf{g}$ is applied simultaneously to each row of the matrix and to each of the elements of the vector. This requires that the number of rows n of the given matrix is the same as the length of the vector supplied:

  $$\mathbf{g}\left(\begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1m} \\ \vdots & \ddots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{nm} \end{pmatrix}, \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}\right) \overset{\text{map}}{\Rightarrow} \begin{pmatrix} \mathbf{g}((\alpha_{11}, \cdots, \alpha_{1m})^T, \beta_1) \\ \vdots \\ \mathbf{g}((\alpha_{n1}, \cdots, \alpha_{nm})^T, \beta_n) \end{pmatrix}$$

  where α_{ij} and β_i, $i = 1, \ldots, n$, $n \geq 1$, $j = 1, \ldots, m$, are scalar algebraic expressions;

- for a dyadic operator **h** that takes two vectors of length ℓ and m respectively as formal arguments while the actual arguments are two matrices, the operator **h** is applied to each row of both of the matrices in parallel. This requires that both matrices have the same number of rows n:

$$\mathbf{h}(\begin{pmatrix} \alpha_{11} & \cdots & \alpha_{1\ell} \\ \vdots & \ddots & \vdots \\ \alpha_{n1} & \cdots & \alpha_{n\ell} \end{pmatrix}, \begin{pmatrix} \beta_{11} & \cdots & \beta_{1m} \\ \vdots & \ddots & \vdots \\ \beta_{n1} & \cdots & \beta_{nm} \end{pmatrix}) \stackrel{\text{map}}{\Rightarrow} \begin{pmatrix} \mathbf{h}((\alpha_{11}, \cdots, \alpha_{1\ell})^T, (\beta_{11}, \cdots, \beta_{1m})^T) \\ \vdots \\ \mathbf{h}((\alpha_{n1}, \cdots, \alpha_{n\ell})^T, (\beta_{n1}, \cdots, \beta_{nm})^T) \end{pmatrix}$$

  where α_{ij} and β_{ik}, $i = 1, \ldots, n$, $n \geq 1$, $j = 1, \ldots, \ell$, $k = 1, \ldots, m$, are scalar algebraic expressions;

Note that the notational conventions for vector-operators defined above are basically identical to the notational conventions for scalar operators as can be seen by promoting the scalars to vectors and the vectors to matrices and comparing the resulting definitions.

3.12 The Mapping Algorithm

The mapping algorithm proceeds through the following steps:

1. evaluate the actual arguments supplied to the operator;
2. check conformability between the actual arguments (i.e. scalar, vector, or matrix and their rank) with the formal arguments for each alternative definition of the operator;
3. if the arguments are conformable, apply the definition, take the resulting expression, and goto 1.
4. else, if all arguments are scalar, stop;
5. else, map the operator on the actual list-arguments;
6. goto 2.

An operator may have several alternative definitions. Only one definition is selected depending on the structural correspondence between the formal and actual arguments which is ensured by an argument conformability check. For example, in this way one, two, and three-dimensional divergence operators can be defined sharing the same operator name '`div`':

```
div(a)           := diff(a, x).                          % 1D scalar divergence
div([ a, b ])    := diff(a, x) + diff(b, y).             % 2D divergence
div([ a, b, c ]) := diff(a, x) + diff(b, y) + diff(c, z). % 3D divergence
```

Note that the algorithm adopts strict evaluation as opposed to the lazy evaluation implemented in most functional languages. That is, the arguments are evaluated before an operator is expanded, e.g.

```
div(nu * [u, v])
```

step 1 gives

```
div([nu * u, nu * v])
```

step 2 finds the second definition of '`div`' for two-dimensional divergence and step 3 results in

```
diff(nu * u, x) + diff(nu * v, y)
```

The result is scalar and symbolic evaluation stops here.

3.13 Example 3

The following example illustrates the implicit mapping of a divergence operator on the rows of a Jacobian matrix constructed in the process of translating the example problem into scalar form. Consider a two-dimensional diffusion problem with two chemical compounds c_1 and c_2 in a medium with viscosity constant ν (possibly space dependent). The two parabolic PDEs describing the problem are respectively

$$\frac{\partial c_1}{\partial t} = \nabla \cdot (\nu \nabla c_1) + r_1(c_1, c_2) \tag{7}$$

$$\frac{\partial c_2}{\partial t} = \nabla \cdot (\nu \nabla c_2) + r_2(c_1, c_2) \tag{8}$$

where the r_i, $i = 1, 2$, are so-called reaction terms. We can write Eqs. (7) and (8) as a single vector equation

$$\frac{\partial \mathbf{C}}{\partial t} = \nabla \cdot (\nu \nabla \mathbf{C}) + \mathbf{R} \tag{9}$$

where $\mathbf{C} = (\, c_1, c_2 \,)^T$ and $\mathbf{R} = (\, r_1(c_1, c_2), r_2(c_1, c_2) \,)^T$. Again, the specification of the two-dimensional gradient and divergence operators is

```
grad(u)       := [ diff(u, x), diff(u, y) ].
div([ u, v ]) := diff(u, x) + diff(v, y).
C             := [ c_1, c_2 ].
R             := [ r_1(c_1, c_2), r_2(c_1, c_2) ].
```

with the two vectors '`C`' and '`R`'. The PDE in Eq. (9) can be specified as

```
diff(C, t) = div(nu * grad(C)) + R
```

This is translated into the form

```
  [ diff(c_1, t),
    diff(c_2, t) ]
= [ diff(nu * diff(c_1, x), x) + diff(nu * diff(c_1, y), y) + r_1(c_1, c_2),
    diff(nu * diff(c_2, x), x) + diff(nu * diff(c_2, y), y) + r_2(c_1, c_2) ]
```

The output in LaTeX is

$$\begin{pmatrix} \frac{\partial c_1}{\partial t} \\ \frac{\partial c_2}{\partial t} \end{pmatrix} = \begin{pmatrix} \frac{\partial \left(\nu \frac{\partial c_1}{\partial x}\right)}{\partial x} + \frac{\partial \left(\nu \frac{\partial c_1}{\partial y}\right)}{\partial y} + \mathrm{r}_1(c_1, c_2) \\ \frac{\partial \left(\nu \frac{\partial c_2}{\partial x}\right)}{\partial x} + \frac{\partial \left(\nu \frac{\partial c_2}{\partial y}\right)}{\partial y} + \mathrm{r}_2(c_1, c_2) \end{pmatrix}$$

4 Using the `vecalg` Vector Algebra Package

In the previously presented examples, we have tacitly assumed that the user is familiar with defining his own gradient and divergence operators. Since these operators are used quite often in the specification of PDEs, we provide PDE operators in the '`vecalg`' package. The '`vecalg`' package defines the prefix operators

`grad`	gradient	∇
`div`	divergence	$\nabla\cdot$
`curl`	curl	$\nabla\times$
`lapl`	Laplacian	Δ or ∇^2

The package implements the operators independent of a coordinate system. The actual coordinate system for the PDE operators can be redefined by the user without reloading the package. To this end, two special vectors are used for specifying a coordinate system: the '`coordinates`' and '`coefficients`' vectors that respectively comprise the vector of spatial coordinates and metric coefficients (i.e. scale factors) of the coordinate system. Both vectors should be of the same rank. When the user assigns new values to these vectors, either explicitly or implicitly using the '`coordinate_system`' command, he can use any of the PDE operators listed above with the redefined coordinate system. For example, the PDE operators are cast into a two-dimensional Cartesian (x, y) coordinate system by

```
coordinates  := [x, y].
coefficients := [1, 1].
```

or by including the '`coordinate_system("2D Cartesian")`' command which executes these definitions. The coordinate system can be changed into another system or sub-coordinate system. For example, the PDE operators are cast into a polar (r, θ) coordinate system by

```
coordinates  := [r, theta].
coefficients := [1, r].
```

or by including the '`coordinate_system("Polar")`' command which executes these definitions.

The '`vecalg`' package also demonstrates the usefulness of the implicit mapping mechanism implemented in the interpreter. For example, the definition of the prefix '`grad`' operator is:

```
grad X := diff(X, coordinates) / coefficients.
```

Since the partial derivative operator '`diff`' and division '`/`' are assumed to be scalar, the '`diff`' operator is implicitly mapped on the elements of the '`coordinates`' and '`coefficients`' vectors. Thereby obtaining the gradient vector of the scalar argument. The definition of the one, two-, and three-dimensional divergence operators are:

```
div X            := diff(X, coordinates:1) / coefficients:1.
div [X1, X2]     := ( diff(coefficients:2 * X1, coordinates:1)
                    + diff(coefficients:1 * X2, coordinates:2)
                    ) / (coefficients:1 * coefficients:2).
div [X1, X2, X3] := ( diff(coefficients:2 * coefficients:3 * X1, coordinates:1)
                    + diff(coefficients:3 * coefficients:1 * X2, coordinates:2)
                    + diff(coefficients:1 * coefficients:2 * X3, coordinates:3)
                    ) / (coefficients:1 * coefficients:2 * coefficients:3).
```

where '`coordinates:1`' yields the first element of vector '`coordinates`', etc. With the above definitions, the definition of the Laplacian is simply:

```
lapl X := div grad X.
```

The complete source of the '`vecalg`' package is given in Appendix A.

4.1 Using Vector Notation

Besides the provisions for changing the coordinate system for PDE operators, the '`vecalg`' package also provides a form of vector notation for specifying PDEs. For the vector notation, the symbol '`nabla`' can be used as a symbolic constant denoting the symbol ∇ in combination with vectors, scalars, and the operators

`+`	sum
`*`	product
`.*`	dot product
`#*`	cross product
`^2`	square

The following basic notational conventions are supported:

`nabla *` *expr*	=	`grad` *expr*
`nabla .*` *expr*	=	`div` *expr*
`nabla #*` *expr*	=	`curl` *expr*
`nabla^2 *` *expr*	=	`lapl` *expr*
`nabla .* nabla`	=	`nabla^2`

Note that it is still essential to use the scalar-product operator '`*`' in combination with the '`nabla`' symbol.

Using combinations of the '`nabla`', '`+`', '`*`', '`.*`', and '`#*`' symbols, PDEs can be specified in vector notation, e.g. '`([u, v] + nu * nabla) .* [p, q]`' which gives '`nu * (diff(p, x) + diff(q, y)) + p * u + q * v`' using a Cartesian (x, y) coordinate system.

4.2 Example 4

Consider $\frac{\partial \phi}{\partial t} = (\nabla \cdot (d\,\nabla))\,\phi$ in a (x, y, z) Cartesian coordinate system. The PDE can be specified as

```
include vecalg.
coordinates  := [x, y, z].
coefficients := [1, 1, 1].
PDE := diff(phi, t) = (nabla .* (d * nabla)) * phi.
```

In the example, an identifier '`PDE`' is used to store the PDE. Expressions assigned to identifiers are always evaluated first, hence, the coordinate system must be defined before the assignment to '`PDE`'. Now, '`PDE`' evaluates to

```
diff(phi, t) =
     diff(d * diff(phi, x), x) + diff(d * diff(phi, y), y) + diff(d * diff(phi, z), z)
```

4.3 Example 5

In this example we illustrate how so-called 'equation functions' can be used for storing equations permanently so they can be reused with different coordinate systems. When equations are stored with functions, the equations are *not* evaluated until the function is expanded in contrast to assignments to identifiers. In this way we are able to specify the equations *before* a coordinate system is defined. When an appropriate coordinate system is defined, the functions can be expanded and the resulting PDEs in vector notation are cast into the coordinate system and translated into scalar form.

The flow of a fluid through space is described by two fundamental equations of hydrodynamics, the continuity equation and the Navier Stokes equation:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho\,\mathbf{V} = 0 \tag{10}$$

$$\frac{\partial \mathbf{V}}{\partial t} = -(\mathbf{V} \cdot \nabla)\,\mathbf{V} - \frac{1}{\rho}\nabla p + \nu\,\nabla^2\,\mathbf{V} \tag{11}$$

where ρ is the mass density of the fluid, ν the kinematic viscosity, and $\mathbf{V}$ is the velocity of the fluid at each point in space. In general, the pressure p is given in terms of the density and temperature through an equation of state. Furthermore, we assume that the temperature is constant throughout the fluid and thereby omit the need for an equation embodying the conservation of energy.

In this example, we will consider incompressible, time-independent flows for which the time derivatives in Eqs. (10) and (11) are set to zero and the density ρ is constant. The equations are rewritten into

$$\nabla \cdot \mathbf{V} = 0 \tag{12}$$

$$(\mathbf{V} \cdot \nabla)\,\mathbf{V} = -\frac{1}{\rho}\,\nabla p + \nu\,\nabla^2\,\mathbf{V} \tag{13}$$

First, we specify Eqs. (12) and (13) using '`equation`' functions as

```
include vecalg.
equation("Continuity")    := nabla .* V = 0.
equation("Navier Stokes") := (V .* nabla) * V = - 1/rho * nabla * p + nu * nabla^2 * V.
```

Then, we define a two-dimensional Cartesian coordinate system and a vector '`V`':

```
coordinates  := [x, y].
coefficients := [1, 1].
V := [u, v].
```

Using these definitions, the evaluation of '`equation("Continuity")`' results in the LaTeX output

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$

and '`equation("Navier Stokes")`' results in the LaTeX output

$$\begin{pmatrix} u\,\frac{\partial u}{\partial x} + v\,\frac{\partial u}{\partial y} \\ u\,\frac{\partial v}{\partial x} + v\,\frac{\partial v}{\partial y} \end{pmatrix} = \begin{pmatrix} \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) - \frac{1}{\rho}\,\frac{\partial p}{\partial x} \\ \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right) - \frac{1}{\rho}\,\frac{\partial p}{\partial y} \end{pmatrix}$$

In a similar way the equations can be cast into a polar coordinate system by redefining the '`coordinates`' and '`coefficients`' vectors:

```
coordinates  := [r, theta].
coefficients := [1, r].
```

now '`equation("Continuity")`' results in

$$\frac{1}{r}\left(\frac{\partial(r\,u)}{\partial r} + \frac{\partial v}{\partial \theta}\right) = 0$$

and '`equation("Navier Stokes")`' yields

$$\begin{pmatrix} u\,\frac{\partial u}{\partial r} + \frac{v}{r}\,\frac{\partial u}{\partial \theta} \\ u\,\frac{\partial v}{\partial r} + \frac{v}{r}\,\frac{\partial v}{\partial \theta} \end{pmatrix} = \begin{pmatrix} \frac{\nu}{r}\left(\frac{\partial\left(r\,\frac{\partial u}{\partial r}\right)}{\partial r} + \frac{\partial\left(\frac{1}{r}\,\frac{\partial u}{\partial \theta}\right)}{\partial \theta}\right) - \frac{1}{\rho}\,\frac{\partial p}{\partial r} \\ \frac{\nu}{r}\left(\frac{\partial\left(r\,\frac{\partial v}{\partial r}\right)}{\partial r} + \frac{\partial\left(\frac{1}{r}\,\frac{\partial v}{\partial \theta}\right)}{\partial \theta}\right) - \frac{1}{r\,\rho}\,\frac{\partial p}{\partial \theta} \end{pmatrix}$$

5 Using the `linalg` Linear Algebra Package

The '`linalg`' package provides basic vector and matrix operations. Since all the operators can be redefined, in principle MAPLE, MATHEMATICA, REDUCE, or MATLAB compatibility can be obtained by rewriting the necessary matrix and vector definitions. The matrix and vector operators are

expr `+` *expr*	scalar and element-wise addition
expr `-` *expr*	scalar and element-wise subtraction or negation
expr `*` *expr*	scalar and element-wise product
expr `/` *expr*	scalar and element-wise division
expr `^` *expr*	scalar and matrix power, with '*expr* `^T`' matrix transpose
expr `.*` *expr*	dot product
expr `#*` *expr*	cross product
expr `&*` *expr*	matrix-vector and matrix-matrix product
expr `	matrix transpose (note: backquote!)
expr `\` *expr*	matrix-vector solve (like MATLAB left division)
expr `++` *expr*	matrix and vector concatenation
expr `//` *expr*	matrix and vector stacking, and scalar string concatenation

Identity and zero matrices are generated by

`ident(`*rank*`)`	identity matrix
`zero(`*rank*`)`	zero square matrix
`zero(`*rowrank*`,` *colrank*`)`	zero matrix

6 Declarations

In this section we describe the declarative language constructs for introducing prefix, postfix, and infix operator syntax, for defining symbolic constants and inert operators, and for providing LaTeX descriptions for functions and operators for obtaining special forms of LaTeX output. More formally, the productions for *decl* are:

decl ::= *syntax-decl*
| *inert-decl*
| *latex-decl*

Declaration and statement-blocks comprise the two main parts of the interpreter's input:

input ::= *decl* '.'
| *stmt-block* '.'

Each of which is terminated by a '.'. Statement blocks consist of definitions, expressions to be evaluated, and statements which are further discussed in Section 7.1.

6.1 Prefix, Postfix, and Infix Operator Declarations

In the '`vecalg`' package the PDE operators '`grad`', '`div`', '`curl`', and '`lapl`' are declared as prefix operators and the '`#*`' and '`.*`' as infix operators. The declaration of a prefix, postfix, or infix operator has the form

syntax-decl ::= 'prefix' *identifier* ['has precedence(' *prec* ')']
| 'postfix' *identifier* ['has precedence(' *prec* ')']
| 'infix' *identifier* ['is associative(' *assoc* ')'] ['has precedence(' *prec* ')']

prec ::= 'none'
| *integer*
| *identifier*

assoc ::= 'left'
| 'right'
| 'none'
| *identifier*

where *prec* is either an integer denoting the Prolog operator precedence between 1 and 1200 (see also Table 1), '`none`' in which case prefix, postfix, or infix syntax will be disabled, or the identifier of an already defined prefix, postfix, or infix operator whose precedence should be used. If the operator is not already declared prefix, postfix, or infix, the default precedence for prefix and postfix operators is 100 and for infix operators 450 (between the precedences of '`+`' and '`*`'). For *assoc*, '`left`' or '`right`' denote left/right-associativity, '`none`' denotes non-associativity (default). An identifier for *assoc* denotes an already declared infix operator whose associativity should be used. For example,

```
prefix grad has precedence(+).
infix .* is associative(none) has precedence(450).
```

declares prefix syntax for '`grad`' with the same precedence as unary plus and infix syntax for the non-associative dot-product operator '`.*`' with precedence 450 (both are the default associativity and precedence for infix operators).

6.2 Defining Symbolic Constants and Inert Operators

Constants are integers, floating point numbers, strings, and explicitly declared named constants. Named constants can be parameterized in which case we will refer to these constants as '*inert operators*'. The productions for *constant* are

constant ::= 'true'
| 'false'
| 'infinity'
| 'nil'
| 'undefined'
| *number*
| *string*
| *identifier*
| *identifier* '(' *arg* [',' *arg*]* ')'
| *identifier arg*
| *arg identifier*
| *arg identifier arg*

where the last three productions require the inert operator *identifier* to be a prefix, postfix, or infix operator. A *number* is a signed integer or signed floating point number with 15 digits precision. A *string* is a sequence of characters enclosed within double quotes ('`"`'). Constants '`nil`' and '`undefined`' represent a so-called null ($\top$) value and an undefined value ($\bot$), respectively.

Inert operators serve as place-holders for expressions and are *not* subject to implicit mapping. Furthermore, it is prohibited to define a body for an explicitly declared constant or inert operator.

The declaration of symbolic constants and inert operators is of the form

inert-decl ::= *identifier* '`is inert`'
| *identifier* '`arity`' *integer* '`is inert`'
| *identifier* '`(`' *arg* ['`,`' *arg*]* '`)`' '`is inert`'

The first production declares a constant, the second and third productions both declare inert operators where '`arity` *integer*' denotes the number of arguments of the operator and in the last production *arg* are dummy arguments.

6.2.1 Using Symbolic Constants and Inert Operators

Constants and inert operators can be used as formal arguments in an operator definition. In the following example, '`x`' and '`y`' are named constants (coordinates) and a special 'domain' data structure is introduced using two inert infix operators '`=`' and '`..`':

```
x        is inert.
y        is inert.
infix =  has precedence(964) is inert.
infix .. has precedence(600) is inert.
```

Note that for convenience, several declarations for the same function/operator can be combined into one declaration. Hence '`infix = has precedence(964) is inert.`' is legal.

The inert definitions are useful for defining alternative implementations for functions and operators that depend on the type of data structures given as actual arguments. To this end, the formal arguments for an operator definition may contain constants and inert operators. For example, to implement different methods of integration in different directions, we can define

```
int(U, x = A .. B) := sum(U, i = floor(n*A) .. ceil(n*B)).
int(U, y = A .. B) := sum(U, j = floor(m*A)+1 .. ceil(m*B)).
```

where we assume that '`n`' and '`m`' are the number of grid points in the x and y directions, respectively. In the example the '`int`' definitions yield different summations for different '`x`' and '`y`' coordinates. So

```
int(int(u * v, x = 1 .. 10), y = 0 .. 1)
```

yields

```
sum(sum(u * v, i = floor(n) .. ceil(10 * n)), j = 1 .. ceil(m))
```

Note that the infix '`=`' is inert which also implies that the operator is not subject to implicit mapping. Hence, for example the equation '`[u, v] = [1, 2]`' is not further evaluated.

Introducing new named constants should be done with great care as the formal arguments of the operator definitions that follow may be affected. For example, after the above definitions are given, any '`x`' or '`y`' used as an identifier for a formal argument will be taken literally! To avoid such problems, we suggest that by convention all identifiers for the formal arguments should start with an upper case letter.

6.2.2 The Search-Order of Definitions

Recall from Section that the implicit mapping algorithm selects a definition of an operator for operator expansion depending on the structural correspondence between the actual and formal arguments. Understanding the selection method is especially crucial for the adoption

of multiple definitions with inert operators as arguments. The idea is that more specific definitions should always be applied before more general ones. To this end, the selection of a definition, when multiple definitions for an operator are given, proceeds by searching the definitions in lexicographical order. The lexicographical order applies to each argument of the definition's head from left to right: numeric constants first in numerical order, then symbolic constants in alphabetical order, then inert operators in alphabetical order, and last, any pure formal arguments. For inert operators with the same name and arity, the lexicographical order is applied recursively on their arguments from left to right. In this way, more specific definitions of an operator are found first before more general ones. For example, the following definitions may be given in any order:

```
1:  f(N, M)          := N - M.
2:  f(N, 0)          := N.
3:  f(0, M)          := -M.
4:  f(N, x = L .. U) := (U - L + 1) * N.
5:  f(N, X = Y)      := f(N, Y).
6:  f(0, X = Y)      := 0.
7:  f(N, x = 0 .. U) := (U + 1) * N.
```

For expansion of function '`f`', the search order of the definitions above is 6, 3, 2, 7, 4, 5, and 1. So, '`f(0, 0)`' expands definition 3 (where 2 could also be applied, but whose place in the order comes after definition 3), '`f(0, x = 1 .. 3)`' expands definition 4, '`f(1, y = 0 .. 3)`' expands definition 6.

Expansion of an operator can be masked with a '`when`' construct, for example:

```
g(0, M) := undefined when M > 0.
g(N, M) := N + M.
```

In which case a definition for expansion is searched by taking the lexicographical search order into account. However, when the condition of a '`when`' construct does not evaluate to '`true`', the definition is skipped and another applicable definition for expansion is searched for. For the example above, application '`g(0, 1)`' results in '`undefined`' while '`g(0, 0)`' results in '`0`'. Note that the condition for a '`when`' construct should evaluate to '`true`' before the definition can be expanded. This means that '`g(0, a)`' evaluates to '`a`' because the condition '`a > 0`' cannot be evaluated unless '`a`' has a numeric value.

6.3 LaTeX Descriptions

In principle, the LaTeX package of Citadel adopts mathematical style for most symbols having a mathematical meaning such as '`alpha`', '`nabla`', and '`forall`'. In general, by using underscores the LaTeX output of identifiers appears with all underscore extensions printed as subscripts. So, `T_0` gives T_0, `flow_liquid_1` gives $flow_{liquid_1}$, and `test_beta` gives $test_\beta$. Identifiers may contain of Greek upper and lower case letters which are are output as Greek symbols. In addition to Greek symbols, other special symbols can be used:

Identifier	Math symbol
aleph	$\aleph$
hbar	$\hbar$
imath	$\imath$
jmath	$\jmath$
l	ℓ
Re	$\Re$
Im	$\Im$
varepsilon	ε
vartheta	ϑ
varpi	ϖ
varrho	ϱ
varsigma	ς
varphi	φ

So, for example `l_alpha` gives ℓ_α and `hbar0` gives $\hbar 0$ but `hbarbara` gives *hbarbara*.

Annotated identifiers are identifiers with (several) identifier extensions added using underscores. Below is a table of annotations:

Annotation	Math symbol	Example	
_angle	$\angle$	rot_angle	$rot_{\angle}$
_bot	$\bot$	low_bot	$low_{\bot}$
_dprime	$''$	u_dprime	u''
_flat	$\flat$	d_flat	$d^{\flat}$
_infty	∞	T_infty	T_{∞}
_minus	$-$	f_minus	f^{-}
_natural	$\natural$	d_natural	$d^{\natural}$
_plus	$+$	f_plus	f^{+}
_prime	$'$	u_prime	u'
_sharp	$\sharp$	d_sharp	$d^{\sharp}$
_star	$*$	xi_star	ξ^{*}
_top	$\top$	hi_top	$hi_{\top}$

Annotations can be composed, for example `d_star_infty` gives d^*_∞ and `rot_prime_angle` gives $rot'_{\angle}$.

Like annotations, math accents are given as identifier extensions, possibly combined with annotiations. Below is a table of accents:

Accent	Example	
_acute	a_acute	$\acute{a}$
_bar	a_bar	$\bar{a}$
_breve	a_breve	$\breve{a}$
_check	a_check	$\check{a}$
_ddot	a_ddot	$\ddot{a}$
_dot	a_dot	$\dot{a}$
_grave	a_grave	$\grave{a}$
_hat	a_hat	$\hat{a}$
_tilde	a_tilde	$\tilde{a}$
_vec	a_vec	$\vec{a}$

So, for example `a_hat_star_infty` gives $\hat{a}^*_\infty$ and `u_prime_vec` gives $\vec{u'}$ which is different from `u_vec_prime` $\vec{u}'$.

The default LaTeX output of identifiers can be overruled by explicit LaTeX descriptions. There are three distinguished ways in which a special LaTeX description can be given for an object to override the 'standard' LaTeX output:

1. the most simple form is to associate a LaTeX string with the name of an object using one of the forms:

latex-decl ::= *identifier* 'has latex_name(' *string* ')'
| *identifier* 'arity' *integer* 'has latex_name(' *string* ')'
| *identifier* '(' *arg* [',' *arg*]* ')' 'has latex_name(' *string* ')'

where '**arity** *integer*' denotes the operator's arity and *arg* are dummy arguments. Examples are:

```
true        has latex_name("\sc t").
grad(_)     has latex_name("\nabla").
div arity 1 has latex_name("\nabla\cdot").
infix  .*   has latex_name("\cdot").
infix  ..   is associative(none) has precedence(600) has latex_name("\,.\,.\,").
```

Note that the '**has latex_name**' can be combined with other declarations.

2. a description for typesetting the operator and its arguments has the form:

latex-decl ::= *head* 'latex' ['precedence(' *prec* ');'] *latex-body*
latex-body ::= *string*
| *expr*
| *expr* ',' *latex-body*

The description is similar to an operator definition although the body represents the LaTeX output. The precedence of the LaTeX description controls the placement of brackets, which depends on the precedence of the other operators used in an expression. In this case, the use of '**none**' prohibits the placement of brackets. The '**precedence**' keyword is optional. For prefix, postfix, and infix operators the default precedence is the precedence of the operator. For functions, the default precedence is 1200, which means that brackets will always be placed around the function. Strings in the *latex-body* are taken literally as LaTeX commands. Examples are:

```
X #* Y             latex X, "\times", Y.
abs(X)             latex precedence(none); "\left|", X, "\right|".
int(E, X = A .. B) latex precedence(none); "\int_", A, "^", B, E, "\,d", X.
sum(E, I = A .. B) latex precedence(501);  "\sum_{", I, "=", A, "}^", B, E.
```

in which the last two declarations also make use of the domain data structure described in the previous section.

3. the most advanced form allows so-called LaTeX environments for functions and operators to be created to enable two-dimensional mathematical typesetting with LaTeX. The general form of the declaration is:

latex-decl ::= *head* 'latex' *latex-body* ';' *latex-environment-list* ';' *latex-body*

The first and second *latex-body* consists of the begin and end-commands of the LaTeX environment. The *latex-environment-list* comprises a list of local LaTeX definitions, each consisting of *head* ':=' '[' *latex-body* ']' where *latex-body* is a LaTeX description only valid local to the environment. Strings in *latex-body* are taken as LaTeX commands, which can be changed by using '$STRING(*string*)' to print a string. The local definitions are recursively expanded which can be prohibited by using '$ESCAPE('*expr*)' to escape from the local definitions in the environment.

The local definitions construct the environment, which is best illustrated using two examples:

```
(_ ; _) latex "\begin{array}{l}";
              [ (S1 ; S2) := [S1, "\\", S2]
              ];
              "\end{array}".
```

This description can be used to obtain a LaTeX typesetting for the vertical alignment of statements separated by the infix ';' operator[7].

The following example shows how a LaTeX environment for conditional expressions is created:

```
infix \\ is associative(right) has precedence(962).
infix if is associative(right) has precedence(960).
postfix otherwise.
_ \\ _ latex "\left\{\begin{array}{ll}";
             [ X \\ Y      := [X, "\\", Y]
             , X if C      := [$ESCAPE(X), "&{\bf if~}", $ESCAPE(C)]
             , X otherwise := [$ESCAPE(X), "&{\bf otherwise}"]
             ];
             "\end{array}\right.".
```

The infix '`\\`' operator constructs a case statement together with infix '`if`' and postfix '`otherwise`' operators. The arguments of the local '`if`' and '`otherwise`' definitions should escape the environment or an illegal LaTeX layout will be obtained for nested conditional expressions. For example

```
u = a/b if b > 0 and c \\
    a   if b == 0 \\
    -a  otherwise
```

is output in LaTeX as:

$$u = \begin{cases} \frac{a}{b} & \textbf{if } b > 0 \wedge c \\ a & \textbf{if } b == 0 \\ -a & \textbf{otherwise} \end{cases}$$

An example of a nested conditional expressions is:

```
u = ( a if b > 0 \\
      b otherwise   ) if c \\
    1/a               otherwise
```

which is output in LaTeX as:

$$u = \begin{cases} \begin{cases} a & \textbf{if } b > 0 \\ b & \textbf{otherwise} \end{cases} & \textbf{if } c \\ \frac{1}{a} & \textbf{otherwise} \end{cases}$$

7 Special Programming Constructs

In this section we present some more advanced programming constructs. These constructs extend the interpreter to a full programming language with imperative language features.

[7] Brackets should be used in '`(S1 ; S2) :=`' because the precedence of ';' is higher than ':='.

7.1 Procedures and Statement Blocks

The interpreter accepts sequences of statements, called a *statement blocks*. Basically, a statement block is a sequence of statements and expressions separated by a '`;`', where the value of the block is the last evaluated expression.

stmt-block ::= *stmt*
| *stmt* '`;`' *stmt-block*

stmt ::= '`foreach`' *identifier* '`in`' *expr* '`do`' *stmt*
| '`clear`' *identifier*
| '`clear`' *identifier* '`arity`' *integer*
| '`clear`' *identifier* '`(`' *arg* ['`,`' *arg*]* '`)`'
| '`remove`' *identifier*
| '`remove`' *identifier* '`arity`' *integer*
| '`remove`' *identifier* '`(`' *arg* ['`,`' *arg*]* '`)`'
| '`restart`'
| '`return(`' *expr* '`)`'
| '`(`' *stmt-block* '`)`'
| *def*
| *expr*

The '`clear`' keyword clears the definition for an identifier or function while '`remove`' completely removes the definition and obliterates all declarations given for an identifier or function. The '`foreach`' construct iterates a statement by assigning each element of a list (hence *expr* should evaluate to a list) to a local iteration variable not visible outside the construct. The '`return`' keyword can only be used within a procedure to return a value. '`return(nil)`' can be used to exit a procedure without returning a result value and '`return(undefined)`' can be used to exit a procedure with an undefined value.

A procedure executes a statement block instead of returning an expression like a 'normal' function or operator. Procedures are declared using a '`procedure`' keyword as a body of the operator definition. An optional value can be returned using a '`return`' keyword. An example procedure '`fib`' computes Fibonacci numbers:

```
fib(N) :=
    procedure(
      local a, b;
      [a, b] := 1;
      foreach i in 2 to N - 1 do
        [a, b] := [b, a + b];
      return(b)
    ).
```

Note that the expression '`2 to N - 1`' generates a list of integers from '`2`' up to and including '`N - 1`'. Procedure '`fib`' is scalar, hence '`fib`' is implicitly mapped on a vector: '`fib(1 to 5)`' gives vector '`[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`'.

7.2 Recursive Functions

The syntactical construct for lists '`[`*expr* `|` *list*`]`' allows for defining list-recursive functions and operators. For example, with this construct the dot product operator is defined in the '`linalg`' package as

```
[X]      .* [Y]      := X * Y.
[X | Xs] .* [Y | Ys] := X * Y + (Xs .* Ys).
```

Here, the 'singular' formal arguments '`X`' and '`Y`' denote scalar elements and the 'plural' formal arguments '`Xs`' and '`Ys`' denote the remaining part of the vectors.

Since the implicit mapping algorithm ensures that the elements of the vector arguments are scalar on application of the dot product, these definitions only apply to vectors. If matrices are supplied, implicit mapping is performed first:

```
[ [a11, a12],
  [a21, a22] ] .* [ [b11, b12],
                    [b21, b22] ]
```

evaluates to

```
[ a11 * b11 + a12 * b12,
  a21 * b21 + a22 * b22 ]
```

Note that when scalar arguments are supplied, the dot product operator is not applied at all as is the case for '`1 .* 2`'.

7.3 Some Efficiency Considerations for Recursive Functions

For writing recursive functions, the efficiency optimizations that are applicable to functional languages in general should be employed. Consider for example the computation of Fibonacci numbers by

```
fib(0) := 0.
fib(1) := 1.
fib(N) := fib(N-2) + fib(N-1).
```

This definition, however, is very inefficient and takes a number of steps to compute '`fib(N)`' roughly proportional to the value of '`fib(N)`', see e.g. [2] from which this example has been adopted. A more efficient implementation that takes only '`N`' steps to compute '`fib(N)`' is

```
fib(N) := A where [A, _] = twofib(N).
twofib(0) := [0, 1].
twofib(N) := [B, A + B] where [A, B] = twofib(N-1).
```

Furthermore, it is advised to use the built-in list primitives when possible. For example

```
fac(N) := reduce(*, 1 to N).
```

is an efficient implementation for computing faculty numbers.

7.4 Deferred-Operator Expansion

An operator can be protected from expansion by using the prefix '\' operator. This construct is very useful when expansion of an operator should be deferred to a later stage when operators are passed through several symbolic operations. More formally, an expression of the form

deferred-op ::= '\' *operator*

is called a 'deferred operation'. The '`apply`' construct applies deferred operations:

- `apply(`*deferred-op*`)`: apply deferred operation. Examples are:

```
> apply(\sqrt(2)).
1.41421356237310
> a := 10; b := a + apply(\a) where a = 2.
12
```

- `apply(`*deferred-op*`,` *tuple*`)`: apply deferred operation with extra last arguments in *tuple*. Examples are:

```
> apply(\ ^(2), x).
2 ^ x
> plus := \(+); apply(plus, (x, 2)).
x + 2
> plus := (+); apply(\plus, (x, 2)).
plus(x, 2)
> plus := (+); apply(plus, (x, 2)).
apply(+, (x, 2))
```

  Note the difference between the last three applications. In the last application, '`apply(+, (x, 2))`' cannot be evaluated.

- `apply(`*tuple*`,` *deferred-op*`)`: apply deferred operation with extra first arguments in *tuple*. Examples are:

```
> gradient := \diff([ x, y ]); apply(u, gradient).
[diff(u, x), diff(u, y)]
> apply(1 to 4, [ \ +(2), \ -(2), \ *(2), \ /(2) ]).
[3, 0, 6, 2]
```

  Here too, '`\ +(2)`' protects '`+(2)`' from being evaluated to '`2`'.

where the last example shows that '`apply`' is mapped first on both lists in parallel, i.e. '`apply(1, \ +(2))`' = 3, '`apply(2, \ -(2))`' = 0, etc.

7.5 Protecting Expressions from Evaluation

An expression can be protected from evaluation by enclosing it in '{' and '}' brackets. This may be necessary for expressions that have to be passed onto operators literally. For example, passing an expression to be interpreted as an assignment:

```
a := 1;
b := interpret(s) where s = (a := 2).
```

would result in garbage:

```
ERROR [eval]:
  1 is constant and cannot be assigned
ERROR [eval]:
  Error interpreting 1 := 2
s where s = (1 := 2)
```

This problem can be remedied by expression protection for '`a := 2`':

```
b := interpret(s) where s = {a := 2}.
```

results in the assigment of 2 to '`a`' as well as to '`b`'. A protected expression can be forced to be evaluated using the '`eval`' function.

7.6 Simple IO Functions

The built-in IO functions are:

`format(`*string*`)`	write SWI-Prolog formatted *string* to terminal
`format(`*string*`, `*list*`)`	write *list* of expressions using SWI-Prolog *string* format
`input(`*string*`)`	return expression entered by user after prompt *string*
`sprint(`*expr*`)`	convert symbolic *expr* to string

7.7 Error Handling

Error messages can be incorporated in operator definitions for catching conditions for which language constructs are illegal. The '`error`' function generates an error message and aborts further evaluation. For example, it is undesirable to apply the dot product on scalar arguments the following definitions are included in the '`linalg`' package:

```
_       .* _       := error("dot product of scalars").
[_ | _] .* _       := error("dot product of vector with scalar").
_       .* [_ | _] := error("dot product of scalar with vector").
```

where the '`_`' denote wildcard formal arguments. For example, '`1 .* 2`'. results in an error message:

```
ERROR [eval]:
  dot product of scalars
  signaled on evaluating
  1 .* 2
```

Acknowledgements

The authors would like to acknowledge valuable discussions with Gerard Cats of the Royal Netherlands Meteorological Institute, The Netherlands, and Ilja Heitlager for comments and for providing the REDUCE examples.

Appendix

A The `vecalg` Package Source

```
%       vecalg.s
%
%       Purpose:        Vector algebra package and PDE vector notation
%       Copyright:      R.A. van Engelen, Leiden University, 1997

include linalg.         % import dot- and cross-product operators

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Define grad, div, curl, and lapl
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

prefix grad has latex_name("\nabla").
prefix div  has latex_name("\nabla\cdot").
prefix curl has latex_name("\nabla\times").
prefix lapl has latex_name("\nabla^2").

% Note: Vectors 'coordinates' and 'coefficients' define a coordinate system.
%       Scalar operator 'diff(X, Y)' represents differentiation of X wrt. Y.
%       In def. of 'grad X' vectors 'coordinates' and 'coefficients' are mapped.
```

```
grad X := diff(X, coordinates) / coefficients.

div X            := diff(X, coordinates:1) / coefficients:1.
div [X1, X2]     := ( diff(coefficients:2 * X1, coordinates:1)
                    + diff(coefficients:1 * X2, coordinates:2)
    ) / (coefficients:1 * coefficients:2).
div [X1, X2, X3] := ( diff(coefficients:2 * coefficients:3 * X1, coordinates:1)
                    + diff(coefficients:3 * coefficients:1 * X2, coordinates:2)
                    + diff(coefficients:1 * coefficients:2 * X3, coordinates:3)
    ) / (coefficients:1 * coefficients:2 * coefficients:3).

curl _            := error("Illegal use of curl").      % Scalar argument???
curl [X1, X2, X3] := [ ( diff(coefficients:3 * X3, coordinates:2)
                       - diff(coefficients:2 * X2, coordinates:3)
       ) / (coefficients:2 * coefficients:3)
                     , ( diff(coefficients:1 * X1, coordinates:3)
                       - diff(coefficients:3 * X3, coordinates:1)
       ) / (coefficients:3 * coefficients:1)
                     , ( diff(coefficients:2 * X2, coordinates:1)
                       - diff(coefficients:1 * X1, coordinates:2)
       ) / (coefficients:1 * coefficients:2)
                     ].

lapl X := div grad X.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Vector notation
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Note: 'nabla' represents the gradient symbol. It is used to formulate a PDE
%       in full vector notation with dyadic operators +, *, .*, #*, and ^.

nabla        is inert.
nabla_square is inert.

nabla_plus(Vector)                     is inert.
nabla_times(Factor)                    is inert.
nabla_times_plus(Factor, Vector)       is inert.
nabla_square_plus(Term)                is inert.
nabla_square_times(Factor)             is inert.
nabla_square_times_plus(Factor, Term)  is inert.
nabla_dot(Vector)                      is inert.
nabla_cross(Vector)                    is inert.
nabla_times_dot(Factor)                is inert.
nabla_times_cross(Factor)              is inert.

% Extend definition of addition

nabla                         + [X | Xs]                    := nabla_plus([X | Xs]).
nabla_square                  + X                           := nabla_square_plus(X).
nabla_plus([Y | Ys])          + [X | Xs]                    := nabla_plus([X | Xs] + [Y | Ys]).
nabla_plus([Y | Ys])          + X                           := nabla_plus(X + [Y | Ys]).
nabla_plus(Y)                 + [X | Xs]                    := nabla_plus([X | Xs] + Y).
nabla_plus(Y)                 + X                           := nabla_plus(X + Y).
nabla_square_plus(Y)          + X                           := nabla_square_plus(X + Y).
nabla_times(Y)                + [X | Xs]                    := nabla_times_plus(Y, [X | Xs]).
nabla_times(Y)                + X                           := nabla_times_plus(Y, X).
nabla_square_times(Y)         + X                           := nabla_square_times_plus(Y, X).
nabla_times_plus(Y, [X | Xs]) + [Z | Zs]                    := nabla_times_plus(Y, [X | Xs] + [Z | Zs]).
nabla_times_plus(Y, [X | Xs]) + Z                           := nabla_times_plus(Y, [X | Xs] + Z).
nabla_times_plus(Y, X)        + [Z | Zs]                    := nabla_times_plus(Y, X + [Z | Zs]).
nabla_times_plus(Y, X)        + Z                           := nabla_times_plus(Y, X + Z).
nabla_square_times_plus(Y, X) + Z                           := nabla_square_times_plus(Y, X + Z).
[X | Xs]                      + nabla                       := nabla_plus([X | Xs]).
X                             + nabla_square                := nabla_square_plus(X).
[X | Xs]                      + nabla_plus([Y | Ys])        := nabla_plus([X | Xs] + [Y | Ys]).
X                             + nabla_plus([Y | Ys])        := nabla_plus(X + [Y | Ys]).
[X | Xs]                      + nabla_plus(Y)               := nabla_plus([X | Xs] + Y).
X                             + nabla_plus(Y)               := nabla_plus(X + Y).
X                             + nabla_square_plus(Y)        := nabla_square_plus(X + Y).
[X | Xs]                      + nabla_times(Y)              := nabla_times_plus(Y, [X | Xs]).
X                             + nabla_times(Y)              := nabla_times_plus(Y, X).
X                             + nabla_square_times(Y)       := nabla_square_times_plus(Y, X).
[Z | Zs]                      + nabla_times_plus(Y, [X | Xs]) := nabla_times_plus(Y, [X | Xs] + [Z | Zs]).
Z                             + nabla_times_plus(Y, [X | Xs]) := nabla_times_plus(Y, [X | Xs] + Z).
[Z | Zs]                      + nabla_times_plus(Y, X)      := nabla_times_plus(Y, X + [Z | Zs]).
Z                             + nabla_times_plus(Y, X)      := nabla_times_plus(Y, X + Z).
Z                             + nabla_square_times_plus(Y, X) := nabla_times_plus(Y, X + Z).

% Extend definition of product
```

```
nabla                          * X                     := grad X.
nabla_square                   * X                     := lapl X.
nabla_plus([X | Xs])           * [Y | Ys]              := [X | Xs] * [Y | Ys] + grad [Y | Ys].
nabla_plus([X | Xs])           * Y                     := [X | Xs] * Y + grad Y.
nabla_plus(X)                  * Y                     := X * Y + grad Y.
nabla_times(X)                 * Y                     := X * grad Y.
nabla_times_plus(Y, [X | Xs]) * Z                     := [X | Xs] * Z + Y * grad Z.
nabla_times_plus(Y, X)         * Z                     := X * Z + Y * grad Z.
nabla_square_plus(X)           * Y                     := X * Y + lapl Y.
nabla_square_times(X)          * Y                     := X * lapl Y.
nabla_square_times_plus(Y, X) * Z                     := X * Z + Y * lapl Z.
nabla_dot([X | Xs])            * Y                     := [Y | Xs] .* grad Y.
nabla_cross([X | Xs])          * Y                     := [Y | Xs] #* grad Y.
nabla_times_dot(X)             * Y                     := div(X * grad Y).
nabla_times_cross(X)           * Y                     := div(X * grad Y).
X                              * nabla                 := nabla_times(X).
X                              * nabla_square          := nabla_square_times(X).
X                              * nabla_times(Y)        := nabla_times(X * Y).
X                              * nabla_square_times(Y) := nabla_square_times(X * Y).

% Extend definition of power

nabla ^ 2 := nabla_square.

% Extend definition of dot product

nabla                          .* nabla          := nabla_square.
nabla                          .* nabla_times(X) := nabla_times_dot(X).
nabla                          .* [X | Xs]       := div [X | Xs].
nabla_plus([X | Xs])           .* [Y | Ys]       := [X | Xs] .* [Y | Ys] + div [Y | Ys].
nabla_plus(X)                  .* Y              := X * Y + div Y.
nabla_times(X)                 .* [Y | Ys]       := X * div [Y | Ys].
nabla_times(X)                 .* Y              := X * div Y.
nabla_times_plus(Y, [X | Xs]) .* [Z | Zs]       := [X | Xs] .* [Z | Zs] + Y * div [Z | Zs].
nabla_times_plus(Y, X)         .* Z              := X * Z + Y * div Z.
[X | Xs]                       .* nabla          := nabla_dot([X | Xs]).

% Extend definition of cross product

nabla                 #* nabla_times(X) := nabla_times_cross(X).
nabla                 #* [X | Xs]       := curl [X | Xs].
nabla_plus([X | Xs]) #* [Y | Ys]       := [X | Xs] #* [Y | Ys] + curl [Y | Ys].
nabla_times(X)        #* [Y | Ys]       := X * curl [Y | Ys].
[X | Xs]              #* nabla          := nabla_cross([X | Xs]).

% (Part of) Library of commonly used coordinate systems

coordinate_system("1D Cartesian") :=
  procedure(
    coordinates  := [x];
    coefficients := [1];
    return(nil)
  ).
coordinate_system("2D Cartesian") :=
  procedure(
    coordinates  := [x, y];
    coefficients := [1, 1];
    return(nil)
  ).
coordinate_system("3D Cartesian") :=
  procedure(
    coordinates  := [x, y, z];
    coefficients := [1, 1, 1];
    return(nil)
  ).
coordinate_system("Polar") :=
  procedure(
    coordinates  := [r, theta];
    coefficients := [1, r];
    return(nil)
  ).
coordinate_system(X) := error("Unknown coordinate system " // sprint(X)).
```

B The `linalg` Package Source

```
%       linalg.s
%
%       Purpose:        Matrix/vector operations script
%       Copyright:      R.A. van Engelen, Leiden University, 1997
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Matrix/vector concatenation (put matrix and vector above each other)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

infix // is associative(left).

  [X | VecX]   //   [Y | VecY]   := append([X | VecX], [Y | VecY]).
[ [X | RowX] ] // [ [Y | RowY]
                 | MatY       ] := [ [X | RowX]
                                   , [Y | RowY]
                                   | MatY       ].
[ [X | RowX]
| MatX       ] // [ [Y | RowY]
                 | MatY       ] := [ [X | RowX]
                                   | MatX //
                                     [ [Y | RowY]
                                     | MatY       ] ].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Matrix/vector concatenation (put matrix and vector besides each other)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

infix ++ is associative(left).

[X]        ++ [Y]            := [ [X, Y] ].
[X | ColX] ++ [Y | ColY]     := [ [X, Y] | ColX ++ ColY ].

[X]        ++ [ [Y | RowY] ] := [ [X, Y | RowY] ].
[X | ColX] ++ [ [Y | RowY]
               | MatY       ] := [ [X, Y | RowY]
                                 | ColX ++ MatY  ].

[ [X | RowX] ] ++ [ [Y | RowY] ] := [ [X | RowX] // [Y | RowY] ].
[ [X | RowX]
| MatX       ] ++ [ [Y | RowY]
                 | MatY       ] := [ [X | RowX] // [Y | RowY]
                                   | MatX       ++ MatY       ].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Create zero matrix
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

zero(Rank) := zero(Rank, Rank).
zero(RowRank, ColRank) := fill(RowRank, fill(ColRank, 0)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Create identity matrix
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ident(1) := [[1]].
ident(N) :=
        if (N <= 0)
        then (error("Cannot create identity matrix of rank " // sprint(N)))
        else ([[1 | fill(N - 1, 0)] | zero(N - 1, 1) ++ ident(N - 1)]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Matrix transpose
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

postfix ‘ has latex_name("^T").

[ [X | Row] ]‘ := [X | Row].
[ [X | Row]
| Mat       ]‘ := [X | Row] ++ Mat‘ .
[X]‘           := [ [X] ].
[X | Vec]‘     := [ [X | Vec] ].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Matrix-matrix and vector-vector addition
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
[[_ | _] | _] +  [_ | _]      := error("matrix + vector").
 [_ | _]      + [[_ | _] | _] := error("vector + matrix").
[[_ | _] | _] -  [_ | _]      := error("matrix - vector").
 [_ | _]      - [[_ | _] | _] := error("vector - matrix").

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Dot/inner product
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

infix .* has latex_name("\cdot").

[X]      .* [Y]      := X * Y.
[X | Xs] .* [Y | Ys] := X * Y + Xs .* Ys.

_        .* _        := error("dot product of scalars").
[_ | _] .* _        := error("dot product of vector with scalar").
_        .* [_ | _] := error("dot product of scalar with vector").

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Cross/outer product
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

infix #* has latex_name("\times").

[X]          #* [Y]          := [X * Y].
[X1, X2]     #* [Y1, Y2]     := [X1 * Y2 - X2 * Y1, X2 * Y1 - X1 * Y2].
[X1, X2, X3] #* [Y1, Y2, Y3] := [X2 * Y3 - X3 * Y2, X3 * Y1 - X1 * Y3, X1 * Y2 - X2 * Y1].

_        #* _        := error("cross product of scalars").
[_ | _] #* _        := error("cross product of vector with scalar").
_        #* [_ | _] := error("cross product of scalar with vector").

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Matrix-matrix and matrix-vector product
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

infix &* is associative(left) has latex_name("\,").

 [X | Col]  &*  [Y]        := [X | Col] * Y.
 [X | Col]  &* [[Y | Row]] := matmult_([X | Col], [Y | Row]).
[[X | Row]] &*  [Y | Col]  := [X | Row] .* [Y | Col].

[[X | Row1] | Rows1] &*  [Y | Row2]          := matmult_([[X | Row1] | Rows1], [[Y | Row2]]).
[[X | Row1] | Rows1] &* [[Y | Row2] | Rows2] := matmult_([[X | Row1] | Rows1], [[Y | Row2] | Rows2]‘).

 [_ | _]      &* _             := error("matrix product with scalar").
[[_ | _] | _] &* _             := error("matrix product with scalar").
_             &*  [_ | _]      := error("matrix product with scalar").
_             &* [[_ | _] | _] := error("matrix product with scalar").

% matmult_: matrix product with transpose of matrix

matmult_([[X | Row] | Rows], [[Y | Col]]) :=
        [[X | Row] .* [Y | Col] | matmult_(Rows, [[Y | Col]])].
matmult_([[X | Row] | Rows], [[Y | Col] | Cols]) :=
        [dotprodrows_([X | Row], [[Y | Col] | Cols]) | matmult_(Rows, [[Y | Col] | Cols]) ].
matmult_([], [[_ | _] | _]) := [].
matmult_([[X] | Rows], [Y | Col]) := [X * [Y | Col] | matmult_(Rows, [Y | Col])].
matmult_([X | Rows], [Y | Col]) := [X * [Y | Col] | matmult_(Rows, [Y | Col])].
matmult_([], [_ | _]) := [].
matmult_(_, _) := error("incompatible matrix dimensions").

dotprodrows_([_ | _   ],                   []) := [].
dotprodrows_([X | Row1], [[Y | Row2] | Rows2])
  := [[X | Row1] .* [Y | Row2] | dotprodrows_([X | Row1], Rows2)].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Gaussian elimination
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

gauss arity 1 is intrinsic(module = gauss).     % Prolog predicate gauss/2

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

```
%       Matrix power
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[[X | Row] | Mat] ^ -1 :=
        map(drop(Rank, gauss([[X | Row] | Mat] ++ ident(Rank))))
        where Rank = length([[X | Row] | Mat]).
[[X | Row] | Mat] ^ 0 := ident(length([X | Row])).
[[X | Row] | Mat] ^ 1 := [[X | Row] | Mat].
[[X | Row] | Mat] ^ 2 := [[X | Row] | Mat] &* [[X | Row] | Mat].
[[X | Row] | Mat] ^ N := if (op(0, N) == "T")
                         then ([[X | Row] | Mat]')
                         else (if (N mod 2 == 0)
                               then (([[X | Row] | Mat] ^ (N div 2)) ^ 2)
                               else ([[X | Row] | Mat] &* [[X | Row] | Mat] ^ (N-1))).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%       Solve (MATLAB left division)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

[[X | Row] | Mat] \ [Y | Vec] := [[X | Row] | Mat] ^ -1 &* [Y | Vec].
```

References

[1] $\mathcal{AMS}$-LaTeX, Version 1.1 User's Guide, American Mathematical Society, 1991.

[2] R. Bird and P. Wadler, *Introduction to Functional Programming*, Prentice Hall International Series in Computer Science, C.A.R. Hoare (series ed), Prentice Hall, 1988.

[3] A. Bonadio, *Theorist*, Prescience Corp., 1989, San Francisco, CA.

[4] I. Bratko, *Prolog, Programming for Artifical Intelligence*, 2nd ed., Addison-Wesley Publishing Company, Inc., 1990.

[5] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt, *MAPLE Reference Manual*, Waterloo, 1988.

[6] A.C. Hearn, *REDUCE User's Manual, version 3.6*, RAND, Santa Monica CA, July 1995.

[7] R. van Engelen, L. Wolters, and G. Cats, *Ctadel: A Generator of Multi-Platform High Performance Codes for PDE-based Scientific Applications*, proceedings of the 10^{th} ACM International Conference on Supercomputing, ACM Press, New York, 1996, pp. 86–93.

[8] R. van Engelen, I. Heitlager, L. Wolters, and G. Cats, *Incorporating Application Dependent Information in an Automatic Code Generating Environment*, proceedings of the 11^{th} ACM International Conference on Supercomputing, ACM Press, New York, 1997, pp. 180–187.

[9] R. van Engelen, L. Wolters, and G. Cats, *Tomorrows's Weather Forecast: Automatic Code Generation for Atmospheric Modeling*, in IEEE Journal of Computational Science and Engineering, Vol. 4, No. 3, July/September 1997.

[10] D. Harper, *AVECTOR: A vector algebra and calculus package*, in A.C. Hearn, *REDUCE User's Manual, version 3.6*, RAND, Santa Monica CA, July 1995.

[11] N. Kajler, *CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems*, in proceedings of ISSAC'92, Berkley, USA, July 1992, pp. 376–386.

[12] N. Kajler and N. Soiffer, *A Survey of User Interfaces for Computer Algebra Systems*, to appear in Journal of Symbolic Computation, 1994.

[13] L. Lamport, *LaTeX: A Document Preparation System*, Addison-Wesley, 1986.

[14] R. Liska, M. Yu. Shashkov, and A.V. Solovjov, *Support-operators Method for PDE Discretization: Symbolic Algorithms and Realization*, Mathematics and Computers in Simulation, IMACS, 1993, pp. 173–183.

[15] *MATLAB, High-Performance Numeric Computation and Visualization Software*, User's Guide, The Math Works Inc., 1992.

[16] L. Sterling and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press, 1986.

[17] J. Wielemaker, *SWI-Prolog Reference Manual*, University of Amsterdam, 1995. Available by anonymous ftp from `swi.psy.uva.nl`.

[18] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, Boston, 1988.