Rascal
the Advanced Scientific CALculator

Userguide

Sebastian Ritterbusch

26th February 2001

Contents

1 Introduction

Rascal is based on an extremely modular approach; depending on the user's needs, various powerful modules can be compiled into the system. Thus Rascal can be a light-weight tool as well as a very powerful computational system. This user documentation is created during compilation and describes the included features.

2 Basic Usage

Invoking the program leads to a prompt where you can enter expressions. The expressions or definitions are being evaluated as soon you press return. A semicolon at the end suppresses any output. As an example:

```
>5+2
 7
>5+2;
>
```

Variables are also supported, where the names are case-sensitive alphabetic letters and may be followed by numbers. An assignment is done using the equal sign:

```
>MyVariable2=7;
>MyVariable2*MyVariable2
 49
>
```

Undefined variables are always assumed to be integer zeros.

The user can also define functions of one variable (n-ary functions can be implemented using vectors or matrices as arguments). The names follow the same rules as the names of variables:

```
>MyNiceFunction17(x)=x*x+5*x;
>MyNiceFunction17(2)
 14
>
```

The semicolon at the end of the definition is mandatory and the calling-method is "call by variable", similar to parametric "#define"s in C++ with the difference that changes in the operand-variables are not forwarded to the original variable. Of course functions can be nested and invoked with any data-type. If the data-type does not support a certain operation, an error occurs.

Conditional expressions can be realized using the C-style "?:"-operator:

```
>5==3?2.3:17
 17
>abs(x)=x>0?x:-x;
>abs(-18)
 18
>abs(12)
 12
>
```

If the condition before the questionmark is true, the first expression is returned, else the second. In contrast to C the result-types of the two alternatives here can be different, also both alternatives are computed before the decision takes place, thus recursions are not possible.

You can exit Rascal by entering "quit" followed by a return.

Furthermore the different modules predefine functions, which are documented in the following sections.

3 Generic Data Types

Rascal has a generic subsystem, which supports integers, doubles, strings and matrices.

3.1 Integers and Doubles

Simple numbers are being interpreted as integers, which in this version of Rascal are represented as integers on the underlying computer architecture. No rounding errors will occur, but undetected overflows may occur.

Using double precision floating-point numbers overcomes this problem, but rounding errors may occur. A number is being interpreted as a floating-point number when there is a decimal-point within or at the end of the number and/or an exponent. As an example `1.234e+12` represents $1.234 \cdot 10^{+12}$. Be warned that there is no exact representation for 0.1 and many many other numbers in binary floating-point representations.

All operands like `+,-,*,/,^,%` are defined for integers and doubles, together with standard functions `sin, cos, tan, cot, asin, acos, atan, acot, sinh, cosh, tanh, coth, asinh, acosh, atanh, acoth, log, exp, sqrt`. A postfix `!` computes the factorial of the operand (currently just in integer).

The values can be compared using the operators `==,!=,<,<=,>,>=`, logical expression can be connected using `&,|` and the logical negation `~`.

3.2 Strings

Mostly strings are used internally, as an example all datatypes have a function called `output` which defines how values of that type can be printed on the screen.

Besides strings can be used as "evaluation variable": As all the standard operators and functions are also defined for strings, one can determine the expression Rascal would evaluate if the argument only had been a real value.

Strings are enclosed in quotes:

```
>f(x)=x*x+2*x;
>f(f("y"))
 "y*y+2*(y)*y*y+2*(y)+2*(y*y+2*(y))"
>
```

But the operators `==, !=` are still used to compare two strings.

3.3 Matrices and Vectors

In Rascal vectors are just matrices with either just one row or one column. You can enter a matrix by using brackets, where values within a line are separated by spaces, lines are separated by semicolons. matrices of same size can be added and subtracted using the usual operators:

```
>A=[1 2;3 4;5 6]
 [1 2;3 4;5 6]
>B=[-3 -4;2 7;2 9]
 [-3 -4;2 7;2 9]
>A+B
 [-2 -2;5 11;7 15]
>A-B
 [4 6;1 -3;3 -3]
>
```

Scalars can be multiplied to the matrices and two matrices can be multiplied if the number of columns of the first matches the number of rows of the second. Dividing by a matrix means multiplying with the inverse, of course this is only defined for quadratic matrices.

Types using the matrix inversion must be able to be compared to integers. If there is no inverse, an empty matrix is being returned.

```
>[1 2]*[3;4]
 [11]
>[1;2]*[3 4]
 [3 4;6 8]
>A=[1 3;4 13];
>1/A
 [13 -3;-4 1]
>[2 0;0 3]/A
 [26 -6;-12 3]
>
```

The cells of matrices can be of any type; here an example for an integer, double, matrix, string matrix:

```
>A=[1 2.34;[1 2;3 4] "hu"];
>
```

Now A looks like the following:

$$A = \begin{pmatrix} 1 & 2.34 \\ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & "hu" \end{pmatrix}$$

The summation of matrices and scalars is defined as the addition between the matrix and the identity of same size times the scalar. vector valued functions can be defined easily, and cells of vectors can be accessed to define n-ary functions:

```
>f(x)=[x*x;x+2];
>f(8)
 [64;10]
>f([1 2;3 4])
 [[7 10;15 22];[3 2;3 6]]
>f[1 2;3 4]
 [[7 10;15 22];[3 2;3 6]]
>g(x)=x(1)+x(2)*x(3);
>g[12 -5 2]
 2
>
```

4 Integer Fractions

With this module, Rascal supports integer fractions, which is an alternative to using floating-point numbers:

```
>1/3-(2/7)/(3/8)
 -3/7
>A=[1 2;3 4];
>1/A
 [-2 1;3/2 -1/2]
>1.0/A
 [-2 1;1.5 -0.5]
>
```

As you can see any operation with a double will round the fraction to the next floating-point number. The fractions are normalized in each step, only the numerator can be negative.

The standard-operators and functions should be defined for fractions, thus the user shouldn't feel any difference between fractions and doubles, but that the latter is less accurate.

Still the abilities of the fractions are limited to the abilities of the underlying Integer, the following is likely to happen on 32 bit-CPUs:

```
>1/65536+1/65535
 -131071/65536
>
```

Currently fractions use *long long int* internally, but use the usual *int* output routines.

5 Complex Arithmetics

This module introduces complex arithmetics to Rascal. Complex numbers can be created by using the "complex"-function and it is often handy to define the purely imaginary constant i.

```
>i=complex(0,1);
>(5+3*i)*(2+i)
 complex(7,11)
>
```

The real and imaginary part of a complex number can be accessed like the cells of vectors: The first cell is the real, the second the imaginary value. Additionally like matrices complex values can consist of all different types of data. The "transpose" of a complex value is the complex conjugate.

```
>f(x)=x*x(1)+x(2);
>f(complex("x",4))
 complex("x*x+4","4*(x)")
>(2+3*complex(0,1))'
 complex(2,-3)
```

The following standard functions are defined for complex: `exp, log, pow, sqrt, sin, cos, tan, cot, asin, acos, atan, acot, sinh, cosh, tanh, coth, asinh, acoth, atanh, acoth`.

The `log` represents the main value of the natural logarithm, the `pow(a,b)` or `a^b` is evaluated as $a^b = e^{b \ln a}$, the `sqrt(a)` as $a^{\frac{1}{2}}$ and the rest is determined based on these functions and the pendants on the real axis.

One could also define a complex out of matrices, but this is not advisable as this easily gets confusing and Rascal prefers complex values within matrices.

6 Taylor Arithmetics

This concept offers the opportunity to accurately compute derivatives of functions. The relationship between the resulting vector and the derivative at the evaluation point is the following:

$$f(\text{taylor}[x\ 1\ 0\ 0\ldots]) = [f(x)\ \frac{f'(x)}{1!}\ \frac{f''(x)}{2!}\ \ldots]$$

As an example $f(x) = \frac{x+2}{x-1}$ with $f'(x) = -\frac{3}{(x-1)^2}$:

```
>f(x)=(x+2)/(x-1);
>f(taylor[2 1])
 taylor[4 -3]
>X(u)=taylor[u 1];
>f(X(0))
 taylor[-2 -3]
>
```

Together with the string arithmetic this module can also be used to determine formulas for the derivatives:

```
>f(x)=(x+2)/(x-1);
>X(u)=taylor[u 1];
>f(X("z"))
 taylor["(z+2)/(z-1)"
"(1-(z+2)/(z-1))/(z-1)"]
>
```

Currently only the basic operations `+,-,*,/` are supported and the cells can be accessed the same way like the cells of vectors.

7 Questions?

Send questions, ideas, hints and congratulations to *rascal@ritterbusch.de*.

8 Licence