Chapter 4

Semantics of M

MORE important than the syntax of M— as introduced in the previous chapter — is the *meaning* of the M-statements and expressions. What does the system do, when it sees this or that?

4.1 Expression

Example 1 Input:

```
a; a + 1; a + b * (a-4); f(a,b);
```

Example 1: Almost everything you can build using `+`, `-`, `*`, `/`, `(` and `)` is an expression, as long as the parentheses match and as long the operators are accompanied by two arguments. The notation for a function call is a bit different to this rule but not uncommon.

The meaning of expressions is straightforward: the values of the arguments are added, subtracted, multiplied or divided, according to standard mathematical conventions. Again, as usual, functions are called by the function name followed by a comma separated list containing the arguments. Besides numbers, names and function calls there are other syntactic constructs that can be used as expressions amongst which are *ureals* (see 4.1.2), *attribute lists* (see 4.2) or *measurements* (see 4.4).

4.1.1 Equation

Practically any expression can be assigned to a name.

Example 2 Input:

```
a = 1; b = a; c = < 1 :g 3% >;
```

Example 2: After having said `a = 1;` you cannot ever change the content of `a` again. And whenever you say `a` it will be interpreted as if you had said `1`.

These assignments are really *equations*. That means that it doesn't matter whether the equation defining `a` is written at the beginning or at the end of your M-code. The only requirement on the order of such equations is that all of the defining equations must be known prior to the *evaluation* of an expression. In the above example you cannot evaluate `b` until the equation defining `a` is known to the system. Avoid circular definitions, such as in Example 3.

Example 3 Input:

```
a = b; b = c; c = a;
```

Example 3: This would eventually lead to an infinite loop. The system will detect a circular definition and issue an error message.

Internally there is a table which holds an entry for each name; so to say a big directory of names where each name's content can be looked up. Beside the syntax rule `eqn` there is also a rule `asgmnt`. The difference between the two of them is that the former allows to define a symbol `exactly` once, whereas the latter allows to overwrite the definition of a symbol with another content. Its use, however, is deprecated.

4.1.2 Leaf

As seen above, an expression can be constructed of names or of numbers. Yet another basic element for expressions is a so called or it can be a so called *ureal*. A syntactically correct `ureal` is embraced by < and > containing three items: the *mean*, the *distribution* and the second parameter for the distribution. In most cases the second parameter is the *standard deviation* of

the random variable. The mean and the second parameter may be given as a number or as a name, whereas the distribution is encoded by some mnemonics. Table 5 lists the different distributions and their interpretation of the second argument.

Table 5 Distribution Mnemonics for M

mnemonic	Meaning and Interpretation of the second parameter
`:g` or `:␣`	Gaussian or Normal distribution. The second argument is taken as standard deviation.
`:l`	Log-normal distribution. The second argument is taken as standard deviation. Note, that the first argument is taken as the mean which is not coinciding with the most probable value (see Figure 1 on page 14).
`:t`	Triangular distribution. The second argument b of the leaf `<a :t b>` denotes the support of the triangular distribution function, i.e. observed values are within $[a-b, a+b]$. The standard deviation then is $b/\sqrt{6}$.
`:r`	Rectangular distribution. The second argument b of the leaf `<a :r b>` denotes the support of the rectangular distribution function, i.e. observed values are within $[a-b, a+b]$. The standard deviation then is $b/\sqrt{3}$.

Both ureals and numbers can have a *unit* attached. A ureal has even up to 3 syntactical positions to attach a unit. The mean, the second parameter and the entire ureal triplet can take an associated unit. If more then one unit is given to a ureal, the system tries to deduce the final unit from the combination of given units. It issues an error message, if some units mismatch. A few examples will clarify the numerous possibilities:

Example 4 Input:

```
<3.5 : 0.05> [ml]
```

Example 4: This describes an entity of a normal distributed random variable of 3.5 ml with a standard deviation of 0.05 ml.

Example 5 Input:

```
<3.5[m] :r 5[mm]>
```

Example 5: This denotes a 3.5 m sample with rectangular distribution in the range of $[3.495\,\mathrm{m} \ldots 3.505\,\mathrm{m}]$.

Example 6 Input:

```
<4.5 : 0.6 [V]> [ml]
```

Example 6: This will issue an error message, because the unit of this entity cannot be deduced, since V and ml are two just too different types of units.

Instead of a unit the second argument can have a `%`-sign. This will be translated into the respective percentage of the first parameter.

Example 7 Input:

```
<4 :t 5%>
```

Example 7: A random variable with mean 4 and with triangular distribution. The spread is ±5%, i.e. `<4 :t 0.2>`

4.1.3 Units

The system is aware of units. That is, it knows a lot of *SI accepted* units and some non-decimal ones (e.g. miles or yards and so forth). It also knows the engineering prefixes as m (milli) for $1/1000$. Table 6 enumerates all *known units* and *engineering prefixes*. Units are always following a number. Section 3.2.4 shows some instances of correct units. Note a peculiarity with temperatures: Is 1K equal to 1°C? The answer is sometimes "Yes" and sometimes "No". In multiplicative context, i.e. when *temperature differences* are the issue, then 1K is indeed equal to 1°C. Only when *absolute temperatures* are meant then $0°\mathrm{C} = 273.15\,\mathrm{K}$. To distinguish these cases, the system requires in the latter case `[°Cabs]` instead of merely `[°C]`. It will assume that $1\,°\mathrm{C} = 1\,\mathrm{K}$ by default. Examples 19 and 20 on page 112 show the use of absolute and relative temperatures.

Table 6 List of Known and Accepted Units and Engineering Prefixes

Time

s	second
min	minute
h	hour
d	day
y	year

Distance

m	meter
ft	feet
in	inch
yd	yard
mi	mile

Mass

g	gramme
t	tonne

Current

A	Ampère

Temperature

K	Kelvin
°C	Centigrade

Luminosity

cd	Candela

Quantity

mol	Mol

Acc. Units

Hz	Hertz
l	liter
Pa	Pascal
bar	10^5 Pa
J	Joule
W	Watt
N	Newton
C	Coulomb
V	Volt
F	Farad
G	Gauss
H	Henry
T	Tesla

Eng. prefixes

a	atto	10^{-18}
f	femto	10^{-15}
p	pico	10^{-12}
n	nano	10^{-9}
mu	mikro	10^{-6}
m	milli	10^{-3}
d	dezi	10^{-1}
da	deka	10^{1}
h	hekto	10^{2}
k	kilo	10^{3}
M	mega	10^{6}
G	giga	10^{9}
T	tera	10^{12}
P	peta	10^{15}
E	exa	10^{18}

4.2 Attribute List

To be able to describe measurement methods we need some notion of "collection of different attributes". Consider a particular substance. Various properties may be of importance in different contexts. Yet they belong to the same substance. An *attribute list* collects a set of equations, and thus is the container for collecting a set of properties.

Example 8 Input:

```
water1 = "H2O"(alpha = 0.001 [1/K], purity = 98%);
water2 = Water(alpha = 0.001 [1/K], purity = 95%,
               molar = 2*<1.00794 : 0.00007>[g/mol]
                       + <15.9994 : 0.0003> [g/mol]);
get(alpha, water1);
get(molar, water2);
get(name, water1);
```

Example 8: This example shows two attribute lists describing a substance using almost the same attribute and value pairs. The name of the attribute list is placed just in front of the first parenthesis. It is either an arbitrary string within double quotes or a name starting with a capital letter. This is in order to distinguish an attribute list and a function call. The attribute list's name is also an attribute.

The predefined function `get` is used for accessing an attribute and its value. It takes two arguments, the attribute name and the attribute list where the attribute should be looked up.

The first call to `get` yields "1 1/kK", i.e. one per thousand degree Kelvin, the second returns the expression for the molar mass as defined in the attribute list. The third call evaluates `name` which resolves to `H2O` which cannot be evaluated any further and therefore issues an error. The attribute `name` is thus only for debugging purposes in order to inspect an attribute list by its name.

Example 8 Output:

```
1 [1/kK];
2 * <1.00794 : 7e-005>[g/mol] + <15.9994 : 0.0003>[g/mol];
CEval: <#207#> H2O is undefined!
1 Error:
CEval: <#209#> <#207#> H2O is undefined! by get()
```

4.3 Functions

Functions may be defined in two ways as shown in Example 9. Formal parameters like `a` and `b` and local variables like `c` are scoped. That means that they are visible within the function body but not from outside of the function body.

Example 9 Input:

```
function func1(a,b) {
        c = (a - 1) * (b + 1);
        c*c;
};

func2 = function(a,b) {
        c = (a - 1) * (b + 1);
        c*c;
};

a = <2 : 0.5%>;
func1(a,7);
```

Example 9: The function definition for `func1` and for `func2` are exactly equivalent as can be seen in the output.

The return value of a function call is the last expression of the function body. A defined function is called by its function name *and* by passing the formal parameters within parenthesis.

In this simple example the function call results already in a lengthy expression.

Example 9 Output:

```
func1 = dfunction(a,b) {
   c = (a - 1) * (b + 1);
   c * c;
   };
func2 = dfunction(a,b) {
   c = (a - 1) * (b + 1);
   c * c;
```

```
   };
a = <2 : 0.01>;
((<2 : 0.01> - 1) * (7 + 1)) * ((<2 : 0.01> - 1) * (7 + 1));
```

Functions can be a member of an attribute list like any other entity too. Therefore one can define a function within an attribute list as well as in the global scope. If a function is taken out of its surrounding attribute list by `get` the question is: where are nonlocal variables looked up? Either they are looked up in the current environment at evaluation time, i.e. they are *dynamically bound* or they are looked up in the scope valid at definition time, i.e. *statically bound*. The keyword `function` intelligently defines a function using in almost all cases the intended variable binding. Examples 10 and 11 show each a situation where the keyword `function` is using the static and the dynamic variable binding. The keyword `dfunction` is used to explicitly define a function fetching nonlocal variables from the current environment at evaluation time, and the keyword `sfunction` is defining a function searching for nonlocal variables in the defining scope. The keyword *function* uses automatically static binding (i.e. is read as `sfunction`) when it appears within an attribute list. It is defining a dynamically bound function (i.e. is read as `dfunction`) everywhere else. However, you can always explicitly control the function type using `dfunction` or `sfunction` instead of `function`.

Example 10 Input:

```
a = 1;
list = List(a = 22,
            function sfun(p){p + a;});
fun = get(sfun,list);

eval(fun);
eval(fun(3));
```

Example 10: The function call to `fun` evaluates ultimatively the function `sfun` within the attribute list `List`. Now realize the problem about the nonlocal variable `a` within the body of `sfun`: at defining time the function was defined in a scope where an `a` was defined to be `22`, but at calling time of `fun` and therefore of `sfun` the function is evaluated within the global scope where an `a` is defined to be `1`.

Example 10 Output:

```
a = 1;
list = "List"(a = 22,
              sfun = dfunction(p) {p + a;});
fun = get(sfun, list);
sfunction(p) {p + a;};
3 + 22;
```

The call `eval(fun)` returns `sfunction(a) {p + a;};` indicating that nonlocal variables (such as `a`) are looked up in the surrounding scope at definition time and therefore the function call `fun(3)` is looking up `a` in the attribute list `List` and returning `3 + 22`. Functions being defined within an attribute list are per default being evaluated within the defining environment.

Example 11 Input:

```
a = 1;
b = 2;
fun = function(a) {b + a;};

eval(fun(3));
eval(fun);
```

Example 11: The call to `eval(fun)` returns the value of `fun` as `dfunction(a) {b + a;};` The keyword `dfunction` indicates that nonlocal variables in the function body (such as `b`) are looked up in the *current* scope. The evaluation of `fun(3)` therefore returns `2 + 3;`. The formal parameter `a` clashes in this case with a variable name in the global scope. Within the function body, the global variable `a` is not accessible therefore.

Example 11 Output:

```
a = 1;
b = 2;
fun = dfunction(a) {b + a;};
2 + 3;
dfunction(a) {b + a;};
```

4.3.1 Predefined Functions

There are a number of predefined functions. They can be classified into *mathematical*, *manipulating* and *interface* functions.

Mathematical Functions

Mathematical functions can be called like ordinary functions. The only difference is that the body is implemented within the system.

`exp(x)` The function evaluates the exponential $y = \exp(x)$.

`log(x)` The function evaluates the natural logarithm $y = \log(x)$.

`pow(b,e)` The function evaluates $y = b^e$.

`diff(f,x)` The function evaluates $y = \frac{\partial f}{\partial x}$, the derivative of the expression `f` with respect to `x`. Note that `x` must be a variable rather than an expression. Otherwise the derivation process would have to recognize all subexpressions of `f` that are equal to the expression `x`. This is a *quite* complex task, in fact this is one of the tasks that justify the use of a symbolical computation tool such as MAPLE or MATHEMATICA. Apart from being very complicated, it is not needed to compute derivatives of expressions with respect to an arbitrary expression. All that is needed in the context of uncertainty evaluation or sensitivity analysis are derivatives with respect to plain input data, in other words, with respect to leafs in the expression tree. The process of differentiation is explained in more details in Section 5.3.3 on page 121.

`mean(x)` The function returns the mean, i.e. $\mu = \mathrm{E}[X]$. Note that this is not strictly a mathematical function: it has no derivative defined. Therefore, a measurement method using this function cannot be evaluated by the `calciso` method (see 4.3.1) which is computing the uncertainty using derivatives.

uncertainty(x) The function returns the uncertainty $\sigma = \sqrt{\mathrm{Var}(X)}$. Note again that this is not strictly a mathematical function. A measurement method should not be described using this function if it must be evaluated by the `calciso` method (see 4.3.1) which is using derivatives.

cov(x,y) and cor(x,y) The expressions `cov(x,y)` or `cor(x,y)` can be used in two ways. First, as a normal function call. In this case they evaluate either the covariance or the correlation between their two arguments according to Equation (2.18) and (2.6). Second, both expressions can stand on the left hand side of an assignment as in `cov(x,y) = 0.4` or `cor(x,y) = 0.03`. In this case they define either the covariance or the correlation between two random variables to be 0.4 or 0.03 respectively. The system can check the feasibility of the assigned values only very late. Exactly speaking, it can check the feasibility only then the arguments' `x` and `y` variance can be evaluated. Furthermore the system evaluates only those expressions that are needed for the result. So, you can define infeasible covariances for 'unused' variables and neither you nor the system will realize it. On the other side, the system will issue a warning if it detects infeasible covariance or correlation entries during evaluation of an expression. That is the correlation ρ, computed according to Equation (2.6), must lay between $-1 \leq \rho \leq 1$.

These two function names, `cov` and `cor`, are the only ones that can be placed on either side of a "=" sign. On the right hand side they are evaluated like ordinary functions. On the left hand side they are treated like any other name: they are assigned the expression of the right hand side.

Manipulating functions

The following manipulating functions are left in the system for two reasons. First and above all, they are needed for the evaluation of measurement descriptions and during the subsequent uncertainty evaluation. Second, it is interesting and instructive to see their results at least for debugging if nothing else.

get The expression `get(a,b)` assumes and checks that `a` evaluates to an attribute list and that `b` is a name of an attribute in that attribute list `a`. After all these checks are passed, it effectively returns the value of attribute `b`. Examples using `get` are Example 8, 10 and 14.

tree The expression `tree(a)` prints the expression tree of `a` into the default output stream which is normally connected to a logging file. The function `tree` serves thus mainly for understanding and debugging the calculator by printing out internal data of a concrete expression tree.

Example 12 Input:

```
vecelem = <5 : 5%>;
a = ( 1, 3, vecelem );
b = ( 2, a, 4 );
ve = get(2,get(1,b));
tree(vecelem, ve);
```

Example 12: The output of this example into the logging file is as follows.

Example 12 Output:

```
vecelem = <5 : 0.25>;
a = (1, 3, <5 : 0.25>);
b = (2, (1, 3, <5 : 0.25>), 4);
ve = get(2, get(1, b));
#2"v0002" #2"v0002"
```

The string `#2"v0002" #2"v0002"` means two things: first, that the internal node `#2` is representing the content of both `vecelem` and `ve`; second, that the internal name of node `#2` is `v0002`. Both the node numbers and the internal name are only of some importance for debugging purposes. It is irrelevant for the user of the MUSAC-system.

eval The expression `eval(a)` expands `a` into its equivalent expression consisting only of operators and leafs. In fact, it combines all available information about its argument into one big data structure (c.f. Figure 12 on page 118). The result of `eval(a)` is an expression tree on which any subsequent analyses are performed.

Example 13 Input:

```
a = 1;
b = 2;
fun = function(a) {b + a;};
```

```
c = fun(3);
eval(c);
eval(fun);
```

Example 13: The result of the first call to `eval` is `2 + 3` which nicely shows the scope of the variables: within the function body `a` represents the formal parameter, and `b` is taken to be the global one.

The second call of `eval` returns the value of `fun` which is a function value being printed out in the logging output as `dfunction(a) {b + a;};` The keyword `dfunction` indicates that variables in the function body (such as `b`) are looked up in the *current* scope.

Example 13 Output:

```
a = 1;
b = 2;
fun = function(a) {b + a;};
c = fun(3);
eval(c);
eval(fun);
```

`diff` The expression `diff(a,b)` expands `a` into its tree and returns an expression tree representing the derivative of `a` with respect to `b`, see also page 78.

`fit` The function `fit` is the M-interface into the regression machinery described in Section 2.3. The function call `fit(xvals, yvals, type=2)` prepares for the computation of a regression of "type 2" (see Table 7 for the currently available types in the MUSAC-system) between the x-data in `xvals` and the y-data in `yvals`. It returns an attribute list containing among other entries two function definitions: The functions are named `fun` and `inv` the former being the *regression* function the latter being the *measurement* function.

Example 14 Input:

```
xvals = (<1 : 1%>, <2 : 1%>, <3 : 1%>);
yvals = (<2 : 1%>, <3 : 1%>, <4 : 1%>);
```

```
calib = fit(type = 1, xvals, yvals);

f = get(fun,calib);
g = get(inv,calib);

a = <1.5 : 1%>;

y = g(a);
x = f(a);

eval(calib);
iso(x,y);
iso(cor(get(iso_p0,calib), get(iso_p1,calib)));
```

Example 14: In this example, the data to be fitted are three points whose data are stored in `xvals` and in `yvals`. The builtin function `fit` is called with this data to return an attribute list named `Regression` into the variable `calib`. This calibration object contains two function definitions: one for the regression function `fun` and one for its inverse `inv`. They are fetched with `get` and assigned to the function names `f` and `g`. Then, the two functions are called with an arbitrary argument `a` and their results are stored in `x` and `y`. The actual regression, i.e. the finding of the parameters `p0` and `p1` for the straight line, is performed during the first function call using them. In this example this is done by the ISO-evaluator (c.f. Section 4.3.1). Once they are computed, they are stored within the attribute list for later reuse. The last call to `iso` evaluates the correlation between the two parameters.

Table 7 Regression Functions in MUSAC

regression type	fitted function	
`type = 0`	$f(x) = p_0 x$	0-linear
`type = 1`	$f(x) = p_0 + p_1 x$	linear
`type = 2`	$f(x) = p_0 + p_1 x + p_2 x^2$	quadratic
`type = 3`	$f(x) = p_0 + p_1 x + p_2 x^2 + p_3 x^3$	cubic
`type = 4`	$f(x) = p_0 + p_1 x^{1+p_2}$	sub- or hyper-linear
`type = 5`	$f(x) = p_0 + p_1 * \exp(p_2 x)$	exponential

Example 14 Output:

```
xvals = (<1 : 0.01>, <2 : 0.02>, <3 : 0.03>);
yvals = (<2 : 0.02>, <3 : 0.03>, <4 : 0.04>);
calibration = fit(xvals, yvals, type=1 ... );
f = get(fun, calibration);
g = get(inv, calibration);
a = <1.5 : 0.015>;
y = g(a);
x = f(a);
"Regression"(fun=sfunction(x) {p0 + p1 * x;},
             inv=sfunction(x) {(x - p0) / p1;},
        iso_p0=<1 : 0.041467>, iso_p1=<1 : 0.0250609>,
        parnum=2, scalex=3, scaley=4, type=1,
        xvals=(<1 : 0.01>, <2 : 0.02>, <3 : 0.03>),
        yvals=(<2 : 0.02>, <3 : 0.03>, <4 : 0.04>));
(<2.5 : 0.0232497>, <0.5 : 0.0340955>);
-0.903603;
```

Interface functions

Last but not least there are the interface functions. Those are the entry points for the effective evaluation of the measurement uncertainty of an expression. They take an argument, expand it by using `eval`, do several plausibility checks, and finally perform their specific uncertainty evaluation.

`calciso` or `iso` This is the function performing the evaluation of the measurement uncertainty strictly according to the GUM [2] as explained in section 2.2.3. The entire process for a GUM evaluation can be listed as follows:

1. Collect all leafs of y that have an uncertainty attributed. Those are the influences x_i on which y is dependent.
2. Compute the derivatives of y with respect to each influence x_i, evaluate them at x and store them in a vector $\boldsymbol{d}$.
3. Extract, compute and evaluate the covariances between each influence, and store them in the covariance matrix $\boldsymbol{C}$.
4. Finally compute $\sigma^2 = \boldsymbol{d}^\top \boldsymbol{C}\boldsymbol{d}$ and return σ, the uncertainty of y.

Suppose that Y is the expression to be evaluated. Then Algorithm 7 is executed in order to compute the uncertainty of Y according to Equation (2.16) on page 24.

Algorithm 7 Evaluation of Y According to GUM and Equation (2.16)

function $\langle m_y, u_y \rangle = \mathsf{Iso}(Y, \boldsymbol{C}^G)$
— Y: expression to be evaluated, $\boldsymbol{C}^G$: global covariance table —
— m_y first order approximation of $\mathrm{E}[Y]$ and
u_y^2 first order approximation of $\mathrm{Var}(Y)$ —

$I \leftarrow \{X \mid \mathrm{Var}(X) \neq 0\}$ — *collect all influences of* Y —

foreach $X_i \in I$ **do** — *compute derivative and evaluate it at the mean* —
$\boldsymbol{d}_i \leftarrow \left.\frac{\partial Y}{\partial X_i}\right|_{\mathrm{E}[X_i]}$

foreach $(i,j) \in [1,|I|] \times [1,|I|]$ **do** — *compute covariances according to Equation* (2.18) *for each combination of influences* —
$\boldsymbol{C}^L_{(i,j)} \leftarrow \mathsf{Covariance}(\boldsymbol{C}^G, X_i, X_j)$

$u_y^2 \leftarrow \boldsymbol{d}^\top \boldsymbol{C}^L \boldsymbol{d}$ — *compute the variance of* Y —

Note the use of the global and local covariance matrices $\boldsymbol{C}^G$ and $\boldsymbol{C}^L$. The global covariance matrix contains globally defined covariances, and it works as a cache for implicit covariances computed by function Covariance which looks for the needed covariance in $\boldsymbol{C}^G$. If it is not there then it computes the implicit covariance by Equation (2.18) and stores it in $\boldsymbol{C}^G$. Thus the global covariance $\boldsymbol{C}^G$ is effectively a caching table for the evaluations of Equation (2.18).

The local covariance $\boldsymbol{C}^L$ just stores the results of Equation (2.18) in the correct order such that they match the evaluated derivatives stored in vector $\boldsymbol{d}$. Then the computation of the variance of Y is a simple evaluation of the quadratic function $\boldsymbol{d}^\top \boldsymbol{C}^L \boldsymbol{d}$.

Note further, that both functions `uncertainty` and `mean` are not allowed within the expression of `y`, since they both are not differentiable operations.

ureal The function call `ureal(y)` evaluates the expression `y` by applying the simple rules stated in Section 2.2.4, ignoring any explicit or implicitly arising correlations.

mc The function `mc(y)` is the interface for calling a *Monte Carlo simulator*. The Monte Carlo simulation is explained in more details in Section 2.2.5 and 5.3.3. A Monte Carlo simulation needs several other parameters to be set. Those parameters are set by attributes in the function calls argument list. The parameters for a simulation are

size The size of each sample simulating a random variable, default is 10000.

seed The random seed to start the random generator, default is none, i.e. each simulation starts with a new random seed taken from the current time.

file A filename where the resulting samples are saved, default is none, i.e. samples are not stored in any file.

4.4 Measurement

Let's reason once again about what a measurement is. There is no concept for something like a *true value*. In fact the true value is never known. We'd have to count atoms, to get at it...When someone is performing a measurement, he or she is trying to get a good estimate for the true value. A measurement is thus one single sample out of a random variable around an expected value. The distribution of that sample is dependent on various influences, such as properties of the substance under consideration, the measurement tool employed and environmental conditions. The art of measuring is twofold: first one has to ensure that the expected value of the sample is close to the true value, and second one has to ensure that the distribution of the sample is as narrow as possible.

The language M provides some means to describe such aforementioned influences. The syntactic construct to combine substance under consideration, measurement tool used, environmental conditions and the expected value is called a *measurement*. It consists of exactly these four parts.

Example 15 Input:

```
vol = 10 [ml]                # expected value
        "H2O"(pure=99%)      # substance
        TitroMat( .... )     # instrument
        (<20 :r 3>[°Cabs]); # environment
```

Example 15: This example shows that the two items `substance` and `instrument` are attribute lists. The item `environment` is also an attribute list with the name "Environment". But if — as in this example — the environmental condition consist only of the temperature, then the `Environment(temperature = ...)` can be abbreviated as shown. If a measurement doesn't depend on environmental conditions at all, then the fourth item can be left out.

The attribute lists become quite lengthy statements very quickly and hence hard to read and maintain. Therefore we suggest to define substance and instrument and maybe even the environment in separate statements as in the Example 16.

Example 16 Input:

```
vol_subst = "H2O"(pure=99%);
vol_tool = TitroMat( .... );
vol = 10 [ml] vol_subst vol_tool (<20 :r 3>[°Cabs]);
```

Example 16: For readability it is worthwhile to split the definitions for instrument, substance and maybe environment too involved in a measurement.

Meaning The value of the measurement is found as follows: The attribute list for the instrument must contain a function named `do()` taking no argument. This function is called and its result is the value of this measurement. That's almost all, that there is.

The `do()` function is a bit special in that it has some predefined local variables: `value`, `subst` and `env`. If the `do()` function wants to access any attribute of the substance under consideration, it can get it using `get(subst,...)` similarly it can fetch any attribute of the environment. The variable `value` is always equal to the expected value of the underlying measurement and can be used within `do()` like any other variable. Because the `do()` function is defined within the scope of the instrument each of the instrument's attribute is visible within the `do()` function body.

Example 17 Input:

```
# library function for applying the correction between
# calibration and measurement temperature
#
function caltemp_correction(caltemp) {
   # caltemp: calibration temperature [unit]
   # curr_temp: environmental temperature [unit]
   1 - (get(temperature,env) - caltemp)*get(alpha,subst);
};

# definition of items involved in the measurement
#
vol_subst = H2O(alpha = 2.1e-4 [1/K]);

vol_tool = Flask(
   tolerance = <0 :r 0.03> [ml],
   calibration_temp = 20[°Cabs],
   function do() {
      caltemp_correction(calibration_temp) * value
         + tolerance;
   }
);

# definition of the measurement
#
vol = 10 [ml] vol_subst vol_tool (<19 :r 3> [°Cabs]);

# evaluations of the measurement
eval(vol);
iso(vol);
```

Example 17: This is a simple example of a volume measurement. The function `caltemp_correction` is part of a *function library* storing functions for single influences and/or separate measurement steps.

In the second section of this example two attribute lists are defined. The first representing the substance being measured and the other describing the instru-

ment executing the measurement. In this particular case the substance is described by its name `H2O` and by only one additional attribute `alpha` quantifying the thermal expansion coefficient of the substance which is 2.1×10^{-4} units per degree Kelvin. The second attribute list represents the instrument. It consists of the mandatory function definition for `do()`, and arbitrary additional attributes. The body of function `do()` contains the mathematical models describing any measurement being executed with this instrument. In this particular case every volume measurement executed with that `Flask` description computes a correction for temperature differences between calibration temperature and environmental temperature and adds a constant amount of uncertainty to the measurement value, representing the tolerance of the flask.

Note how the library function `caltemp_correction` depends on the name of the substance's attribute `alpha` describing the thermal expansion coefficient. It is mere convention how the attributes (i.e. the influences) are named. Therefore it is up to the user or the application using the MUSAC-calculator to ensure name consistency between library functions and concrete entity instances.

In the last two sections, the measurement `vol` is described to be executed in an environment of 19 °C, and finally this volume measurement is evaluated first by `eval` for instructional purposes in this example and finally by `iso` to yield the result of the described volume measurement according to the ISO method. The output is as follows:

Example 17 Output:

```
caltemp_correction = dfunction(caltemp) {
   1 - (get(temperature, env) - caltemp) * get(alpha, subst);
};
vol_subst = "H2O"(alpha = 210 [1/MK]);
vol_tool = "Flask"(tolerance = <0 :r 3e-005>[l],
                   calibration_temp = 293.16 [K],
                    do = dfunction() {
                      caltemp_correction(calibration_temp) *
                        value + tolerance;
                   });
vol = 10 [ml] vol_subst vol_tool "Environment"(temperature =
                                              <292.16 :r 3>[K]);
(1 - (<292.16 :r 3>[K] - 293.16 [K])* 210 [1/MK]) * 10 [ml] +
<0 :r 3e-005>[l];
<10.0021 : 0.0176983>[ml];
```

The result of `eval` is printed on the penultimate row. It shows the expression witch is evaluated by the `iso`. This result is printed on the last line of

the output `iso` evaluation reports `<10.0021 : 0.0176983>[ml];`, i.e. the measurement result is 10.0021 ml with a combined uncertainty of 0.0177 ml.

Example 18 Input:

```
x1 = x2 / <2 : 2%>;
x2 = x3 / <2 : 2%>;
x3 = <4 : 2%>;
x4 = <5 : 2%>;
x5 = <10: 2%>;
x6 = <20: 2%>;

xvals = (x1, x2, x3, x4, x5, x6);
yvals = (<0.332295 : 0.0165>, <0.393916 : 0.018>,
         <0.426571 : 0.021>,  <0.470667 : 0.0225>,
         <0.552436 : 0.03>,  <0.888801 : 0.045>);

any_sub = "Arbitrary_substance"();

cali_tool = "A_tool"(calibration = fit(type = 1,
                                       xvals, yvals),
                     reading_tol = 5%,
                     do = function() {
                           g = get(inv, calibration);
                           a = value * reading_tol;
                           g(<value : a>);
                     });

a = 0.5 any_sub cali_tool ;
iso(a);
```

Example 18: This example shows a calibration. Incidentally, it is the same calibration that is described in Section 2.3.6. In the first block there are 3 calibration standards produced by a dilution process. And the second three calibration standards are independent each with an uncertainty of 2%. These x-values are collected in the vector `xvals`. The vector `yvals` collects the y values as they were generated according to the process depicted in Figure 2 on page 32.

For this example we assume that the substance is not influencing the measurement at all but because a `measurement` in M requires a substance we define `any_sub` to contain an empty substance, i.e. with no further attributes. The tool called `A_tool` contains three attributes: `calibration`, `reading_tol` and the mandatory `do`-function. The `do`-function first fetches the measurement function from the calibration object `calibration`. Second, it computes the uncertainty that is introduced by the *y*-reading mechanism. In this case it is always 5% of the reading. Third, it calls the calibrated measurement function.

The complete output is as follows:

Example 18 Output:

```
x1 = x2 / <2 : 0.04>;
x2 = x3 / <2 : 0.04>;
x3 = <4 : 0.08>;
x4 = <5 : 0.1>;
x5 = <10 : 0.2>;
x6 = <20 : 0.4>;
xvals = ((<4 : 0.08> / <2 : 0.04>) / <2 : 0.04>,
         <4 : 0.08> / <2 : 0.04>, <4 : 0.08>, <5 : 0.1>,
         <10 : 0.2>, <20 : 0.4>);
yvals = (<0.332295 : 0.0165>, <0.393916 : 0.018>,
         <0.426571 : 0.021>, <0.470667 : 0.0225>,
         <0.552436 : 0.03>, <0.888801 : 0.045>);
any_sub = "Arbitrary_substance";
cali_tool = "A_tool"(calibration = fit(xvals, yvals,
        fun=sfunction(x) {p0 + p1 * x;},
        inv=sfunction(x) {(x - p0) / p1;},
        parnum=2, scalex=20, scaley=0.888801, type=1,
         xvals=((<4 : 0.08> / <2 : 0.04>) / <2 : 0.04>,
               <4 : 0.08> / <2 : 0.04>, <4 : 0.08>,
               <5 : 0.1>, <10 : 0.2>, <20 : 0.4>),
        yvals=(<0.332295 : 0.0165>, <0.393916 : 0.018>,
               <0.426571 : 0.021>, <0.470667 : 0.0225>,
               <0.552436 : 0.03>, <0.888801 : 0.045>)),
        reading_tol = 0.05,
        do = dfunction() {
           g = get(inv, calibration);
```

```
            a = value * reading_tol;
            g(<value : a>);
        });
a = 0.5 any_sub cali_tool
        "Environment"(temperature = 293.16 [K]);
<6.62213 : 1.00025>;
```

Note that in this example the calibration data is in the global scope, i.e. some other constructs could reuse them, whereas the calibration object `calibration` is in the inner scope of the instrument definition. The instrument "owns", so to say, its own calibration. In the end, it remains a question of style and taste which attribute to put into which scope. But generally it is a good guideline to attach an attribute to where it *naturally* belongs, e.g. the reading uncertainty belongs to the instrument, a particular calibration, yielding a particular set of function parameters, belongs to one instance of instrument. Physically the same instrument, but operating with a different calibration parameter set, should be considered as a different instrument in the description of a measurement procedure. Therefore the calibration object `calibration` is modeled as an attribute belonging to the instrument. Finally, the second last M-statement describes a measurement of 0.5 units of `A_subst` measured by the instrument `cali_tool`. This quantity is evaluated with `iso(a)` using Algorithm 7.

Notice that in this example the Algorithm 7 cannot be applied directly. Before it can be applied the system executes first the line fitting, yielding the calibration parameter set, and then executes the evaluation of the inverse calibration function, using the well known Newton-iteration.