CubeLib 4.8 | Derived Metrics

Introduction in CubePL and Cube’s derived metrics

September 2023
The Scalasca Development Team
scalasca@fz-juelich.de
Attention

The Cube Derived Metrics User Guide is currently being rewritten and still incomplete. However, it should already contain enough information to get you started and avoid the most common pitfalls.
Contents

1 Introduction into Cube Derived Metrics 3
  1.1 Introduction .................................................. 3
  1.2 Creation of a derived metric in Cube ......................... 3
    1.2.1 Using the C writer .................................... 3
    1.2.2 Using the C++ library .................................. 4
  1.3 The three kinds of derived metrics .......................... 4

2 Reference of CubePL 7
  2.1 Supported calls in CubePL ................................. 7

3 Syntax of CubePL 9
  3.1 Syntax step by step ......................................... 9
    3.1.1 Expressions ............................................. 9
      3.1.1.1 Constants ......................................... 9
      3.1.1.2 Arithmetical Expressions ........................ 9
      3.1.1.3 Boolean Expressions .............................. 9
      3.1.1.4 Function call expressions ......................... 9
    3.1.2 Lambda function (In-place function definition) ............ 10
    3.1.3 Control structures ................................... 10
      3.1.3.1 Condition IF-ELSE ................................ 10
      3.1.3.2 Loop WHILE ...................................... 10
    3.1.4 Variables ............................................... 11
      3.1.4.1 User defined Variables ............................ 11
      3.1.4.2 Predefined Variables .............................. 11
    3.1.5 Different ways to refer another metric ................... 14
      3.1.5.1 Context sensitive reference to another metric .. 14
      3.1.5.2 Context insensitive reference to another metric 15
      3.1.5.3 Direct reference to another metric ............... 15
      3.1.5.4 Definition of an encapsulated calculation within CubePL expression using metrics of Cube. 16
      3.1.5.5 Definition of an initialization expression for ghost metrics within CubePL expression. 17
      3.1.5.6 Definition of an initialization expression for ghost metrics within CubePL expression. 17
    3.2 Grammar of CubePL ....................................... 18

4 Examples of CubePL Expressions 19
  4.1 Simple Examples ........................................... 19
  4.2 Complex Examples .......................................... 19
    4.2.1 Different made-up expressions ........................ 19
    4.2.2 Special metrics of Scalasca ........................... 20
Contents

Bibliography 21
1 Introduction into Cube Derived Metrics

1.1 Introduction

With Cube 4.1.0 derived metrics are introduced and provide a powerful tool for performance data processing.

Unlike normal metrics of the cube, derived metrics do not store any data inside of the cube report, but derive their value according to an expression, formulated using CubePL (Cube Processing Language) syntax ('3').

Cube supports different kinds ('1.3') of derived metrics. Derived metric can be a child metric of any metric of cube or other derived metrics.

Derived metrics are naturally metrics with the value DOUBLE

1.2 Creation of a derived metric in Cube

Derived metrics play the very same role as a normal metric in cube, therefore one proceeds with creation of derived metrics as usual. The only difference is the specification of the CubePL expression.

1.2.1 Using the C writer

To create a derived metric in a C program, one calls cube_metric_set_expression(...) subsequently to cube_def_met(...). Here is a code example:

```c
cube_metric * met = cube_def_met( cube,
   "Derived metric",
   "derivedmetric1",
   "DOUBLE",
   "sec",
   ",",
   "https://doc.web.org/metrics.html#derived",
   "Average execution time",
   NULL,
   CUBE_METRIC_PREDERIVED_EXCLUSIVE
 );

cube_metric_set_expression(met, "metric::time(i)/metric::visits(e)" );
```

Derived metric specification allows to define an CubePL expression, which will be executed only once before the actual metric expression is executed. To specify an initialization expression, one uses the call cube_metric_set_init_expression(...).
Introduction into Cube Derived Metrics

Prederived metrics allow to redefine operators "+" and "-" in the aggregation formula by any user defined expression using same CubePL syntax.

Any such expression should be a formula of "arg1" and "arg2" parameters. E.g. max(arg1, arg2).

To specify an expression for an aggregation operator one uses the call `cube_metric_set_aggr_expression(metric, operator, expression)`, where `operator` can be `CUBE_METRIC_AGGR_PLUS` or `CUBE_METRIC_AGGR_MINUS`.

1.2.2 Using the C++ library

To create a derived metric in C++ programs, one specifies the expression as the last parameter of the metric definition. This parameter can be omitted, in this case the derived metric returns always 0.

Here is a code example:

```c++
Metric * met = cube.def_met("Derived metric", "derivedmetric1", "DOUBLE", "sec", 
"https://doc.web.org/metrics.html#derived", "Average execution time", NULL, 
CUBE_METRIC_PREDERIVED_EXCLUSIVE, 
"${unit}*metric::time(i)/metric::visits(e)", 
'\{ ${unit}=1000; \}', 
'min(arg1,arg2)', 
""
);
```

`CUBE_METRIC_PREDERIVED_EXCLUSIVE` defines the kind of the derived metric (see '1.3' below).

Derived metric specification allows to define a CubePL expression, which will be executed only once before the actual metric expression is executed. This expression is the last argument in the parameter `{ ${unit}=1000; }`.

Operator "+" is redefined like `min(arg1,arg2)`.

Note, that the CubePL expression should be valid at the time of execution. It means, if there is a reference to metric, this metric should be defined before.

1.3 The three kinds of derived metrics

Cube provides an API to calculate different inclusive and exclusive values aggregated over different dimensions.

In the context of derived metrics it is sometimes useful, when evaluation of the expression is done after the aggregation of the metrics in the expression, sometimes it is needed to aggregate values of the derived metrics.

Therefor Cube provides three kinds of derived metrics:

1. **Prederived exclusive metric** - metric, which value of which expression is observed
1.3 The three kinds of derived metrics

as "being stored" inside of cube report and it behaves as a usual exclusive (along call tree) metric.

One specifies this type of metric using constant

CUBE_METRIC_PREDERIVED_EXCLUSIVE

For this metric one can redefine only operator "+". Operator "-" will be ignored.

2. **Prederived inclusive metric** - metric, which value of which expression is observed as "being stored" inside of cube report and it behaves as a usual inclusive (along call tree) metric.

One specifies this type of metric using constant

CUBE_METRIC_PREDERIVED_INCLUSIVE

For this metric one can redefine both operators, "+" and "-".

3. **Postderived metric** - metric, which expression evaluated only after metrics, which were used inside of the expression, get aggregated according to the aggregation context (along system tree or along call tree).

If expression doesn't contain references to another metrics, no aggregation is done. In this case the expression gets calculated and its value returned.

One specifies this type of metric using constant

CUBE_METRIC_POSTDERIVE

This metric doesn't allow to redefine any operators, because calculation of this metric is performed AFTER aggregation within another metric.
## 2 Reference of CubePL

### 2.1 Supported calls in CubePL

Here is the table of currently supported calls in CubePL:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>123.34</td>
<td>a numerical constant</td>
</tr>
<tr>
<td>&quot;STRING&quot;</td>
<td>a string constant</td>
</tr>
<tr>
<td>-x</td>
<td>negative value of x</td>
</tr>
<tr>
<td>()</td>
<td>grouping (priority)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>+ - / *</td>
<td>arithmetical operations</td>
</tr>
<tr>
<td>A^x</td>
<td>A in power of x</td>
</tr>
<tr>
<td>&gt; &lt; &gt;= &lt;= != ==</td>
<td>numerical comparison</td>
</tr>
<tr>
<td>and or xor not</td>
<td>boolean operations</td>
</tr>
<tr>
<td>eq</td>
<td>string comparison</td>
</tr>
<tr>
<td>seq</td>
<td>case insensitive string comparison</td>
</tr>
<tr>
<td>=~ /expression/</td>
<td>matching of regular expression</td>
</tr>
<tr>
<td>sin(x) cos(x) asin(x) acos(x) tan(x) cot(x) atan(x) acot(x)</td>
<td>trigonometrical functions</td>
</tr>
<tr>
<td>ln(x)</td>
<td>natural logarithm of x</td>
</tr>
<tr>
<td>exp(x)</td>
<td>natural exponent of x</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root value of x</td>
</tr>
<tr>
<td>random(x)</td>
<td>random number from 0 to x</td>
</tr>
<tr>
<td>pos(x)</td>
<td>returns x, if x&gt;, or 0</td>
</tr>
<tr>
<td>neg(x)</td>
<td>returns x, if x&lt;0, or 0</td>
</tr>
<tr>
<td>sgn(x)</td>
<td>returns -1, if x&lt;0, 0 if x is 0 and 1 is x&gt;0</td>
</tr>
<tr>
<td>abs(x)</td>
<td>returns absolut value</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>returns smallest integer value, which is bigger than x</td>
</tr>
<tr>
<td>floor(x)</td>
<td>returns biggest integer value, which is smaller than x</td>
</tr>
<tr>
<td>min(x,y)</td>
<td>returns smaller value of x and y</td>
</tr>
<tr>
<td>max(x,y)</td>
<td>returns bigger value of x and y</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>uppercase(x)</td>
<td>returns uppercase version of x</td>
</tr>
<tr>
<td>lowercase(x)</td>
<td>returns lowercase version of x</td>
</tr>
<tr>
<td>env(&quot;x&quot;)</td>
<td>returns a value of the environment variable x</td>
</tr>
<tr>
<td>local(var)</td>
<td>Declares (to compile time) a variable var as local. It cannot be redeclared later</td>
</tr>
<tr>
<td>global(var)</td>
<td>Declares (to compile time) a variable var as global. It cannot be redeclared later</td>
</tr>
<tr>
<td>${var}</td>
<td>0th element of the variable with name var</td>
</tr>
<tr>
<td>${var}[x]</td>
<td>x-th element of the variable with name var</td>
</tr>
<tr>
<td>sizeof(var)</td>
<td>number of elements in the variable var</td>
</tr>
<tr>
<td>defined(var)</td>
<td>returns true (1), if the variable var is defined, otherwise it returns false (0)</td>
</tr>
<tr>
<td>metric::[uniq_name](i</td>
<td>e</td>
</tr>
<tr>
<td>metric::context::[uniq_name](i</td>
<td>e</td>
</tr>
<tr>
<td>metric::fixed::[uniq_name](i</td>
<td>e</td>
</tr>
<tr>
<td>metric::call::[uniq_name](c_id, i</td>
<td>e</td>
</tr>
<tr>
<td>cube::metric::set::<a href="%22X%22," title="Y">uniq_name</a></td>
<td>sets a property X of a metric uniq_name to value Y. Currently supported are only &quot;value&quot; with value &quot;VOID&quot; or else</td>
</tr>
<tr>
<td>cube::metric::get::<a href="%22X%22">uniq_name</a></td>
<td>returns a property X of a metric uniq_name. Currently supported are</td>
</tr>
<tr>
<td></td>
<td>1. unique name - returns unique name of a metric uniq_name</td>
</tr>
<tr>
<td></td>
<td>2. display name - returns display name of a metric uniq_name</td>
</tr>
<tr>
<td></td>
<td>3. uom - returns unit of measurement of a metric uniq_name</td>
</tr>
<tr>
<td></td>
<td>4. dtype - returns data type of a metric uniq_name as a string</td>
</tr>
<tr>
<td></td>
<td>5. url - returns url of online help for the metric uniq_name</td>
</tr>
<tr>
<td></td>
<td>6. description - returns description of a metric uniq_name</td>
</tr>
<tr>
<td></td>
<td>7. value - returns property &quot;value&quot; for the metric uniq_name. Its value &quot;VOID&quot; deactivates metric and it returns always a zero.</td>
</tr>
</tbody>
</table>
3 Syntax of CubePL

3.1 Syntax step by step

This chapter introduced the CubePL syntax of the most common programming structures.

3.1.1 Expressions

3.1.1.1 Constants

Any constant expression is an expression. eg. 123.0, "someString"

3.1.1.2 Arithmetical Expressions

Common mathematical notation of an arithmetical expression is valid in CubePL. Any arithmetical expression is an expression. eg. 123.0 + 23.6, sin(23)

3.1.1.3 Boolean Expressions

Common literal notation of a boolean expression is valid in CubePL. Any boolean expression is an expression.

expression or expression

expression and expression

expression xor expression

not expression

Any expression can be a term of a boolean expression. Non zero value is observed as TRUE, zero value is FALSE.

3.1.1.4 Function call expressions

Function calls have the following syntax:

name ( expression )
3 Syntax of CubePL

3.1.2 Lambda function (In-place function definition)

To define a function in-place, one uses the following syntax:

```c
{  
  [statement];  
  [statement];  
  ...  
  return [expression];  
}
```

In-place definition of a function is an expression. It means, it can appear everywhere, where one can use an expression.

3.1.3 Control structures

Control structures like if-else or while are statements.

3.1.3.1 Condition IF-ELSE

One can execute series of statements under a condition using the short form of the if statement:

```c
if ( condition )
{
  [statement];  
  [statement];  
  ...  
}
```

or full form with else:

```c
if ( condition )
{
  [statement];  
  [statement];  
  ...  
}
else
{
  [statement];  
  [statement];  
  ...  
}
```

condition is a boolean expression

3.1.3.2 Loop WHILE

One can execute a series of statements as long as a condition is true using the while statement:

```c
while ( condition )
{
  [statement];  
  [statement];  
  ...  
}
```
Sequence of statements will be executed till condition is not fulfilled, max 1000000000 times.

### 3.1.4 Variables

CubePL allows to work with memory, by using variables. All variables are multi-typed:

- In string context value of a numeric variable is presented in string format.
- In numerical context a string variable is converted to its numerical representation, if possible. Otherwise it is 0.

All variables are arrays. Indexless access to the variable assumes mutually index value 0.

#### 3.1.4.1 User defined Variables

The user can define a variable using the syntax:

```plaintext
${ name } = [expression];
```

or

```plaintext
${ name }[index] = [expression];
```

Currently its name is a fixed string of characters. In later versions of CubePL it will allow any expression.

Example of a string context:

```plaintext
${name} seq "STRING"
```

Example of a numeric context:

```plaintext
${name} >= 0.34
```

Example of an array access to the variable:

```plaintext
${name}[${i}] >= 0.34
```

Access to the variable is an expression.

Index for the access to the value is also an expression.

#### 3.1.4.2 Predefined Variables

Cube provides a set of predefined variables for every calculation, which values are independent. Following predefined variables do contain the general information about the cube:
## Syntax of CubePL

<table>
<thead>
<tr>
<th>Predefined variable</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cube::#mirrors</code></td>
<td>Number of mirrors in cube file</td>
</tr>
<tr>
<td><code>cube::#metrics</code></td>
<td>Number of metrics in cube file</td>
</tr>
<tr>
<td><code>cube::#root::metrics</code></td>
<td>Number of root metrics in cube file</td>
</tr>
<tr>
<td><code>cube::#regions</code></td>
<td>Number of regions in cube file</td>
</tr>
<tr>
<td><code>cube::#callpaths</code></td>
<td>Number of call paths in cube file</td>
</tr>
<tr>
<td><code>cube::#root::callpaths</code></td>
<td>Number of root call paths</td>
</tr>
<tr>
<td><code>cube::#locations</code></td>
<td>Number of locations in cube file</td>
</tr>
<tr>
<td><code>cube::#locations::void</code></td>
<td>Number of void locations in cube file</td>
</tr>
<tr>
<td><code>cube::#locations::nonvoid</code></td>
<td>Number of nonvoid locations in cube file</td>
</tr>
<tr>
<td><code>cube::#locationgroups</code></td>
<td>Number of location groups in cube file</td>
</tr>
<tr>
<td><code>cube::#locationgroups::void</code></td>
<td>Number of void location groups in cube file</td>
</tr>
<tr>
<td><code>cube::#locationgroups::nonvoid</code></td>
<td>Number of nonvoid location groups in cube file</td>
</tr>
<tr>
<td><code>cube::#stns</code></td>
<td>Number of system tree nodes in cube file</td>
</tr>
<tr>
<td><code>cube::#rootstns</code></td>
<td>Number of root system tree nodes in cube file</td>
</tr>
<tr>
<td><code>cube::filename</code></td>
<td>Name of the cube file</td>
</tr>
</tbody>
</table>

CubePL engine defines a set of variables, which do depend on an index, their "id". Using call `sizeof(...)` one can run over them and inspect within CubePL expression.

<table>
<thead>
<tr>
<th>Predefined variable</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cube::metric::unit::name</code></td>
<td>Unique name of the metric</td>
</tr>
<tr>
<td><code>cube::metric::disp::name</code></td>
<td>Display name of the metric</td>
</tr>
<tr>
<td><code>cube::metric::url</code></td>
<td>URL of the documentation of the metric</td>
</tr>
<tr>
<td><code>cube::metric::description</code></td>
<td>Description of the metric</td>
</tr>
<tr>
<td><code>cube::metric::dtype</code></td>
<td>Data type of the metric</td>
</tr>
<tr>
<td><code>cube::metric::uom</code></td>
<td>Unit of measurement of the metric</td>
</tr>
<tr>
<td><code>cube::metric::expression</code></td>
<td>CubePL expression of the metric</td>
</tr>
<tr>
<td><code>cube::metric::initexpression</code></td>
<td>CubePL initialisation expression of the metric</td>
</tr>
<tr>
<td><code>cube::metric::parentd::id</code></td>
<td>ID of the parent of the metric</td>
</tr>
<tr>
<td><code>cube::metric::#children</code></td>
<td>Number of children in the metric</td>
</tr>
<tr>
<td><code>cube::callpath::mod</code></td>
<td>Module of the call path</td>
</tr>
<tr>
<td><code>cube::callpath::line</code></td>
<td>File line of the call path</td>
</tr>
<tr>
<td><code>cube::calleeid</code></td>
<td>ID of the callee region</td>
</tr>
<tr>
<td><code>cube::callpath::#children</code></td>
<td>Number of children in the call path</td>
</tr>
<tr>
<td><code>cube::callpath::parent::id</code></td>
<td>ID of the parent callpath</td>
</tr>
<tr>
<td><code>cube::region::name</code></td>
<td>Name of the region</td>
</tr>
<tr>
<td><code>cube::region::mangled::name</code></td>
<td>Mangled name of the region</td>
</tr>
<tr>
<td><code>cube::region::paradigm</code></td>
<td>Name of the paradigm of the region</td>
</tr>
<tr>
<td><code>cube::region::role</code></td>
<td>Name of the role the region is playing within the paradigm</td>
</tr>
<tr>
<td><code>cube::region::url</code></td>
<td>URL with the description of the region of the call path</td>
</tr>
</tbody>
</table>
CubePL engine sets a series of context sensitive variables, which value depends on the parameters, for which the derived metric is being calculated. Their value can be used to refer to the values of context insensitive variables described above:
### 3 Syntax of CubePL

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculation::region::id</td>
<td>ID of the region, for what is the metric being calculated</td>
</tr>
<tr>
<td>calculation::region::#elements</td>
<td>Number of selected regions, for which is metric being calculated. Usually bigger than 1 only for postderived metrics</td>
</tr>
<tr>
<td>calculation::sysres::id</td>
<td>ID (local within system resource type) if the system resource, for what is the metric being calculated</td>
</tr>
<tr>
<td>calculation::sysres::sysid</td>
<td>ID (global) if the system resource, for what is the metric being calculated</td>
</tr>
<tr>
<td>calculation::sysres::kind</td>
<td>Type of the system element:</td>
</tr>
<tr>
<td></td>
<td>• 0 = unknown</td>
</tr>
<tr>
<td></td>
<td>• 5 = system tree node</td>
</tr>
<tr>
<td></td>
<td>• 6 = location group</td>
</tr>
<tr>
<td></td>
<td>• 7 = location</td>
</tr>
<tr>
<td>calculation::sysres::#elements</td>
<td>Number of selected system tree elements, for which is metric being calculated. Usually bigger than 1 only for postderived metrics</td>
</tr>
</tbody>
</table>

### 3.1.5 Different ways to refer an another metric

#### 3.1.5.1 Context sensitive reference to another metric

To use values of another metric in the same calculation context, one uses syntax:

```
metric::[uniq.name]( modificator, modificator )
```

or

```
metric::[uniq.name]( modificator)
```

or

```
metric::[uniq.name]()
```

There are three version of this call:

1. with two arguments (call path and system);
2. with one argument (call path);
3. with no argument (an arguments takes as '*').


Metric reference is an expression.
3.1.5.2 Context insensitive reference to another metric

To use values of another metric in the same fixed calculation context (e.g. aggregated over threads), one uses syntax:

```
metric::fixed::[uniq.name]( modificator, modificator )
```

or

```
metric::fixed::[uniq.name]( modificator)
```

or

```
metric::fixed::[uniq.name]()
```

There are three versions of this call:
1. with two arguments (call path and system);
2. with one argument (call path);
3. with no argument (an argument takes as '*').

```
```

Metric reference is an expression.

3.1.5.3 Direct reference to another metric

To use values of another metric with a specific call path id and system resource id (!), one uses syntax:

```
metric::call::[uniq.name]( callpath id, modificator, sysres id, modificator )
```

or

```
metric::call::[uniq.name]( callpath id, modificator )
```

There are two versions of this call:
1. with four arguments (call path and system); - Only calculation of an inclusive or exclusive value is performed
2. with one argument (call path); - Aggregation over system tree is performed additionally to the calculation of the inclusive or exclusive value for the calltree id

```
modificator specifies flavor of the calculation: i - inclusive, e - exclusive.
```

Metric reference is an expression.

Note that "sysres id" is a global identifier and can be referred using ${calculation::sysres::id}.


3.1.5.4 Definition of an encapsulated calculation within CubePL expression using metrics of Cube.

One special mechanism of CubePL processing engine allows some level of calculation separation. A derived metric can be created within another CubePL expression. Such “ghost” metric gets its name and properties and exists inside of the cube object as a casual metric. Only difference to the casual metric is in the fact that ghost metric is not visible in GUI and tools and is not stored inside of the metric tree of the cube file.

One can refer such metric as a casual metric using metric references (see '3.1.5.1', '3.1.5.2' and '3.1.5.3').

Example for definition of such metric:

```cube::metric::nvisitors(e)
<<
{ 
	${return}=0;
	if ( ${cube::locationgroup::void}[${calculation::sysres::id}] != 1)
	{ 
		if (metric::visits()>0 )
		{ 
			${return} = 1;
		};
		return ${return};
	}
	${visitors} = metric::fixed::nvisitors(e);
}

where

cube::metric::prederived::nvisitors(e)

gives a type (prederives, exclusive) and unique name (nvisitors) of this metric. Unique name is used then later to refer to this metric via

${visitors} = metric::fixed::nvisitors(e);

There are kinds of metrics, which can be defined on such manner:

1. cube::metric::prederived::name(e) - An exclusive prederived metric with the name name;
2. cube::metric::prederived::name(i) - An inclusive prederived metric with the name name;
3. cube::metric::postderived::name - A postderived metric with the name name.

Notice that once the metric with some unique name created it exists whole lifetime of the cube object. Therefore one can refer to some somewhere previously defined ghost metric from any following it CubePL expressions.

Ghost Metric definition is a statement.
### 3.1.5.5 Definition of an initialization expression for ghost metrics within CubePL expression.

One can specify within a CubePL expression an initialisation phase for previously created ghost metric. For that purpose one uses expression `cube::init::metric::[name]`. Notice that named metric should be known by the moment of compilation of the CubePL expression.

Example for definition of such metric:

```cpp
cube::init::metric::init::nvisitors
<<
 { global(nvisitors);
 } >>;
```

where

`cube::init::metric::nvisitors`

uses an unique name (nvisitors) of the metric.

Definition of the initialisation phase of a metric is a statement.

### 3.1.5.6 Definition of an initialization expression for ghost metrics within CubePL expression.

One can specify within a CubePL expression an aggregation operator "+" or "-" for previously created ghost metric. For that purpose one uses expression `cube::metric::plus::[name]` or `cube::metric::minus::[name]`. Notice that named metric should be known by the moment of compilation of the CubePL expression.

Example for definition of such metric:

```cpp
cube::init::metric::plus::nvisitors
<<
 max( arg1, arg2)
 >>;
```

or

```cpp
cube::init::metric::minus::nvisitors
<<
 min( arg1, arg2)
 >>;
```

where

`cube::init::metric::plus::nvisitors`

or

`cube::init::metric::minus::nvisitors`

uses an unique name (nvisitors) of the metric.

Definition of the initialisation phase of a metric is a statement.
3 Syntax of CubePL

3.2 Grammar of CubePL

Here will be a full grammar of CubePL expressions (later).
4 Examples of CubePL expressions

4.1 Simple Examples

1. Calculation of an arithmetical expression
   \[ 123.4 + 234 - (23)^2 \]
2. Calculation of an arithmetical expression with different functions
   \[ \sin(23 + \ln(12)) \]

4.2 Complex Examples

4.2.1 Different made-up expressions

1. Definition of a constant function
   
   ```
   { return 24; }
   ```

2. Definition of a more complex function
   
   ```
   { return \sin({\ return 1;}); }
   ```

3. Definition of a function with an access to one variable
   
   ```
   {${a}=123; \ return \${a}; }
   ```

4. Definition of a function with a control structure
   
   ```
   {${a}=\text{metric::visits}();
   \${b}=0;
   \text{if }{\ ${a}>100}()
   \{ \${b}=\text{metric::time}(); \};
   \ \ return \${b};}
   ```

5. Definition of a function with a loop structure
   
   ```
   {${a}=0; \${b}=0;
   \text{while }{\ ${a}<123}()
   \{ \${b}=${b}+\text{metric::time}();
   \${a}=${a}+1;
   \};
   \ return \${b};}
   ```

6. Definition of a function with a access to predefined variable
   
   ```
   {${a}=0;
   ```
4 Examples of CubePL expressions

if (${calculation::region::name}[${calculation::callpath::id}] =~ /^MPI_/)
{
  ${a}=metric::time();
};
return ${a};
}

4.2.2 Special metrics of Scalasca

1. Calculation of an average runtime of a call path (Kenobi metric, postderived)

\[
\frac{\text{metric::time}(i)}{\text{metric::visits}(e)}
\]

2. Calculation of the time, spend in MPI synchronization calls

a) (initialization)

\{
  global(mpi.synchroization);
  ${i}=0;
  while( ${i} < ${cube::#callpaths} )
  {
    ${mpi.synchroization}[$i] = 0;
    ${regionid} = ${cube::callpath::calleeid}[$i];
    if ( (${cube::region::paradigm}[ ${regionid} ] seq "mpi")
        and
        (${cube::region::name}[${regionid}] seq "mpi.barrier" )
        or
        (${cube::region::name}[${regionid}] seq "mpi.win.post" )
        or
        (${cube::region::name}[${regionid}] seq "mpi.win.wait" )
        or
        (${cube::region::name}[${regionid}] seq "mpi.win.start" )
        or
        (${cube::region::name}[${regionid}] seq "mpi.win.complete" )
        or
        (${cube::region::name}[${regionid}] seq "mpi.win.fence" )
        or
        (${cube::region::name}[${regionid}] seq "mpi.win.lock" )
        or
        (${cube::region::name}[${regionid}] seq "mpi.win.unlock" ))
    {
      ${mpi.synchroization}[$i] = 1;
    }
    ${i} = ${i} + 1;
  }
  return 0;
\}

b) (actual calculation)

\{
  ${a}=0;
  if ( ${mpi.synchroization}[${calculation::callpath::id}] == 1 )
  {
    ${a} = metric::time(*,*)-metric::omp_idle_threads(*,*);
  }
  return ${a};
\}

3. Calculation of thecComputational load imbalance (single participant)

– NO EXAMPLE YET –