Quotations

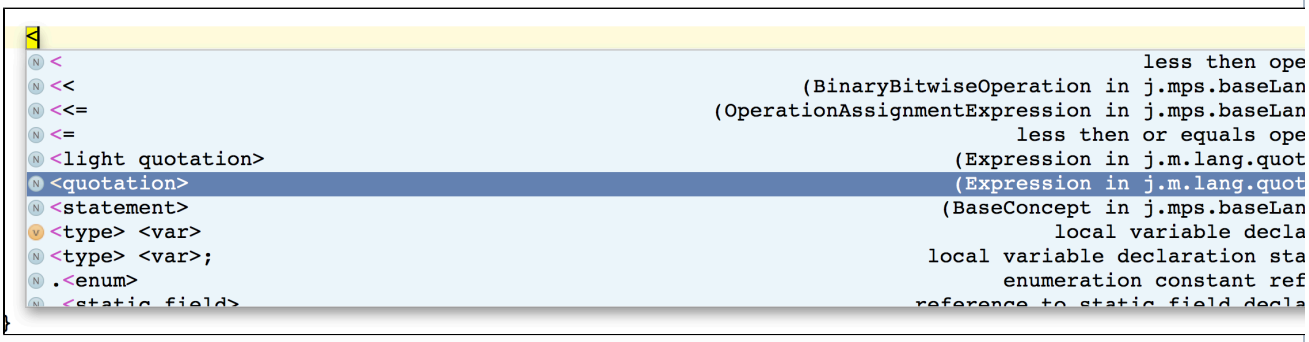
Quotations

A quotation is a language construct that lets you easily create a node with a required structure. Of course, you can create a node using the smodelLanguage and then populate it with appropriate children, properties and references by hand, using the same smodelLanguage. However, there's a simpler - and more visual - way to accomplish this.

The two following constructs will build identical nodes, the first one uses quotation, the second plan model API:

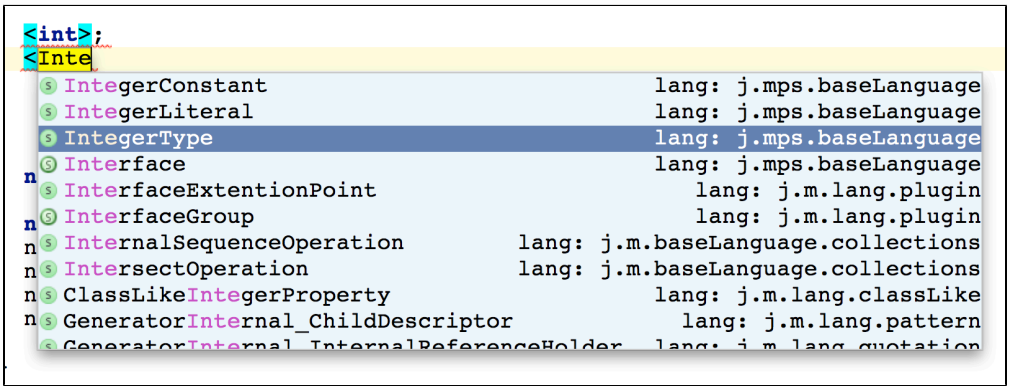
```
node<IntegerType> node = <int>;
node<IntegerType> node2 = new node<IntegerType>();
```

When creating a quotation, type the left angle bracket symbol '<' and pick 'quotation' from the completion menu.



Alternatively, typing '<q' followed by Control + Space will be even faster.

When specifying the top-most concept for the quotation, use the concept name, e.g. IntegerType, not its alias, e.g. int .



Quotations as well as light quotations are defined int the jetbrains.mps.lang.quotation language. You need this language to be set as a used language in order to be able to use it in your models.

A quotation is an expression, whose value is the MPS node written inside the quotation. Think about a quotation as a "node literal", a construction similar to numeric constants and string literals. That is, you write a literal if you statically know what value do you mean. So inside a quotation you don't write an expression, which evaluates to a node, you rather write the node itself. For instance, an expression `2 + 3` evaluates to `5`, an expression `< 2 + 3 >` (angled braces being quotation braces) evaluates to a node PlusExpression with leftOperand being an IntegerConstant 3 and rightOperand being IntegerConstant *5*.

The following two constructs again create the same AST, now the quotation approach yields clear benefits in code brevity:

```
node<PlusExpression> node3 = <2 + 3>;

node<PlusExpression> node4 = new node<PlusExpression>();
node4.leftExpression = new node<IntegerConstant>();
node4.leftExpression : IntegerConstant.value = 2;
node4.rightExpression = new node<IntegerConstant>();
node4.rightExpression : IntegerConstant.value = 3;
```

Antiquotations

For it is a literal, a value of quotation should be known statically. On the other hand, in cases when you know some parts (i.e. children, referents or properties) of your node only dynamically, i.e. those parts that can only be evaluated at runtime and are not known at design time, then you can't use just a quotation to create a node with such parts.

The good news, however, is that if you know the most part of a node statically and you want to replace only several parts by dynamically-evaluated nodes you can use antiquotations. An antiquotation can be of 4 types: child, reference, property and list antiquotation. They all contain an expression, which evaluates dynamically to replace a part of the quoted node by its result. Child and referent antiquotations evaluate to a node, property antiquotation evaluates to string and list antiquotation evaluates to a list of nodes.

For instance, you want to create a ClassifierType with the class ArrayList, but its type parameter is known only dynamically, for instance by calling a method, say, "computeMyTypeParameter()".

Thus, you write the following expression: < ArrayList < %(computeMyTypeParameter())% > >. The construction %(...)% her e is a node antiquotation.

You may also antiquotate reference targets and property values, with ^(...)^ and $(...)$, respectively; or a list of children of one role, using *(...)*.

a) If you want to replace a node somewhere inside a quoted node with a node evaluated by an expression, you use node antiquotation, that is %( )%. As you may guess there's no sense to replace the whole quoted node with an antiquotation with an expression inside, because in such cases you could instead write such an expression directly in your program.

So node antiquotations are used to replace children, grandchildren, great-grandchildren and other descendants of a quoted node. Thus, an expression inside of antiquotation should return a node. To write such an antiquotation, position your caret on a cell for a child and type "%".

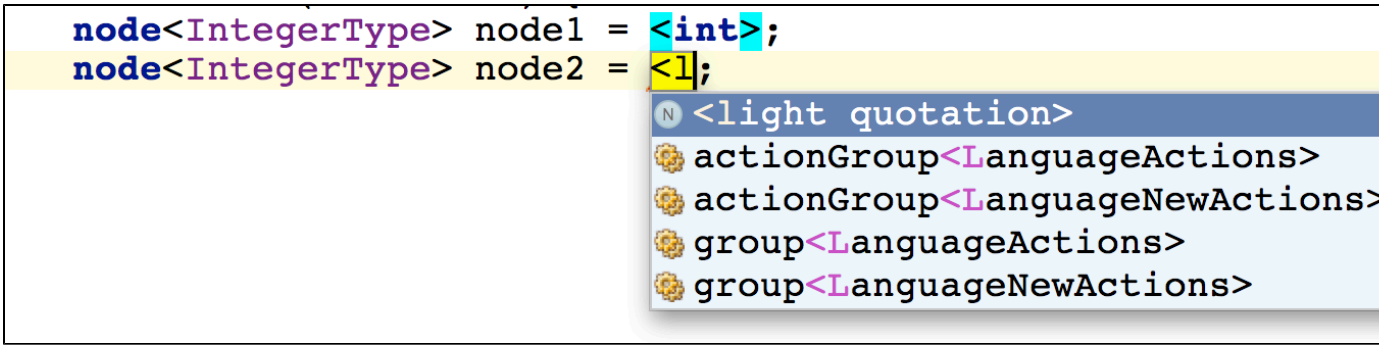
b) If you want to replace a target of a reference from somewhere inside a quoted node with a node evaluated by an expression, you use reference antiquotation, that is ^(...)^ . To write such an antiquotation, position your caret on a cell for a referent and type "^".

c) If you want to replace a child (or some more deeply located descendant), which is of a multiple-cardinality role, and if for that reason you may want to replace it not with a single node but rather with several ones, then use child list (simply list for brevity) antiquotations, *( ). An expression inside a list antiquotation should return a list of nodes, that is of type * *nlist<..>* * or compatible type (i.e.* **{}list<node<..>>* * is ok, too, as well as some others). To write such an antiquotation, position your caret on a cell for a child inside a child collection and type "" . You cannot use it on an empty child collection, so before you press "" you have to enter a single child inside it.

d) If you want to replace a property value of a quoted node by a dynamically calculated value, use property antiquotation $( )$. An expression inside a quotation should return string, which will be a value for an antiquoted property of a quoted node. To write such an antiquotation, position your caret on a cell for a property and type "$".

Light quotations (quotation builders)

Using quotations has its downsides, though. For example, if you're creating quotations of a language in its own definition, you're creating a bootstrapping cycle that needs special treatment during language compilation (See Removing Bootstrapping Dependency). Light quotations provide an alternative approach that will help you eliminate such issues.



The following two constructs, one using a quotation and the second using a light quotation are equivalent:

```
node<IntegerType> node1 = <int>;
node<IntegerType> node2 = <IntegerType()>;
```

Unlike direct model API usage, Light quotations remain conveniently usable for deeper hierarchies:

```
node<PlusExpression> node3 = <2 + 3>;
node<PlusExpression> node4 = <PlusExpression(
  leftExpression: IntegerConstant(value: 2),
  rightExpression: IntegerConstant(value: 3))>;
```