# Step To Core Translation Programmer Guide

## Translation steps:

1. Parsing Step input program
2. Output the Step program in format of XML
3. Module expressions translation
4. Translation of types and variable scopes
5. Output the translated Step program as XML version of Core program
6. Output additional info

## Parsing Step input program:

The translator gets the Step program as input. We use the Step parser to parse the program and insert it into the memory data structures. While parsing, the translator makes XML version of the Step program with Ids (used by additional info). The format of this XML file is according to file "step.dtd". There is usage of simple Visitor pattern to create the output.
In order to print initial values of variables we added function "public void setConstraint(Expr constraint) " to class "Var", that takes as input argument - expression that is initialization of the variable. The function sets the information to objects' constraint field; so later it is easy to get the value directly from the object.

## Module expressions translation

The hardest part is translation of module expressions, since Step has much more possibilities to combine modules.
At first stage of translation the "Translator" builds the tree of modules used by Step program. The building begins from the last module (in Step it is the "main" module where the run starts), so the tree contains modules that have to be called during the program execution, and troughs all the unreachable ones. After the tree is built, we pass through the tree and replace all cases of case distinction, combination of 3 or more modules, augmentations and restrictions.
The next stage is replacing of renaming and hiding operations. To hide or rename some variable or transition in some module expression EXPR, we have to find the basic module containing the variable and to make its new version (with local variable or transition/ renamed variable or transition) and also to make new version of all expressions in the tree path between the EXPR and the basic module.
Here is the end of Step_To_Step translation, at this point we have correct version of initial Step program, but without complex structures.

# Translation of types and variable scopes

Now the Step to Core translation is beginning. The types in Step can be declared in global or local space, but their visibility is global so we have just to move the local declarations. After this we are changing the variable declarations, according to rules mentioned in Step2Core.doc document.
Now we are ready to build the XML version of the Core program, which is our main target.

**Data Structures:**

1. Step parser creates data structures that represent all possible Step language structures (e.g. module, module combination, expression, etc.).
2. Class ProgramTree – tree of modules in program, when the root is the last ("main") module. Each tree node has the link to object of type StepTreeNode, structure contains information about module, module instance or module combination. The tree is very important; it is used to simplify Step module combinations.
3. Class AddInfo is class that contains Step and Core XML ids, <u>all Step parser structures inherit it</u>. Initialization of Step ids happens while creating the XML Step version, and in case that the structure was changed in nontrivial way, we initialize Core ids also (if the structure wasn't changed the Core id is added in the time of creating Core XML version)
4. Class XmlChange – the structure contains information about one nontrivial translation, it has ids of structures and the description of change and can print itself in XLINK format.
5. Class PrettyStepMtsPrinter – used to print source Step program in its' XML representation with adding Ids to all the elements.
6. Class PrettyStepPrinter – for the same purpose as (5), with the difference that it works on data structures, expressions and types, while (5) on the upper structures of the source program – modules and their combinations.
7. Class PrettyCoreMtsPrinter - used to print target Core program in its' XML representation with adding Ids to all the elements.
8. Class PrettyCorePrinter – for the same purpose as (7), with the difference that it works on data structures, expressions and types, while (7) on the upper structures of the source program – modules and their combinations. StepTreeNode class used to print module combinations.