Version 1.4.0 June 2015

Credits

Contents

1 — Introduction

Background   1.1

The MetaR software `http://MetaR.campagnelab.org` is an example of a new kind of interactive tool for data analysis. It was developed by the Campagne laboratory using the Meta Programming System (MPS) (see `http://www.jetbrains.com/mps` [Dmi04]. MPS is a mature Language Workbench that makes it relatively easy to create new languages and tools to help users of these languages [Cam14].

Intended audience   1.2

This booklet is designed to teach how to use MetaR for data analysis. In the first chapters, we will assume that you have no prior scripting or programming experience, but will expect you to know how to use a computer.

High-level data abstractions

MetaR is designed to make it easier to conduct data analysis. To achieve this goal, and in contrast to programming languages such as the R language, Julia or Python, that are often low-level and require good programming skills, MetaR offers high-level data abstractions and provides assistance in manipulating data. High-level abstractions used in MetaR analyses include the following concepts:

1. Tables, Columns, Column Groups and Group usages,
2. Analyses
3. Plot
4. Model

These high-level abstractions will be explained in the following chapters.

R program generation

Analyses developed with MetaR are transformed into R programs when the user needs to execute them. MetaR is tightly integrated with MPS to make it seamless to run analyses without prior knowledge of the R language.

> **R** Note that while most users can use MetaR without knowledge of the R language, all users will need to install the R language in order to execute MetaR analyses.

Docker integration

Starting with version 1.3.1, MetaR can run the R scripts that it generates into Docker containers. Docker is a technology that makes it easier to obtain reproducible script executions. When MetaR does not use Docker, it is possible for the R runtime to try to install a package the first time you run an analysis with a statement that requires the package. While this should not be a problem, we found that R package installation is brittle and can sometimes fail at various

time points, for a variety of reasons[1]. To avoid such problems, which tend to occur when we give a training sessions, we now support running Analyses with Docker. Chapter 4 explains how to configure MetaR to use Docker for reproducible analyses.

Composable language

MetaR is built as an MPS language. In contrast to languages built with compiler technology, MPS languages are composable. Briefly, language composition is the ability to compose, or combine, two or more languages. This ability is particularly useful to extend MetaR. It is possible to define micro-languages, made of one or two statement types, that when composed with MetaR make it possible to customize MetaR for specific application domains. The EdgeR language provides an example of composition, where an R/Bioconductor package called EdgeR has been integrated with MetaR and exposes a new kind of statement to perform calls of differential expression. See Chapters 5 and 10 to learn more about language composition in MetaR.

Solutions and Models   1.4

Developing an analysis with MetaR is done by creating nodes of the MetaR concepts in MPS models. Models exist in MPS solutions. To learn how to create an MPS solution, read the preview of the MPS language workbench available at `http://campagnelab.org/publications/our-books/`[Cam14], or watch the beginning of the MetaR training video at `http://campagnelab.`

---

[1]For instance, because a new version of a package has become available which breaks compilation of the package on your machine.
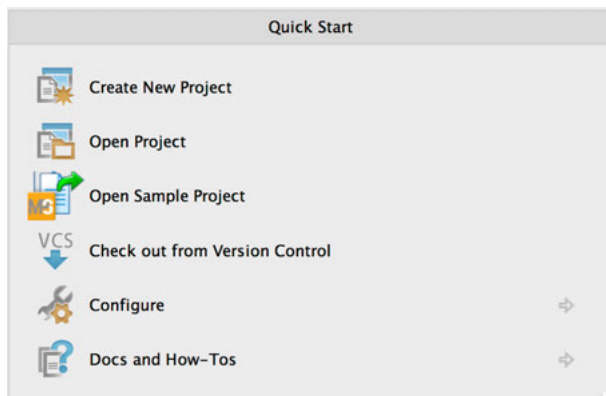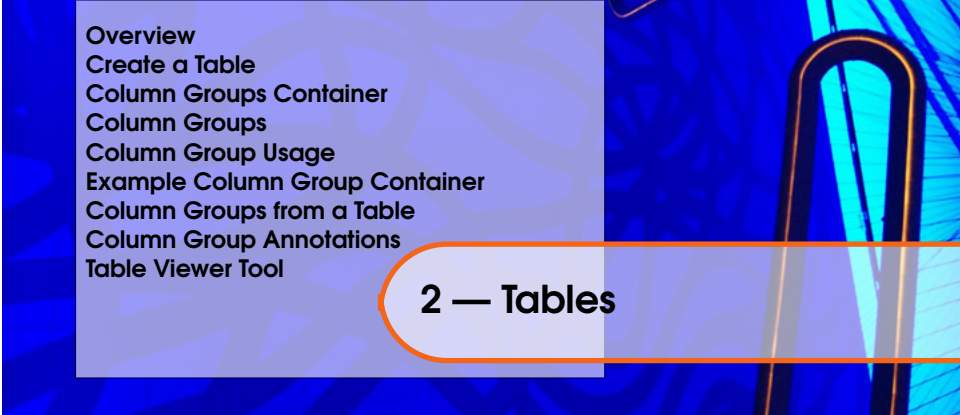
Figure 1.1: **The Quick Start menu.** This menu is displayed when you first open MPS, or when you have no project currently open in MPS. The first item in the menu is used to create a new project.

org/software/metar/video-tutorials/. Figures 1.1-1.2 provide a brief walk-through of the steps you should follow to create a project and solution. After the project open, open the project tab(see [Cam14]) and right-click on the solution name. Import the metaR devkit and create a model.

You can create as many models as you need to store your analyses. In the following chapter, we assume that you have created a solution and at least one model in this solution.

Figure 1.2: **The New Project Dialog.** Use this dialog to create a new Solution. Solutions are used to store models and express your solutions to specific analysis problems. Select the "Solution Project" project type on the left panel, name the project, and name the solution you wish to create.
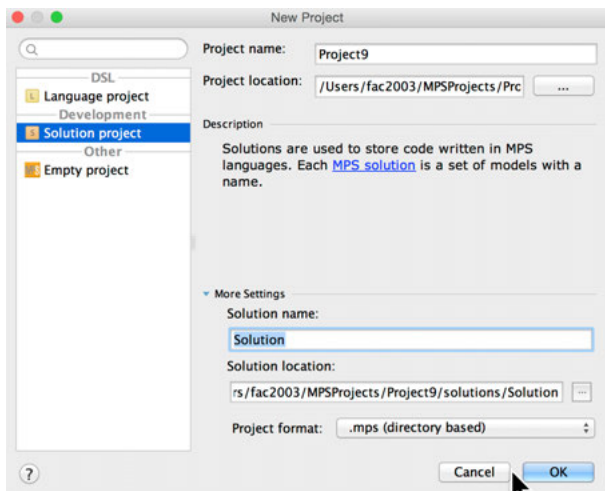
2 — Tables

Overview 2.1

Before you start an analysis, you will need to define Table nodes that represent the data files needed in the analysis. MetaR explicitly models files that contain tables of data. This is done so that you can easily refer to these tables, without having to remember where the original file is located on your computer.

In this Chapter, you will learn how to

1. define a MetaR table,
2. adjust the types of the columns of the data described in the file,
3. annotate a table with groups,
4. link groups in column group usage.

Create a Table 2.2

To create a Table, right-click on a model in the project tab and select right-click ⟩ New ⟩ o.c.metar.tables ⟩ Table . This will create an empty table, as shown in Figure 2.1. Tables have a name, a `pathToResolve` attribute and a list of

columns. The following paragraphs describe these attributes.

name

The table name is set automatically from the path when you use the file selection button. You can change the name to match your analysis needs and make it easier to remember what is in the table.

File Path (pathToResolve)

This attribute contains a path to the TSV file that you wish to analyze. The path may contain references to path variables that will be automatically resolved before MetaR attempts to load table information from the path. Path variable references can be written as $a.b.c/data/file.tsv. Such a reference will require you to define the a.b.c path variable name and associate it with a value on each machine where the table will be used. You can define path variables with the Preferences/Settings MPS menu (MPS 〉 Preferences 〉 PathVariables on Mac, MPS 〉 Settings 〉 PathVariables on PC).

Columns

Columns is an attribute that presents the list of columns identified in the TSV file. Each column has a name, a type, and may be annotated with a set of column groups (see Section 2.4 for details about column groups). MetaR supports the following column types, which map to the R data types:

1. **Numeric**. Any number. Technically, can be a floating number or an integer.
2. **String**. A string of characters.
3. **Boolean**. A type that can only have two values: true or false.
4. **Category**. A type that can take only a limited number of values (e.g., {RED, GREEN, BLUE} would be a category with these values, (RED, GREEN and BLUE in our example).

These types are automatically determined from the data in the table file. However, in case the automatic algorithm failed for a table, you can change the types

Figure 2.1: **New Table.** This figure presents a freshly created Table AST Root node. You can use the button located to the right, after the <no name> red label (...), to open a file selection dialog. Use this dialog to locate a TSV file to configure this table.

manually. To do this, put the cursor over the name of the type in the column section, and use auto-completion in the inspector to change to the desired type.

Figure 2.2 presents a MetaR table annotated with groups.

Column Groups Container   2.3

Column Groups Containers are used to define column groups and annotate these groups with group usages. To create a new Column Group Container, right-click on a model and select New ⟩ o.c.tables.ColumnGroupContainer . This will create the empty container shown in Figure 2.3. You need and should have only one container per model. The container will hold the groups and group usages that you need across the MetaR Tables defined in the model.

Column Groups   2.4

Column Groups can be defined by pressing ⏎ either on top of the « ... » (immediately below Define Groups:), when the list of groups is empty, or with the cursor immediately before the name of an existing group. Each group has a name and an optional set of group usages. Figure 2.4 presents a new

```
Table GSE59364_DC_all.csv …
File Path
  ${org.campagnelab.metaR.home}/data/GSE59364_DC_all.csv
Columns
  gene: string [ ID ]
  mRNA len: numeric [ << ... >> ]
  genomic span: numeric [ << ... >> ]
  DC_normal: numeric [ << ... >> ]
  DC_treated: numeric [ << ... >> ]
  DC0904: numeric [ LPS=NO, counts ]
  DC0907: numeric [ LPS=NO, counts ]
  DCLPS0910: numeric [ LPS=NO, counts ]
  DCLPS0913: numeric [ LPS=NO, counts ]
  A_DC: numeric [ LPS=NO, counts ]
  A_DC_LPS: numeric [ LPS=YES, counts ]
  B_DC: numeric [ LPS=NO, counts ]
  B_DC_LPS: numeric [ LPS=YES, counts ]
  C_DC: numeric [ LPS=NO, counts ]
  C_DC_LPS: numeric [ LPS=YES, counts ]
  C2DC: numeric [ LPS=NO, counts ]
  C2DCLPS: numeric [ LPS=YES, counts ]
  C3DC: numeric [ LPS=NO, counts ]
  C3DCLPS: numeric [ LPS=YES, counts ]
```

Figure 2.2: **Example Table.** The figure presents an example table annotated with groups.

Column Groups and Usages

Define Usages:
  << ... >>

Define Groups:
  << ... >>

<no name> **used for << ... >>**

column group (group for short).

name

The name of the group is a string that should mean something in the context of your analysis. Some MetaR analysis statements require specific group names to be defined in a model container and referenced in an input table. Other groups will be created by you with meaningful names to help identify columns that have certain properties, so that you can refer to them collectively by the group name.

used for

Column groups can be annotated with a set of group usages. You can enter references to usages already defined in the column group container following the used for keyword shown in Figure 2.4. Press ⌐ on the « ... » to insert

the first group usage. Make sure the usage is defined (its name should be visible in the `Define Usages:` section of the container (see Section 2.5 to lean how to create a new Group Usage). You can bind a group usage reference to a Group usage by using auto-completion ($\boxed{\text{Ctrl}}$+$\boxed{}$ to locate the name, then pressing $\boxed{\pi}$ to accept one completion choice), or by typing the name of the group usage directly (note that group usage names are case sensitive).

> (R) Pressing $\boxed{\leftarrow}$ before `<no name>` will not have the desired effect if you have not yet named a group. Make sure you name groups immediately after you create them.

Column Group Usage   2.5

A Column Group Usage can be defined by pressing $\boxed{\leftarrow}$ either on top of the « ... » (immediately below `Define Usages:`), when the list of group usages is empty, or with the cursor immediately before the name of an existing group usage. When the list already contains one or more group usages, you can add more by placing the cursor over a group usage name and pressing $\boxed{\pi}$. Keep pressing $\boxed{\pi}$ to add more empty group usages.

Example Column Group Container   2.6

Figure 2.5 presents an example of a configured Column Group Container. This container defines two groups `LPS=yes` and `LPS=no`, which can be used to annotate Table columns that contain data about gene expression of samples treated with LPS or not. The group usage `LPS_Treatment` is associated to both groups to indicate that they belong together and are two kinds of treatment.

Column Groups from a Table   2.7

An alternative way to automatically create Column Groups and attach them to Columns at the same time is to use a so-called `annotation table`. These

Figure 2.5: **Example Group Container.** This example presents a container with several groups and group usages.

```
           Column Groups and Usages
Define Usages:
  LPS_Treatment
  ID
  heatmap

Define Groups:
  LPS=YES used for LPS_Treatment heatm
  LPS=NO used for LPS_Treatment heatma
  ID used for ID
  counts used for heatmap
```

tables are normal Table nodes (created as specified 2.2) with a special structure and content. When created, MetaR is able to recognize them and allows the user to use these tables to annotate other tables.

The structure of an annotation table requires the following two columns:

- *SampleID*: values of this column are sample names
- *Groups*: values of this column are comma-separated lists of group names

When annotation tables are available in the current model, the user can

1. open the Table to annotate
2. use the "Use Column Groups From Table: <table name>" intention (💡) when the cursor is on top of the Table node

This way, Sample IDs are matched with the column names and the groups listed in the same row are attached to the corresponding Column. In addition, if the groups are not defined in the ColumnGroupContainer, they are created in the annotation process. The ColumnGroupContainer is also created if it does not exist.

To see how this intention works in practice, we will create the same groups shown in Figure 2.2 using an annotation table. To do that, we firstly need to create a TSV file with the content shown in Figure 2.6. The next step is to load this table in a Table node (Figure 2.7). Finally, we invoke the "Use Column

Table	
SampleID	Groups
gene	ID
DC0904	LPS=NO,counts
DC0907	LPS=NO,counts
DCLPS0910	LPS=NO,counts
DCLPS0913	LPS=NO,counts
A_DC	LPS=NO,counts
A_DC_LPS	LPS=YES,counts
B_DC	LPS=NO,counts
B_DC_LPS	LPS=YES,counts
C_DC	LPS=NO,counts
C_DC_LPS	LPS=YES,counts
C2DC	LPS=NO,counts
C2DCLPS	LPS=YES,counts
C3DC	LPS=NO,counts
C3DCLPS	LPS=YES,counts

Figure 2.6: **Content of a Sample Annotation Table** The SampleID column provides the name of the column to which the groups should be assigned. The Groups column provides the names of the groups, separated by a comma, which will be assigned to the column identified by SampleID.

Groups From Table: AnnotationTableForGSE59364_DC_all.cvs" intention as shown in Figure 2.8 to apply the group names from the table to the destination Table. These steps will result in exactly the same annotated table shown in Figure 2.2.

```
Table AnnotationTableForGSE59364_DC_all.cvs ...
File Path
  ${org.campagnelab.metaR.home}/data/AnnotationTableForGSE59364_DC_all.cvs
Columns
  SampleID: string
  Groups: string
```

Figure 2.7: **Table with Samples and Groups.** An annotation table loaded as Table node.

Figure 2.8: **Intention to Annotate a Table using another Table**

Column Group Annotations   2.8

Sometimes, a covariate is a continuous variable that can take different values across samples. This kind of covariate can also be used to estimate linear models (see Chapter 5 and 6).

MetaR 1.3 supports continous covariates. Instead of creating a Column Group for each possible value of the covariate, you can indicate that values of the column group are determined using a special type of table called *Covariate Table*. A covariate table is a normal Table node in metaR with the following characteristics:

1. a column annotated with a group called "sample-key",
2. values of this *sample-key* column must match the column/sample names of the primary table,
3. one or more additional columns from which covariate values are extracted.

Figure 2.9 shows a simple covariate table with two columns: the SampleName column is marked as the *sample-key*. The Age column provides the age covariate values, for each sample identified by SampleName.

```
Table AgeAnnotation.tsv…
File Path
  ${org.campagnelab.metaR.home}/data/AgeAnnotation.t
Columns
  SampleName: string [ sample-key ]
  Age: numeric
```

Figure 2.9: **Sample Covariate Table.** A covariate table must have a column annotated with the sample-key group, whose values are the sample names of the table that will be annotated with the Covariate table. The other columns hold the values that will be associated to each sample when they are selected.

A covariate table can be associated to a column group by pressing ⎡ ⎤ at the end of the group in the editor or by using the intention shown in Figure 2.10.

Figure 2.10: **Intention to Add a Covariate Table** Use this intention when the values of the Age covariate are provided by in a secondary table: the covariate table.

Column Groups and Usages

Define Usages:
  LPS
  ID
  heatmap
  ignore
  age

Define Groups:
  LPS=YES **used for** LPS heatmap
  LPS=NO **used for** LPS heatmap
  ID **used for** ID
  sample-key **used for** << ... >>
  counts **used for** << ... >>
  age **used for** age

              Intentions
    📝 Load Values from a Table ▶

The annotation added to the group allows to refer to a covariate table in the current model and then select a column from that table with the covariate values (Figure 2.11).

counts **used for** << ... >>
age **used for** age [**read values from**   **use covariate** <no useCovariate>]
ignore **used for** << ... >>
          T AgeAnnotation.tsv Table (Demo_with_continous_data)
          T SimulatedData.tsv Table (Demo_with_continous_data)

Figure 2.11: **Column Group Annotation** Using auto-completion ($\boxed{\text{ctrl}}$ + $\boxed{}$ in the no table field, all the tables matching the covariate requirements are proposed. Once selected, a similar action can be used in the no useCovariate field to select a column from that table. Values from that column are matched with the samples when a statement needs to build a linear model.
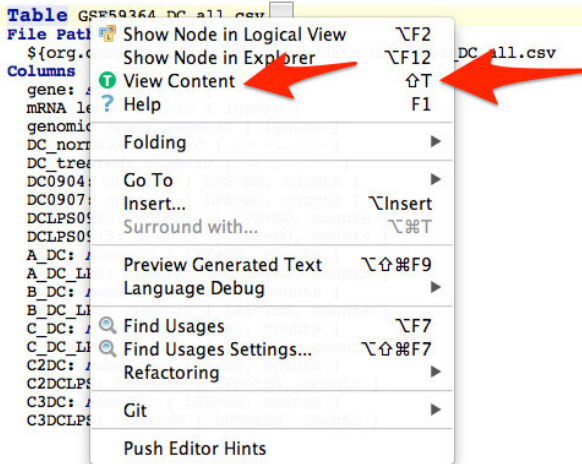
Figure 2.12: **How to activate the Table Viewer Tool**. The Table Viewer Tool can be activated by right-clicking on the Table and selecting "View Content" from the menu. An alternative way is to put the cursor on the Table node and pressing ⇧ + T .

Table Viewer Tool   2.9

(R) The TableViewer Tool was introduced in MetaR version 1.3.

The content of a Table can be inspected with the Table Viewer Tool distributed with MetaR. The tool adds to the MPS interface the capabilities to load the table's content and show it in a graphical context. Wherever a table name appears (in a Table node or in an Analysis script, see Chapter 3), the tool can be opened to see the rows and columns of that table along with their values. Figure 2.12 shows how to activate the tool on a Table node. The first time it is activated, the tool appears at the bottom of the MPS window, as shown in Figure 2.13.

The tool is immediately available for those tables directly loaded from the file system (e.g. Table nodes) and their references. Other Tables become visible after an Analysis script has been run and the content of the table has been created. In this latter case, the tool is available after the first script execution

Figure 2.13: **The Table Viewer Tool in the MPS UI.**

Figure 2.14:
Visualization Options for Table Viewer Tool. Different visualization options are available for the Table Viewer Tool. By right clicking on the tool name in the Tool Buttons bar (if not visible, select View ⟩ Tool Buttons from the MPS menu), a user can explore them and choose the one that best fits a project needs. In our experience, the floating mode or the Docked/Pinned mode with the panel on the right are very effective.

Figure 2.15: **The Table Viewer Tool**. The tool displays the content of the selected table. At the top, you can see the name of the table and the number of records loaded. Columns can be resized and rows scrolled for a full visualization of their values.

(see Chapter 3). Note that in this case, if the structure and/or content of a table changes, the Analysis script must be executed again before the tool will show the latest table data.

3 — Analyses

The MetaR Analysis Root Node 3.1

MetaR analyses are represented with an Analysis AST Root node. An analysis often imports one or more tables, performs data transformations and writes a table or generates some plots. You can create as many Analysis root nodes as needed in a model. You create an analysis by right-clicking on a model and selecting New ⟩ o.c.metar.tables ⟩ Analysis. Analysis can exist as direct child of a model, and for this reason is called a root node. Figure 3.1 presents a new Analysis root node.

name

Each analysis has a name. You should name the analysis after creating it. The name you enter will be shown in the Project Tab after the **A** icon.

statements

An analysis contains a list of statements. You can enter new statements by typing between the curly brackets { }.

```
Analysis <no name>
{
  << ... >>
}
```

Figure 3.1: **New MetaR Analysis Root Node.** This figure presents a freshly created MetaR analysis root node. You can press ⏎ over the « ... » to add Statement nodes to the analysis. Use auto-completion to discover which types of statements are available.

To enter a specific statement, you can type the alias of the statement (for instance import table). When you have typed a complete alias, the node will be inserted in place of the alias. Note that there is no actual indication that the text you are typing matches a valid alias. You need to finish typing the full alias before the node is substituted for the alias. The text you type will remain red until the substitution occurs even if the text is matching a valid alias. See Figure 3.3.

The following sections describe the kinds of statements offered by the MetaR org.campagnelab.MetaR.tables language. The simplest way to learn which statements are supported by the release of MetaR that you are using is to use auto-completion. Figure 3.2 provides a snapshot of the auto-completion menu when looking for statements to insert in the Analysis statement list.

Styles 3.2

MetaR comes with a styles language that allows to customize the graphical aspects of plots and files generated by statements. A Style is represented by a root node in the model with a given name.

Figure 3.2: **Auto-completion Dialog for Statements.** This dialog is obtained by placing the cursor where a Statement is valid, and pressing `ctrl`+ ⎵.

Figure 3.3: **Typing Statement Aliases.** The user has typed "import" on the third line. This text is a prefix of the import table statement, but is shown in red because the alias is not yet complete. The fourth line shows the import statement which is substituted to the text when the user has just finished typing "import table", the complete alias of the statement. Note that pressing ⏎ immediately after import will create the node if the prefix "import" matches a single type of node in the languages imported in this model.

To create a Style, right-click on a model in the project tab and select right-click ⟩ New ⟩ o.campagnelab.styles ⟩ Style . This will create an empty Style, as shown in Figure 3.4.

```
Style  <no name>  extends  <no extends> {
  << ... >>
}
```

Figure 3.4: **New Style.** A newly created Style AST Root node.

Style nodes are identified by a 🟢 icon. A Style is essentially a container for *style items*. Items are well-defined settings that are applied by MetaR during the rendering of a graphical object. Figure 3.5 shows a snapshot of the auto-completion menu when looking for items to add to a style. All possible style items are shown because this style is not yet bound to a specific statement.

Figure 3.5: **Adding Style Items to a Style.** A Style is a container for Style Items. By using auto-completion ctrl + ⎵ , style items can be added to the node.

Styles can extend other styles, which provides an ability to modularize appearance. This is useful if you need to build collections of plots, and most plots have a subset of style attributes, but the title of each plot or the X or Y variable changes. You would define the common style attributes in a style, and extend this style each time you need to specialize the style.

Binding Styles to Statements   3.2.1

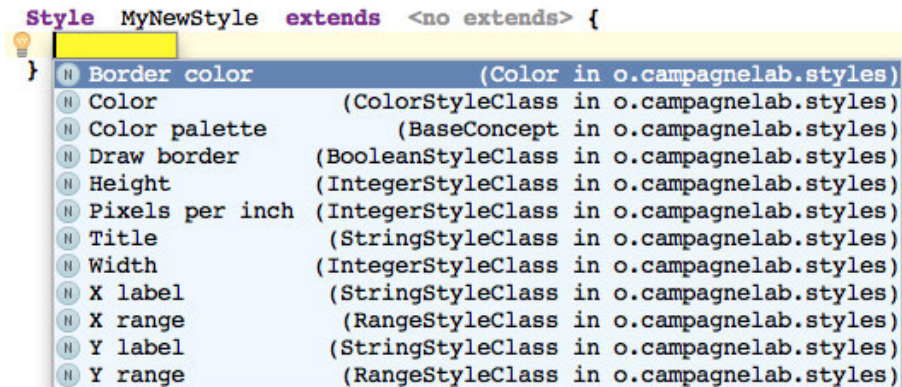In order to use a Style, you have to bind a statement to it. There are different approaches you can follow to create this binding and the choice depends on your needs and flexibility.

The simplest way to create a style and bind it to a statement is to use the intention "Create New Style" available on compatible statements as shown in Figure 3.6. This intention will create a new style and bind it to the statement. This has the advantage that the style items are immediately restricted to only the items compatible with the statement.

Figure 3.6: **Create New Style Intention on Statements.** The user has pressed [option] + [↵] on a statement that can be bound to a style. By selecting the **Create New Style** intention, a new default style is created in the model and bound to the statement.

Once bound to a statement (or more than one), the list of items that can be added to the Style is restricted to the ones compatible with the statement(s). For instance, if the style shown in Figure 3.5 is bound to a statement that uses only a Color Palette and some kind of border drawing, the auto-completion menu will appear as shown in Figure 3.7.

However, styles can be created independently from statements and bound

Figure 3.7: **Style with Restricted Items.** A subset of Style Items is displayed in the auto-completion menu when the Style is bound to a statement.

at a later time. In this case, the list of styles visible to a statement in the auto-completion menu is limited to the compatible styles available (see Figure 3.8). A style is defined compatible with a statement if its items or a subset of them are used by the statement.

Figure 3.8: **Styles visible from a Statement.** A statement can be bound only to Styles that define items compatible with the statement.

A library of Styles could be created and reused across multiple Statements, Analyses or even Models. The capability to extends other Styles promotes reuse, reduces redundancies and allows to keep the number of Styles in a model at a minimum.

Working with Tables 3.3

Import Table 3.3.1

The import table statement makes it possible to import a table, defined in the model, into the analysis. The columns of the imported table will become visible to the statements that follow the import. You can create an import table

Figure 3.9: **New Write Statement.** Use this statement to write the content of a table to a file.

write <ins><no table></ins> **to "**<no path>**"** …

statement by typing the alias import table on an empty line of Analysis. Once you have bound the reference to a table, the import statement will look like this: **import table** **SomeData**. The name in green is the name of the table the statement imports. See Section2.2 to learn how to create a Table Root node.

table

The table attribute is a reference to an AST root node. You can set this reference by typing the name of the table you wish to import, or by using auto-completion ctrl + ☐ to locate the table AST root node.

Write Table   3.3.2

MetaR analyses can create new tables. You can write the data in these tables to a file using the write statement (see Figure 3.9 for a new write statement). The write statement has two attributes: table and filename.

table

This reference should be set to the table that you want to write to a file.

output

The output should be set to a filename where you want the data contained in the table to be written. Notice that output has a button to let you select the output filename.

Identify a Set of Columns

Several types of statements require the user to select one or more columns of a table. MetaR provides several ways to select a set of columns.

- You can use the columns node to identify a set of columns.
- You can use the group node to name a single group, and identify all columns annotated with this group.
- You can use the groups node to name a set of groups, and identify all columns annotated with any of these groups.

Each strategy identifies a set of columns, which may contain one or more columns.

Define Sets of Ids

You can use this statement (alias define Sets of Ids) to define a custom set of ids. Figure 3.10 shows a new define Sets of Ids statement. Sets of ids can be used draw Venn diagrams (see Section 3.9.5), or to select rows of tables with the subset statement (see Section 3.5). To create a new set, use the define statement and choose set of ids with auto-completion.

name

The set of ids must have a name. The name is used to refer to this set of ids in other parts of the analysis.

ids

By pressing ⏎ with the cursor over the « ... » attribute you can add a new id in your set. To add another id, you just have to press again ⏎. Figure 3.11 shows an example of Sets of Ids.

Figure 3.10: **New Set of Ids Statement.** Use this statement to define a custom set of ids.

```
define Set of IDs <no name> {
  << ... >>
}
```

Figure 3.11: **Example of a user defined Sets of Ids.** The set contains 14 elements with IDS named a-d7.

```
define Set of IDs data1 {
  a b c d e f g h i j k d5 d6 d7
}
```

Subset Rows   3.5

You can use this statement (alias subset rows) to filter a table and produce a new table with a subset of the rows of the input table. Figure 3.12 shows a new subset rows statement.

Figure 3.12: **New Subset Rows Statement.** Use this statement to filter the rows of a table.

```
subset rows <no table> filter how?    -> subset
```

```
{
  import table GSE59364_DC_all.csv
  define Set of IDs GeneList {
    MAPK PSEN1
  }
  subset rows GSE59364_DC_all.csv with IDs keep rows matching any ID in GeneList -> for geneslist

  subset rows GSE59364_DC_all.csv when true: $(genomic span ) > 20000 -> genomic span>2000
}
```

Figure 3.13: **Subset Rows Examples.** This figure illustrates the two alternatives available for filtering rows: by expression (with when true:, or with a set of ID values).

table

The input table reference must be set to a table. The table must be either imported or produced by another statement.

filter

The filter determines which rows are kept. Use auto-completion to select one of the alternatives:

1. `when true:` will keep a row when the boolean expression following `when true:` evaluates to `true`.
2. `with IDS` will keep a row when the value of the column marked with group `ID` exists in the list provided as an argument. See the `define` statement to create a list of IDs in Section 3.4.

subset

The output table is called subset by default. Feel free to rename this table to better match the data in it.

Example   3.5.1

Figure 3.14 presents examples of subset row statements.

```
{
  import table GSE59364_DC_all.csv
  define Set of IDs GeneList {
    MAPK PSEN1
  }
  subset rows GSE59364_DC_all.csv with IDs keep rows matching any ID in GeneList -> for geneslist

  subset rows GSE59364_DC_all.csv when true: $(genomic span) > 20000 -> genomic span>2000
}
```

Figure 3.14: **Subset Rows Examples.** This figure illustrates the two alternatives available for filtering rows: by expression (with `when true:`, or with a set of ID values).

Boolean Expressions

One form of the subset rows statement accepts a boolean expression to determine which rows to keep. Boolean expression have one of two values: true or false, and can be constructed by comparing values with an operator. The following operators are supported by MetaR:

- **$(column)** The value operator evaluates to the value of the column in the row currently considered.
- expr1 **==** expr2 true when expr1 evaluates to the same value as expr2.
- expr1 **!=** expr2 true when expr1 does not evaluate to the same value as expr2.
- expr1 **|** expr2 true when expr1 is true or expr2 is true (boolean or. either one of expr1 or expr2 needs to be true for the result to be true).
- expr1 **&** expr2 true when expr1 is true and expr2 is true (boolean and, both expre1 and epr2 must be true for the result to be true).
- expr1 **<** expr2 true when expr1 evaluates to a value less than expr2.
- expr1 **>** expr2 true when expr1 evaluates to a value greater than expr2.
- expr1 **<=** expr2 true when expr1 evaluates to a value less or equal than expr2.
- expr1 **>=** expr2 true when expr1 evaluates to a value greater or equal than expr2.

> **R** MetaR infers the type of the expressions that you enter and will report type errors if the value of some values are not compatible with the test that your perform.

Join Tables

When you perform data analysis, you often need to combine data from different tables into one. This operation is called table joining. MetaR provides a powerful `join` statement that helps join an arbitrary number of tables. The alias of the

```
join ( <no table> ) by _____ -> <no name>
```

Figure 3.15: **New Join Statement.** Use this statement to join two or more tables into one.

statement is join. Figure 3.15 presents a newly created join statement. Join has three attributes: input tables, by and output table.

input tables

Use this attribute to enter references to tables you need to join. You can press ⏎ with the cursor inside the parentheses to create references to additional input tables (either imported or created in the analysis up to that point).

by

Use the by attribute to specify how to join the input tables. Joining tables requires knowing which rows of the input tables go together. This is done by identifying a set of columns whose values must match across rows of the input tables that go together. See Section 3.3.3 to learn how to select groups of columns in MetaR. You can select one of the following joining strategies as value of the by attribute:

- **by columns:** The same column name must be defined in all the input tables.
- **by group:** If each table has exactly one column annotated with this group, the names of the columns do not need to be the same. Otherwise, if more than one columns is in the group, their names must match across all the tables.
- **by groups:** Similar to group, but with any (>=1) number of groups. Press ⏎ to insert new group references.

The group by columns are used to calculate a list of values. When rows of the input tables have the same group by values, they are joined in a single row, and the result put in the destination table.

result

The output table is called Results by default. Feel free to rename this table to match the content of the destination table. Open the 👤 2: Inspector to see which columns will result from the join. Notice that the column groups are transferred to the columns of the output table.

How Join works   3.6.1

The output table produced by the join statement can be very different, depending on the strategy selected to join the input tables and their structure. The statement takes also care to produce unique columns from columns common to all the input tables but not used in the joining strategy. To explain how join works we will apply two different strategies to the input tables (named A and B) presented in Figure 3.16 and check the structure and content of the output table.

Table A		
Gene	Column1[ID]	Column2
Ge1	Val1	34
Ge2	Val2	452
Ge3	Val3	113

Table B			
Gene	Column2	Column3	Column4[ID]
Ge4	1	78	Val1
Ge2	3	13	Val3
Ge1	4	44	Val4

Figure 3.16: **Sample Input Tables for Join Statement.**

Do note that the two tables have columns in common (Gene and Column2) and that each table has a column annotated with a group named ID (Column1 in table A, Column4 in table B).

join by column

As first strategy, we join A and B by the column Gene. The results table generated by `join` is shown in Figure 3.17.

Table Results
(by Column Gene)

Gene	Column1	Column2.A	Column2.B	Column3	Column4
Ge1	Val1	34	4	44	Val4
Ge2	Val2	452	3	13	Val3

Figure 3.17: **Results Table for Join using by Column Strategy.**

If we analyze the structure of the table, we note that:

- there is only one Gene column (the one used for joining)
- Column2, that was present in both the input tables, was split in two columns named Column2.A and Column2.B
- the rest of the columns are the same coming from their respective input tables

Looking at the table's content, we note that:

- rows from input tables with a matching value in the Gene column were merged into a single row.
- all the other columns keep their original value from the source table with respect to the Gene column's value

join by group

Our second strategy is to join A and B by group ID. As shown in Figure 3.16, there are two columns, one in each input table, annotated with such a group and they have different names. The `join` statement is able to work even in this case by matching the values of these two columns. The results table generated by Join is shown in Figure 3.18. Again, let's go over the structure of the table:

- Gene, that was present in both the input tables but this time not used as joining column, was split in two columns named Gene.A and Gene.B

- Column2, that was present in both the input table, was again split in two columns named Column2.A and Column2.B
- the rest of the columns are the same coming from their respective input tables

Table Results
(by Group ID)

Column1	Gene.A	Gene.B	Column2.A	Column2.B	Column3	Column4
Val1	Ge1	Ge4	34	1	78	Val1
Val3	Ge3	Ge2	113	3	13	Val3

Figure 3.18: **Results Table for Join using by Group Strategy.**

Looking at the table's content, we note that:
- rows from input tables with a matching value in Column1 from A and Column4 from B were merged into a single row
- all the other columns keep their original value from the source table with respect to their matching column's value

Example Join   3.6.2

```
{
  import table GSE59364_DC_all.csv
  import table another Table
  join ( GSE59364_DC_all.csv , another Table ) by group ID -> Results
}
```

Figure 3.19: **Example of Join Statement.**

Figure 3.19 presents an example of join statement. The statement joins two tables and creates the Result table. If you open the `2: Inspector`, you will see a preview of the columns for the result table (shown in Figure 3.20).

```
path= /Users/fac2003/R_RESULTS/table_Results_0.tsv  name: Results table.name Results groups= LPS=NO,counts,LPS=YES,ID
Columns (38) :
  C3DC.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
  C3DC.another Table : numeric [ LPS=NO, counts ]
  C2DC.another Table : numeric [ LPS=NO, counts ]
  DC0904.another Table : numeric [ LPS=NO, counts ]
  mRNA len.another Table : numeric [ << ... >> ]
  B_DC.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
  mRNA len.GSE59364_DC_all.csv : numeric [ << ... >> ]
  B_DC.another Table : numeric [ LPS=NO, counts ]
  A_DC.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
  DCLPS0913.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
  DC0904.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
  C2DC.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
  C_DC_LPS.GSE59364_DC_all.csv : numeric [ LPS=YES, counts ]
  DCLPS0910.another Table : numeric [ LPS=NO, counts ]
  B_DC_LPS.another Table : numeric [ LPS=YES, counts ]
  DC_treated.another Table : numeric [ << ... >> ]
  A_DC_LPS.another Table : numeric [ LPS=YES, counts ]
  gene.GSE59364_DC_all.csv : string [ ID ]
  DC_normal.another Table : numeric [ << ... >> ]
  C3DCLPS.GSE59364_DC_all.csv : numeric [ LPS=YES, counts ]
  C_DC_LPS.another Table : numeric [ LPS=YES, counts ]
  B_DC_LPS.GSE59364_DC_all.csv : numeric [ LPS=YES, counts ]
  C2DCLPS.GSE59364_DC_all.csv : numeric [ LPS=YES, counts ]
  DCLPS0910.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
  A_DC_LPS.GSE59364_DC_all.csv : numeric [ LPS=YES, counts ]
  DC_normal.GSE59364_DC_all.csv : numeric [ << ... >> ]
  C3DCLPS.another Table : numeric [ LPS=YES, counts ]
  C_DC.another Table : numeric [ LPS=NO, counts ]
  genomic span.GSE59364_DC_all.csv : numeric [ << ... >> ]
  C_DC.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
  C2DCLPS.another Table : numeric [ LPS=YES, counts ]
  genomic span.another Table : numeric [ << ... >> ]
  A_DC.another Table : numeric [ LPS=NO, counts ]
  DC_treated.GSE59364_DC_all.csv : numeric [ << ... >> ]
  gene.another Table : string [ ID ]
  DCLPS0913.another Table : numeric [ LPS=NO, counts ]
  DC0907.another Table : numeric [ LPS=NO, counts ]
  DC0907.GSE59364_DC_all.csv : numeric [ LPS=NO, counts ]
```

Figure 3.20: **Column Preview for Result Table.** Notice that all these columns were shared across the input tables because their name follows the pattern "colName.tableName".

The statement transform table (alias `transform`) allows to transform columns of an input table and produce a new table. Figure 3.21 shows a new `transform` statement.

```
transform table<no table> -> transformedTable {
}
```

Figure 3.21: **New Transform Table Statement.** Use this statement to transform a table by applying column operations.

table
Use this attribute to select a table to transform. You can press `ctrl`+`  ` to open the autocompletion menu and display all available tables.

operations
A set of table operations, which can be used to transform the input table. See Figure 3.22 for a list of operations available to transform a table. Operations include:

drop column   This operation allows to delete a specific column. Press `ctrl`+`  ` to open the autocompletion menu and display all available columns. Then press `↵` to select one column. This column will be removed/dropped in the result table.

drop columns which match   This operation allows to delete columns which contains a specific pattern. Type the pattern after the `match` keyword.

drop columns which have group   This operation allow to delete columns which have a specific group. To do so, press first `ctrl`+`space` to display the

```
transform table GSE59364_DC_all.csv -> transformedTable {
  drop column A_DC
  drop columns which match span
  drop column which have group LPS=YES
  rename column:A_DC -> newName
}
```

Figure 3.22: **Example of Transform Table Statement.** This example show various table operations available with transform table.

available group and ⏎ to select one group.

add column This operations allows to add a column in the result table. Start by entering the name of the new column. The second argument must contain the expression that will calculate the value that the new column will take for each row of the result table. You may reference columns of the input table in the expression using the $ column selection helper.

rename this operation allows to rename a specific column from the input table. Once you have selected your column of interest, you just need to type a new name under the *(newName)* argument.

result
The output table is called TransformedTable by default, but can be renamed as needed. Use the inspector to see which columns will be produced when the transform statement is executed.

Block with Selected Tables   3.8

Sometimes it can happen that many tables are created during your analysis and, at one point, you might want to work with a restricted set of tables. The statement `with tables` allows you to select specific tables defined in your analysis and execute metaR statements on them. Figure 3.23 presents a newly

created `with tables` statement. All tables created inside the block will be available outside the block. The statement has two attributes: the list of input tables and a statement list. Figure 3.24 shows an example of the `with tables` statement.

input tables

Use this attribute to select the tables you want to see inside the block. To do so, you can use auto-completion and select tables defined in your analysis up to that point. The statements included in the statement list will only see the tables defined in this list.

statement list

Here you can use all statements defined in MetaR. Any tables or plots created in the statement list will be visible after the block ends.

```
with tables ( << ... >> ) for statement:
{
  << ... >>
}
```

Figure 3.23: **New With Tables Statement.** Use this statement to work with a restricted set of tables.

```
with tables ( GSE59364_DC_all.csv filtered ) for statements
{
  subset rows filtered with IDs keep rows matching any ID in P-value -> subset
  join ( subset, GSE59364_DC_all.csv) by group ID -> newjoin
}
```

Figure 3.24: **Example of With Tables Statement.**

```
boxplot with <no col> -> <no name> no style
```

Figure 3.25: **New Box-plot Statement.**

Plotting Data 3.9

MetaR provides simple plotting capabilities[1]. The following types of plots are currently supported:

- boxplot alias `boxplot`
- histogram alias `histogram`
- scatterplot: alias `fit`
- heatmap, alias `heatmap`
- multiplot alias `multiplot`, makes it possible to organize other plots in a matrix of n columns by n rows.

Each type of plot statement will create a plot, identified with the ≋ icon when you are trying to auto-complete a reference to a plot. Plot names are also colored with the same dark blue as the icon.

Generated plots can be customized and refined by binding each statement to a style (see 3.2). A reference to a style is back-colored with the same turquoise color as the ⑤ icon.

boxplot 3.9.1

Figure 3.25 presents a new boxplot statement.

columns

Indicate one or more columns to plot. The values of the columns will be plotted as individual boxplots in the same graph. Press $\boxed{\nwarrow}$ to define more than one column. Use auto-completion to locate individual columns from the imported tables, or the tables created by prior statements.

---

[1]These capabilities are simple, but can be extended very easily by adding new statements to draw other types of visualization. This is a key advantage of using Language Workbench technology.

plot

The attribute after `->` is a plot. Enter a name for the boxplot here. Plot names are colored blue to make them easier to recognize.

style

If a style is bound to the box plot (see section 3.2), the statement will use a `ColorPalette` item from the style to draw the plot. A Color Palette is an AST Root Node identified with the 🎨 icon.

MetaR comes with several pre-defined palettes and colors ready for use. Figure shows the auto-completion menu for a Color Palette item listing the default palettes available.

Figure 3.26: **Auto-completion menu with default Color Palettes.**

You can create a new palette by right-clicking on a model and selecting `New` > `o.campagnelab.styles` > `Color Palette`. `Colors` can be added to a palette and their order will matter in the rendering of the plot. Figure 3.27 shows a Color Palette with some color items and the auto-completion menu listing some of the default colors available.

There are two classes of colors you can use in a palette:

• **Named `Colors`.** New named `Colors` can be created by right-clicking

```
Color palette BoxplotPalette {
   Color : blue
   Color : blueviolet
   Color : gray60
   Color :
}
```

magenta	ColorValueByName	(o.c.m.a.styles.colors)
orange	ColorValueByName	(o.c.m.a.styles.colors)
pink	ColorValueByName	(o.c.m.a.styles.colors)
plum	ColorValueByName	(o.c.m.a.styles.colors)
purple	ColorValueByName	(o.c.m.a.styles.colors)
red	ColorValueByName	(o.c.m.a.styles.colors)
salmon	ColorValueByName	(o.c.m.a.styles.colors)
sienna	ColorValueByName	(o.c.m.a.styles.colors)
turquoise	ColorValueByName	(o.c.m.a.styles.colors)
violet	ColorValueByName	(o.c.m.a.styles.colors)
white	ColorValueByName	(o.c.m.a.styles.colors)
yellow	ColorValueByName	(o.c.m.a.styles.colors)
#RRGGBB		

Figure 3.27: **Auto-completion menu showing some default Colors and the ##RRGGBB option.** The auto-completion menu lists the visible Colors and #RRGGBB option. By selecting this latter, you can create a custom color based on the Red Green Blue color components (RGB).

on a model and selecting New ⟩ o.campagnelab.styles ⟩ Color . A color is identified by the 💧 icon and its name must be valid in the R language.

- **#RRGGBB Colors.** These colors can be created directly inside the Color Palette by selecting the #RRGGBB option in the auto-completion menu for a Color value. The color will be constructed from the combination of the Red, Green and Blue specified in the code.

histogram

Use this statement to plot a histogram of the values of one column.

column
Indicate one column to plot a histogram for. Use auto-completion to locate the column from the imported tables, or the tables created by prior statements.

plot
The attribute after `->` is a plot. Enter a name for the histogram here. Plot names are colored blue to make them easier to recognize.

style
The statement uses a `ColorPalette` item (see 3.9.1) from the associated style (see section 3.2) to draw the plot.

fit x by y

Use this statement (alias `fit`) to plot a scatter plot of x (one column) vs y (another column). A fit is performed, and the R2 adjusted, and P-value corresponding to the linear regression is shown on the plot. Figure 3.28 presents a new `fit` statement.

```
fit <no col> by <no col> with table <no table> -> <no name> no style
```

Figure 3.28: **New Fit X by Y.**

x, y

Indicate which columns should be plotted. Use auto-completion to locate the x and y columns from the imported tables, or the tables created by prior statements.

plot

The attribute after `->` is a plot. Enter a name for the scatterplot here. Plot names are colored blue to make them easier to recognize.

style

The fit statement uses the following items from the associated style (see section 3.2) to customize the scatterplot:

- **Title**. The main title on top of the plot.
- **X label**. A title for the x axis.
- **Y label**. A title for the y axis.
- **X range**. Range of values for the x axis.
- **Y range**. Range of values for the y axis.
- **Width**. Width in pixels of the output image with the plot.
- **Height**. Height in pixels of the output image with the plot.

heatmap   3.9.4

Heatmaps are frequently used to visualize high-throughput data [Coo+07]. Use the heatmap statement (alias heatmap) to construct a heatmap plot. Figure 3.29 presents a new heatmap statement.

```
heatmap with <no table> select data by <no dataSelection>-> <no name> no style [
]
```

Figure 3.29: **New Build Heatmap.** Notice the intention "Add Annotations" attached to the statement. You can use this intention to further customize the heatmap.

table

Specify the table that contains the data that will be used to draw the heatmap. Note that the table must meet certain conditions. An error message will be displayed if these conditions are not met:

- Some columns of the table must be annotated with at least one group whose usage is "heatmap". Such columns are used to provide data values for the heatmap. If you don't have a heatmap usage, just create one and add it to the groups you would like to include on the heatmap.
- The columns of the table must be annotated with groups and group usages to make it possible for you to use these group usages to construct a legend.

select data by

Use this attribute to customize the set of columns to plot on the heatmap. See Section 3.3.3 to learn how to select a set of columns.

plot

The <no name> attribute listed after -> makes it possible to name the plot that will hold the heatmap. Use any name you like. This name will be used to refer to the heatmap plot, for instance if you wish to assemble panels of different heatmaps into one figure.

style

The statement uses the following items from the associated style (see section 3.2) to customize the heatmap:

- **Color palette**. The colors to use in the heat map (see 3.9.1).
- **Draw border**. A boolean value to enable or disable borders in the heatmap. A value of true will draw a border. A value of false will not. By default, borders are enabled.
- **Border color**. The color to use for the border (see section 3.9.1 to check out how to create/refer to colors). If not set, the default color is *grey60*.

Figure 3.30: **Example of Heatmap Plot.** This heatmap has been customized with annotations. The LPS_Treatment group usage is shown in the legend, with two groups: LPS=yes and LPS=no. The values plotted have been scaled by rows, and the rows clustered. Data are from http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE59364.

annotations

You can use the "Attach Annotations" intention (💡) when the cursor is on top of heatmap to customize heatmap annotations. Figure 3.31 shows a heatmap statement with annotations.

```
heatmap with GSE59364_DC_all.csv select data by columns: A_DC B_DC -> heatmap HeatmapStyle [
  annotate with these groups:<< ... >>
  scale values: <no scaling>
  cluster columns: false cluster rows: false
]
```

Figure 3.31: **Build Heatmap With Annotations.** Annotations are shown after you have triggered the "Add Annotations" intention.

annotate with these groups.   You can use this attribute to select usage names that should be listed in the heatmap legend. Listing a group usage will annotate each column of data with the group that is associated to the usage.

scale values.   You can choose to scale values by column or by row. Scaling will make differences easier to see by using colors more effectively.

cluster columns: You can choose to cluster by colors or by rows by changing the boolean values accordingly.

Example

Figure 3.30 presents a heatmap constructed with the build heatmap statement, using annotations and row clustering.

Venn diagram   3.9.5

R   The Venn diagrams were introduced in MetaR version 1.3.

Venn diagrams are useful to show how many elements exist in various intersections among sets of elements [Ven80]. Use the `Venn diagram` statement (alias `venn`) to draw a Venn diagram plot. Figure 3.32 presents a new `Venn diagram` statement.

```
Venn diagram of {   set <no name> from set of ids <no oneSetOfIds> with default color   } -> <no name>
```

Figure 3.32: **New Venn Diagram.** Up to five sets can be shown on a Venn diagram. The name of each set will be shown on the Venn diagram plot.

```
Venn diagram of {   set set1 from set of ids data1 with Color : aquamarine                } -> <no name>
                    set <no name> from table <no table> when true: <expression> with default color
```

Figure 3.33: **Sets type of Venn Diagram.** Here you can see the two types of sets which can be used for a Venn diagram: a user defined set and a set from an annotated table.

name

The `name` attribute can be specified immediately after the `set` keyword. A set must have a name, which will be shown on the Venn diagram to identify this particular set.

set

Up to five sets can be shown on a Venn diagram. The data set are divided in two types:

ids from a user defined set contains only one attribute: `set of ids`. The `set of ids` attribute must refer to a user defined set. see Section 3.4 to learn how to define new sets of ids.

ids from an annotated table contains two attributes:

- `table`. This attribute must refer to a table annotated with an *"ID"* group on a column that will provide identifiers for the set elements.

- **expression**. This expression defines how to select rows of the table to extract ids for the set. The expression must return true when a row of the table is part of the set. When this is the case, the value of the column marked with the ID group is extracted and added to the elements of the set.

(R) Notice that you can turn one type of ids into the other with auto-completion. place the cursor on top of the `set` keyword and invoke `ctrl`+`SPACE` to switch the type of set. Notice that the name and color attributes, if defined, are preserved.

color

The `default color` attribute listed after **with** makes it possible to customize the color used to draw the set on the Venn diagram. By using auto-completion, a menu listing the default colors available will appear. Default colors will be used for each set where a color has not been defined.

plot

The `<no name>` attribute listed after `->` makes it possible to name the plot that will hold the Venn diagram. Use any name you like. This name will be used to refer to the Venn diagram plot.

Example

Figure 3.34 presents an example of a Venn Diagram showing three sets.

multiplot   3.9.6

This statement (alias `multiplot`) makes it possible to assemble a plot as a matrix of *m* x *n* plots. This is convenient if you would like to create a figure from panels of individual plots. In addition, multiplot provides a preview of the multi-panel plot that you can turn on and off at the click of a button. Figure 3.35 presents a new Multiplot.

Figure 3.34: **Example of Venn Diagram with Three Sets.**

```
multiplot -> <no name> [ <no numColumns> cols x <no numRows> rows ]   Preview
<<emptyTable>>
```

Figure 3.35: **New Multiplot Statement.** Click on the Preview/Hide Preview button to toggle the plot preview.

plot

The plot name is shown immediately after -> (initially <no name>). Name the mutliplot to be able to refer to it from other statements (such as render to make a PDF from it).

m cols x *n* rows

Define the number of columns and rows that you wish the multiplot to have. The product of *m* and *n* determines how many plot references must be filled in to construct the multiplot.

Preview/Hide Preview

These buttons will make it possible to preview/ hide the preview for the multiplot. Note that a preview is only available after you have run the analysis. If you don't see the image being refreshed after running the script, remember to hide the preview and show it again to refresh.

plot references

After you set the number of columns and rows, you need to link *m* x *n* references to plots that you have already constructed. Do this in table attribute (shown as «emptyTable» initially).

Multiplot Example

Figure 3.36 presents an example of multiplot.

render 3.9.7

This statement (alias render) makes it possible to save a plot or a multiplot in a local file according to a selected output format and style. Figure 3.37 presents a new Render.

plot

The <no plot> attribute listed after the alias allows you to select the plot or multiplot to render.

Figure 3.36: **Example of Multiplot.** This example shows a multiplot composed of two columns and one row. In the preview, a fit x by y plot is shown on the left, and a heatmap on the right.

Figure 3.37: **New Render Statement.**

rendering format

The rendering format is the format in which the plot will be stored on the file system. The current version of MetaR allows to use only PDF as format.

path

The <no path> attribute has to be set to a filename where you want the rendering is stored. Its extension must be compatible with the selected format (e.g. ".pdf" for PDF). If only a filename is specified

style

4 — Docker Integration

The integration with Docker helps with the reprducibility of your analyses. Installing packages with R does not make it possible to specific exactly the version of the package that you need for an analysis. While it is possible to always install the latest version of a package, the behavior of some packages will change over time. For this reason, it is useful to build snapshots of the R packages used during analysis and report the version number of the snapshot.

Pre-requisites   4.1

You will need a working docker installation on your machine before you can run MetaR analyses with Docker.

Mac OS   4.1.1

Mac users will find it convenient to download and install Kitematic (`https://kitematic.com/`). After you unzip Kitematic, move the application to /Applications and start it. Wait until the virtual machine downloads and starts.

Figure 4.1: **Download the MetaR Docker Image.** Follow steps steps to locate the MetaR Docker image.

When kitematic is ready, use the search box to locate the `rocker-metar` image (see Figure 4.1 for instructions).

Other Platforms 4.1.2

Other users should refer to documented installation steps for their platform (see `http://docs.docker.com/installation/`).

Configuring Docker 4.2

Docker integration can be configured using the MPS Preferences. Open Preferences (Setting on Windows) and locate the Docker configuration (use the search box with the docker keyword). You will be presented with the following dialog shown in Figure 4.2.

path to docker executable

This field must provide a path to a valid docker executable. On linux, try `which docker` in the shell. On Mac, if you installed Kitematic, open the application and start the terminal with the `File ⟩ Open Docker Command Line Terminal Window`. In the window, type `which docker` and copy the location to the field.

Figure 4.2: **Docker Configuration Dialog.** This dialog is available under MPS Preferences/Settings.

docker options

These are the options necessary to connect to the docker server. On Mac, if you installed Kitematic, open the application and start the terminal with the $\boxed{\text{File}}$ $\boxed{\text{Open Docker Command}}\boxed{\text{Line Terminal Window}}$. In the window, type echo 'docker-machine config' and copy the line printed to the field.

image name

This is the name of the docker image that you wish to use with MetaR. Keep the default, or enter a customized image name here. Customizing the image is useful if you need to install additional packages than the ones we use during our training sessions. If you create a new image, make sure you use fac2003/rocker-metar in the FROM field. Images that do not build on fac2003/rocker-metar are not supported at this time.

> **R** Note that you can specify an image tag/version number. Append a colon (:) and the tag after the image name. For instance, use fac2003/rocker-metar to run the rocker-metar image packaged with MetaR 1.3.1.

Figure 4.3: **Run With Docker.** Right-click on an Analysis and do Create <analysis name>. You will be presented with the Run Configuration customization dialog. Check the box to run with docker. Disable the check-box to run directly against the R installation on your computer (this behavior was the default prior for MetaR 1.3-).

> **R** Note that specifying a tag is a good idea if you need to reproduce exactly an analysis in the future. Omitting the tag will always get the latest version of the image, which may change over time.

always use docker

This checkbox can be use to force the use of Docker by default when running an Analysis. This is convenient if you know that you configuration is correct and want to run all analyses with docker.

Running with Docker   4.3

When docker is successfully configured, you can specify to use the docker image when running a MetaR analysis. See instructions in Figure 4.3 to see how to configure running insider a Docker container.

5 — EdgeR

Understanding Language Composition   5.1

The EdgeR language (`org.campagnelab.metaR.edgeR`) has been developed
as a illustration of language composition with MetaR. When you import the
`org.campagnelab.metaR` devkit into MPS, you are able to create analyses
and the statements described in the previous Chapter. If you tried to enter the
`edgeR` alias, the error shown in Figure 5.1 would appear. The reason is that by
default, MetaR does not provide an EdgeR statement.

   If you now also import the `org.campagnelab.metaR.edgeR` language
(use ⌘ + L and import the `edgeR` language), you will be able to use the
`edgeR` statement. A new EdgeR statement is shown in Figure 5.2. Adding

Figure 5.1: **Error When
Typing the EdgeR Alias.**
The `EdgeR` statement is
not yet defined.


```
edgeR
○ No variants for "edgeR"
```

the EdgeR language to MetaR contributes a new kind of Analysis statement, which becomes available through auto-completion. From a user point of view, importing languages is all that is required to extend MetaR with new language constructs. This new statement can be configured by the user and will generate R code together with the other statements. This happens seamlessly and requires no other configuration than declaring that the model uses another language.

```
edgeR counts= <no table> model: ~ 0
  comparing  ____ -> <no name> (normalize with  common dispersion estimations )
```

Figure 5.2: **New EdgeR Statement.** A new EdgeR statement created after you have added the org.campagnelab.metaR.edgeR language to the model's MPS Used Languages.

The edgeR Statement   5.2

The edgeR statement performs tests of differential expression using read counts contained in a table of data. The statement has the following attributes.

counts table
The table must contain columns annotated with the "counts" group. Bind this table reference to a table that contains non-normalized read counts.

model
You can use the model attribute to enter a linear model. EdgeR will use this model to model the mean and variance of the data. You can enter a linear model by typing + followed by the name of a group usage attached to the counts table. Repeat to add multiple factors to the model.

> (R) EdgeR will use an exact test when the model has one factor. but will use a Generalized Linear Model (GLM) when the model has more than one factor. This is handled transparently.

comparing

The comparing attribute makes it possible to define the statistics that should be tested for difference with zero. After you have defined a model with several factors (corresponding to group usage), the factor levels (corresponding to group names) will be offered for auto-completion. The factor level name stands for the average of the columns annotated with the group. See Figure 5.3 for an example.

normalize with

EdgeR supports three types of normalization methods, which estimate variance/dispersion in different ways:

- common dispersion
- trended dispersion
- tagwise dispersion

Place the cursor on the keyword following normalize with and use auto-completion to switch from one type of normalization method to another. The the EdgeR Bioconductor documentation `http://www.bioconductor.org/packages/release/bioc/html/edgeR.html` for details about these approaches.

Example 5.3

Figure 5.3 presents an example where the edgeR statement is configured with a model (one factor: LPS) to call differences between columns labeled with the groups LPS=YES and LPS=NO.

```
edgeR counts= filtered model: ~ 0 + LPS
  comparing LPS=YES - LPS=NO -> Results (normalize with  tagwise dispersion )
```

Figure 5.3: **EdgeR Example.**

6 — Limma Voom

(R) Limma Voom is a statement introduced in MetaR 1.3.

Overview   6.1

The Limma Voom statement makes it possible to use the Limma R package and the Limma Voom adjustment to analyze RNA-Seq data with Limma. Similarly to EdgeR, the language that provides the `Limma Voom` statement must be added to the model where you plan to use the statement in analysis. The name of the language is *org.campagnelab.metar.limma*. Note that this language depends on *org.campagnelab.metar.models*, which should also be imported.

The Limma Voom Statement   6.2

The `limma voom` statement performs tests of differential expression using read counts contained in a table of data. The statement has the following attributes. Figure 6.1 presents a newly created Limma Voom statement.

```
limma voom counts= <no table> model: ~ 0
  comparing        -> <no name>
```

Figure 6.1: **New Limma Voom Statement.**

counts table

The table must contain columns annotated with the "counts" group. Bind this table reference to a table that contains non-normalized read counts.

model

You can use the model attribute to enter a linear model. Limma Voom will use this model to model the mean and variance of the data. You can enter a linear model by typing **+** followed by the name of a group usage attached to the counts table. Repeat to add multiple factors to the model.

comparing

The comparing attribute makes it possible to define the statistics that should be tested for difference with zero. After you have defined a model with several factors (corresponding to group usage), the factor levels (corresponding to group names) will be offered for auto-completion. The factor level name stands for the average of the columns annotated with the group. See Figure 6.2 for an example.

adjusted counts

This attribute is exposed under the Inspector Tab(2: Inspector). It takes a boolean value: either true or false. When true, the limma voom statement will produce a table of adjusted counts. Data in this table is adjusted to remove the effect of covariates described in the model, but not used in the comparing attribute. This is useful to remove the effect of batches, or other cofactors expected to affect expression. Adjusted counts are implemented with the Limma removeBatches function.

Example 6.3

Figure 6.2 presents an example where the Limma Voom statement configured with a model (one factor: LPS) to call differences between columns labeled with the groups LPS=YES and LPS=NO.

```
limma voom counts= filtered model: ~ 0 + LPS
  comparing LPS=NO - LPS=YES -> results adjusted counts: adjusted
```

Figure 6.2: **Limma Voom Example.** This example would generate an error because the model needs more than LPS to adjust for covariates. This is implied in the use of adjusted counts.

7 — Biomart

Overview   7.1

The BioMart project develops software and data services that are made available to the international scientific community. Users of MetaR can access data marts provided by BioMart, providing access to a wide range of research data . Similarly to EdgeR and Limma, BioMart support is provided in a MetaR language extension. This means that in order to access BioMart with MetaR, you first need to add the *Biomart* language to the model where you create the analysis. To do so, you can press ctrl + L and add language *org.campagnelab .metar.biomart*. After adding the *biomart* language to the model, you can create `query biomart` statements, described in the following sections.

> **R** The Biomart language in MetaR was developed by William ER Digan
> and was introduced in MetaR 1.4.

, The query biomart statement makes it possible to interactively specify which data should be retrieved from a mart (using attributes and filters). Data downloaded from Biomart will be stored in a new table. Figure 7.1 presents a newly created query biomart statement. The query biomart statement has the following parameters.

- database This is the source database that will be queried to retrieve data.
- dataset This is the dataset, inside the database that will provide data.
- attributes These attributes describe which columns of data will be downloaded and stored in the result table.
- filters These filters control which rows of data will be retrieved from the dataset.

```
query biomart database select a database and dataset select a dataset
  get attributes << ... >>
  filters << ... >>
  -> resultFromBioMart
```

Figure 7.1: **New Query Biomart Statement.**

database

The first time you create a query biomart statement in an Analysis, the statement will retrieve the list of available Biomart databases. To use one of these databases, use auto-completion over the database attribute (see text *select a database* in Figure 7.1). Then, press Ctrl+ and select one database. Selecting the database will retrieve the set of datasets available in this database, which will become available with auto-completion.

dataset

Once a database is selected, you need to choose a dataset inside this database. To choose one, press Ctrl+ to display all available datasets, on the text *select a*

dataset. You may use type keywords to identify the dataset of interest, similarly to other uses of auto-completion in MPS and MetaR. When a dataset is selected, you will need to configure its children:

- `attributes`, which will be the columns retrieved from the dataset that will populate the result table.
- `filters`, which make it possible to restrict the rows of data to retrieve.

> **R** In a few instances, you may find that a dataset cannot be associated with attributes or filters. In this case, in the auto completion menu for both attributes and filters will display the message *"No available filter or attributes in this dataset"*. The selected dataset is no more available in Biomart. This means that this dataset, although available via the Biomart web service, cannot be used to retrieve data from the web service. You will need to select another database/dataset.

attributes

Attributes are columns of the source dataset that will be written to the result table. Figure 7.2 presents a new biomart attribute. An attribute has three parameters:

- `attribute`, a column you want to retrieve from the dataset and write to the result table. Press `ctrl`+`  ` to display the available column names.
- `type`, the value type of the attribute. Choose a type for the column that will be created from the set: `boolean`, `numeric` or `string` (note that `string` is the default). To change the type, press `ctrl`+`  `.
- `column group`, an attribute can have a group such as "ID". The user can display the autocompletion menu by pressing `ctrl`+`  `. Group must be defined in the Column Group Container to be added to columns created for the result table.

> **R** You must select at least one attribute before you can execute the `query biomart` Statement.

```
query biomart database ENSEMBL GENES 79 (SANGER UK)and dataset Mus musculus genes (GRCm38.p3
  get attributes <no attribute> of types string with column group annotation select a group
  filters << ... >>
  -> resultFromBioMart
```

Figure 7.2: **Select an attribute in a biomart dataset.** An attribute contains any information you want to retrieve to populate the result table. Attributes have a name, a type and a column group annotation. The set of available attributes depends on the specific dataset selected.

filters

Filters make it possible to restrict the result with some criteria. The types of filter available depend on the dataset selected in the statement. Figure 7.3 presents a new biomart filter. There are four kinds of filters:

- boolean filters select rows for which a value is either true or false. For example, if a gene has or does not have a miRBase identifier.
- text filters select rows which match some text in some attribute. The query will return only rows of the dataset that contain which match the text. For example, you can use a text filter to query rows that include a specific GO term.
- list filters select rows that include an element among those of a finite list. Available list elements are determined by the mart dataset. For example, a specific chromosome in a specie can be selected with a list filter.
- id list filters select rows that contain a specific set of identifiers. These ids can be obtained either in a MetaR SetOfIds node, defined before the query biomart statement, or directly from an annotated table, where one column has an ID group.

table

The future table where your result will be stored. Column annotations derived from the attribute type and column groups are displayed under the Inspector Tab

```
query biomart database ENSEMBL GENES 79 (SANGER UK)and dataset Mus musculus genes (GRCm38.p3
  get attributesEnsembl Gene ID from featureof types string with column group annotationID
  filters no filter <no filterWith>
  -> resultFromBioMart
```

Figure 7.3: **New Biomart Filter.** A filter allow you to filter rows of the dataset according to some criterion. It exist four filters categories: boolean, text, list and id list. Filters are related to a specific dataset.

(🖳 2: Inspector).

Examples   7.3

Example 1   7.3.1

Figure 7.4 shows how to obtain a table from the Ensembl database and Human dataset. The result table "resultFromBiomart" contains two columns, the Ensembl Gene and Exon ID, where the first column is annotated as a group "ID". These results are filtered to exclude any gene that does not have a miRBase identifier.

```
query biomart database ENSEMBL GENES 79 (SANGER UK)and dataset Homo sapiens genes (GRCh38.p2)
  get attributesEnsembl Gene ID from featureof types string with column group annotationID
               Ensembl Exon ID from featureof types string with column group annotation select a group
  filters Ensembl Gene ID(s) [e.g. ENSG00000139618]from a set of ids idsset
          with miRBase ID(s)where values are false
  -> resultFromBioMart
```

Figure 7.4: **Biomart Example 1.**

Example 2   7.3.2

Figure 7.5 shows how to obtain a table from the *Paramecium bibliography* database. The result table "resultFromBiomart" contains two columns, the PubMed ID and the abstract, where the publication year is equal or larger than 2000.

```
query biomart database PARAMECIUM BIBLIOGRAPHY (CNRS FRANCE) and dataset Paramecium bibliography
    get attributes PubMed ID from my_attributes of types string with column group annotationID
                    Abstract from my_attributes of types string with column group annotation select a group
    filters Year >= match 2000
    -> resultFromBioMart
```

Figure 7.5: **Biomart Example 2.**

8 — R Functions

Overview 8.1

Since version 1.4, MetaR supports calling R functions directly. This Chapter describes how you can use this feature to take advantage of the many functions available in R packages to transform data in your MetaR Analyses.

Function Stubs 8.1.1

Function stubs are provided that represent functions offered by different packages. Stubs do not provide the code associated with the function, but describe the function name and the arguments of the function and its default values. This information is used to support auto-completion for R functions.

We provide pre-imported stubs for the packages used during the MetaR training sessions, namely: *base*, *graphics*, *data.table*, *pheatmap*, *biomaRt*, *edgeR* and *limma*. While stubs are provided, they are not immediately available in a MetaR analysis. In order to use R function stubs, you need to

- Add *org.campagnelab.R* to the list of Used languages in the model where you need the stubs.

- Use the `import stubs` statement followed with the name of the package that provides the functions that you wish to use.

For instance, if you enter the following statement:

```
import stubs base
```

After typing this statement, the *base* package R functions will become available inside the Analysis where you imported these stubs. You can use R function using the `eval` statement or expression (see Sections 8.6and 8.7).

If you need a package that is not yet provided with MetaR, you should use the `import package` statement (see Section 8.3).

Import Stubs Statement 8.2

The `import stubs` statement (alias import stubs) makes it possible to import functions in packaged already packaged with MetaR. Simply type import stubs, and use auto-completion to locate the package for which you need to import function stubs.

Import Package Statement 8.3

The `import package` statement (alias import package) makes it possible to import functions in any R package. If the package is already provided in MetaR, the `import package` statement will be automatically replaced with the equivalent `import stubs` statement (see Section 8.2). Figure 8.1 presents an import package statement.

name

The name attribute is a string and must be the name of an R package, suitable to install the package in R with `install.packages("name")`.

```
import package <no name>
```

Figure 8.1: **New Import Package Statement.** Enter the name of an R package to import this package into the Analysis. Note that the package will be visible only after you run the Analysis at least once.

> (R) If you need to import a Bioconductor package, use the import bioconductor package statement instead.

After you execute the Analysis that contains the import package statement, the package will be installed in the version of R that you are using, if needed and the package loaded. Use the "Reload Functions and Create Stubs" intention after you have executed the statement to create the Stubs root node for the functions in the package. When you call this intention, the package will be inspected for function declarations, and these declarations will be written to a Stubs root node in the model where the Analysis is located. Following this process, the import package statement is replaced with the import stubs statement, loading the stubs directly from the model. Note that you can inspect the stubs object to learn about the functions available in the package represented by the stubs.

Import Bioconductor Package Statement   8.4

Importing a bioconductor package is very similar to importing a regular R package, but you need to use the import bioconductor package statement. This statement ensure appropriate installation and loading of bioconductor packages in R. Figure 8.2 presents a new import bioconductor package statement.

```
import bioconductor package   <no name>
```

Figure 8.2: **New Bioconductor Import Package Statement.** Enter the name of an Bioconductor package to import this package into the Analysis. Note that the package will be visible only after you run the Analysis at least once.

Stubs   8.5

Stubs represent functions from R regular or bioconductor packages. We do not recommend creating Stubs manually. The easiest way to create Stub root nodes is by using the `import package` and `import bioconductor package` statements. Figure 8.3 presents a snapshot showing a few functions located in the base Stubs root node (R version 3.1.3).

> **R** Stubs distributed with MetaR are packaged in models named after the version of R which provided the package, in an effort to help track differences between major R versions.


```
base ×

    stubs, for package base
        abbreviate <- function ( names.arg = no default, minlength = 4L, use.classes = TRUE, dot = FALSE, strict = FALSE, ) NULL
                                 method = c("left.kept", "both.sides")
        abs <- function ( x = no default ) NULL
        acos <- function ( x = no default ) NULL
        acosh <- function ( x = no default ) NULL
        addNA <- function ( x = no default, ifany = FALSE ) NULL
```

Figure 8.3: **Base Package Stubs Illustration.** This snapshot presents the beginning of the stubs base root node, located in model R3_1_3 (language *org .campagnelab.metar.r.stubs*).

Eval Statement   8.6

The `eval` statement can be used to call R functions that produce some side-effect. When you import the devkit *org.campagnelab.R* into a model, you can

type `eval` in an Analysis node as a statement. The statement will offer auto-completion for function names imported into the analysis. If you do not see the function that you would like to use, make sure you have imported the stubs for the package that contains this function. When you run the analysis, the function named after `eval` will be executed. Note that you cannot retrieve a return value with the `eval` statement. You must use the `eval` expression to obtain a value. The eval statement is useful when you need to call a function that has a side-effect, for instance the setkey function of data.table.

Eval Expression 8.7

The `eval` expression can be used to call R functions inside a MetaR expression. MetaR expressions are used in the `subset` statement, and in some plotting statements (e.g., `boxplot` or `histogram`). When you import the devkit *org.campagnelab.R* into a model, you can type `eval` inside an expression. The `eval` expression will offer auto-completion for function names imported into the analysis. If you do not see the function that you would like to use, make sure you have imported the stubs for the package that contains this function.

Accessing MetaR Columns within R Expressions 8.8

When you use either the eval statement or expression, you will often need to access columns from a MetaR table to pass as arguments to the function. You can do this with the $ node, which bridges between R expressions (used inside R functions) and MetaR columns. After typing $, the node will auto-complete to the set of columns visible at this point of the MetaR `Analysis`.

Example 8.9

Figure 8.4 presents an example where stubs are imported for packages pre-packaged with MetaR and the import package statement is used to import

grDevices.

```
Analysis Testing functions
{

  import stubs base
  import stubs data.table
  import stubs graphics
  import package grDevices
  simulate dataset  with  [
    num of samples:  3
    num of genes:  500
    mean when all factors are false: 1
    discrete factors:  treatment
    effect size: 1
    continuous covariate: temperature , range: [ 0 - 100 ] , slope: 1
  ] -> simulate
  eval boxplot(x = ${sample_1_treatmen t})
  eval boxplot(x = c(1, 2, 3, 4))
}
```

Figure 8.4: **Functions Example.** This example imports stubs and one package, creates a table with three columns (see Simulate statement in Chapter 9) and evaluates two R functions. The first use of the boxplot function plots the values of the column sample_1_treatment that was generated with the simulate statement. The second call to boxplot is given a list of four integers.

9 — Simulating Datasets

Why simulating datasets 9.1

Simulated datasets are useful to check that analyses work as expected. MetaR provides a `Simulate Dataset` command that allows to simulate datasets starting from a few assumptions and parameter values. This approach could be also beneficial at experimental design time to validate certain assumptions before running (expensive) experiments.

In order to use the command inside an Analysis script, the *simulation* language (`org.campagnelab.metaR.simulation`) has to be imported in the current model (use ⌘+L and select the language from the list). Figure 9.1 shows a new `simulate dataset` statement.

The Simulate Dataset Statement 9.2

The `simulate dataset` statement is configurable and lets you create datasets that reflect different simulation scenarios. The output dataset is represented by a `Table` node that can be then further manipulated with other MetaR statements.

```
simulate dataset  with  [
  num of samples:    #sample
  num of genes:    #genes
  mean when all factors are false:  <no mean>
  discrete factors:    factor name
  effect size:  effect of dicrete factors
  continuous covariate:  factor name  , range: [ lower limit - upper limit ] , slope: <no linear_slope>
] -> simulate
```

Figure 9.1: **New Simulate Dataset Statement.** A new `simulate dataset` statement created after you have added the `org.campagnelab.metaR.simulation` language to the model's MPS Used Languages.

num of samples

The number of samples included in the dataset. Each sample is named according to the results of the simulation. If the simulation decides that the sample name *sample_3* has been treated with a discrete factor named *LPS*, it is renamed to *sample_3_LPS* to make it easy to identify the simulated treatment.

num of genes

The number of genes included in the dataset. Each gene is renamed according to the results of the simulation. If the simulation decides that the gene named *gene_2* is affected by a discrete factor named *LPS*, it is renamed to *gene_2_LPS* to make it easy to identify the simulated treatment.

mean

The value of the mean expression level for each gene, assuming no treatment.

discrete factors

List of treatments used in the simulation. About 50% of the samples will be considered treated with each factor specified here. About 30% of the genes will be considered affected by each factor.

effect of discrete factors

The impact of each discrete factor on the data generated by the simulation

continuous covariate

A covariate that will affect the value of the gene expression level. You can define the age of the covariate, its range and the slope. A value is added to the expression value of each gene equal to the product of the slope and the cofactor value. Cofactors are set for each sample using a uniform distribution. For instance, if you indicate an 'age' continous covariate between 0 and 36, each sample will be assigned an age in this range, and the value added to the expression level of the gene will be determine for each sample by multiplying the age of the sample by the slope of the 'age' cofactor.

Example   9.3

```
simulate dataset  with  [
  num of samples:   10
  num of genes:   20
  mean when all factors are false:  5
  discrete factors:   LPS
  effect size: 100
  continuous covariate: age , range: [ 0 - 36 ] , slope: 10
] -> simulate
```

Figure 9.2: **Simulate Dataset Example.** This statement will create a dataset with a single discrete factor (LPS) and a covariate factor (age) with a range of 0 to 36 (for instance, this could be the mouse age in a mouse model).

```
Inspector

org.campagnelab.metar.tables.structure.FutureTable

    path= /Users/mas2182/temp/metaR_results/simulation/table_simulate_0.tsv
        ID,LPS=No,age,counts,LPS=Yes
    Columns (11) :
      gene: string [ ID ]
      sample_1: numeric [ LPS=No, age, counts ]
      sample_2: numeric [ LPS=No, age, counts ]
      sample_3_LPS: numeric [ LPS=Yes, age, counts ]
      sample_4_LPS: numeric [ LPS=Yes, age, counts ]
      sample_5: numeric [ LPS=No, age, counts ]
      sample_6: numeric [ LPS=No, age, counts ]
      sample_7_LPS: numeric [ LPS=Yes, age, counts ]
      sample_8: numeric [ LPS=No, age, counts ]
      sample_9: numeric [ LPS=No, age, counts ]
      sample_10: numeric [ LPS=No, age, counts ]
```

Figure 9.3: **Preview of the Dataset Structure as Shown in the Inspector.**

```
                              Column Groups and Usages

Define Usages:
  ID
  LPS
  age

Define Groups:
  sample-key used for << ... >>
  ID used for  ID ID ID
  LPS=Yes used for LPS
  LPS=No used for LPS
  age used for age [ read values from  CovariateForSimulateDataset_TOBBQGPXLW   use covariate  age
  counts used for << ... >>
```

Figure 9.4: **Column Group Annotations Created in the Model.**

Figure 9.5: **Covariate Table Generated with Simulate Dataset.**

Figure 9.6: **Table Generated with Simulate Dataset.** This is a partial view of the full table.

10 — Extending MetaR

Overview   10.1

Because MetaR is developed in the MPS language workbench, you can use language composition as a way to extend the MetaR language. In this Chapter, we provide a very simple example to illustrate how to extend MetaR through language composition.

Let's assume that you have just learned about the heatmap.2 function provided in the gplots R package. You wish to use this function to create heatmaps with MetaR. To achieve this, you would follow the following steps:

1. Create a new MPS Language.
2. Create a Heatmap.2 language concept in the Structure Aspect of the language (see [Cam14]).
3. Customize the Generator to transform instances of the Heatmap.2 into R code.

Create a new Language

Let's create a new language. To do this, select the project and do right-click
〉New 〉Language . Name the language something like your.domain.heatmap.
the language has been created, select its name under the Project Tab and adjust Dependencies to include *org.campagnelab.metaR.tables*. Set the Scope to
Extends (this will allow statements of this new language to extend concepts of
org.campagnelab.metaR.tables).

Create a new Language Concept

Select the Structure Aspect of the *your.domain.heatmap* language and do right-click
〉New 〉Concept .[1] Name the concept Heatmap2. Define the extends clause to
be Statement (from language *org.campagnelab.metar.tables*). The resulting
concept should appear as shown in Figure 10.1.

concept alias
Define the alias of the concept. Use heatmap2. An instance of the concept will
be created when you type this keyword in the editor.

reference to a table
To plot a heatmap, we will take data from a MetaR table. This can be achieved
by adding a TableRef child to the Heatmap2 concept. Set the cardinality to
exactly one child ([1]).

heatmap produces a plot
The heatmap2 statement will produce a plot, so you need to add a child of type
Plot. You may call this child 'plot' for simplicity. Set the cardinality to exactly
one child ([1]).
    Figure 10.2 presents the completed concept after adding alias, table and plot.

---

[1]You can create a new language in an existing project, or use the New Project Dialog to
create a Project with type "Language".

```
concept Heatmap2 extends      Statement
                 implements <none>

  instance can be root: false
  alias: heatmap2
  short description: <no short description>

  properties:
  << ... >>

  children:
  << ... >>

  references:
  << ... >>
```

Figure 10.1: **Heatmap2 Concept.** The Heatmap2 concept extends Statement.
When the language is composed with MetaR, Heatmap2 will become available
for auto-completion whenever a MetaR Statement can be used.

```
concept Heatmap2 extends      Statement
                 implements  <none>

  instance can be root:  false
  alias:  heatmap2
  short description:  <no short description>

  properties:
  << ... >>

  children:
  table : TableRef[1]
  plot  : Plot[1]

  references:
  << ... >>
```

Figure 10.2: **Complete Heatmap2 Concept.** This figure shows the completed Heatmap2 concept with an alias and children for table and plot.

Figure 10.3: **Editor of the Heatmap2 Statement.** Notice how the editor simply shows the name of the statement, delegates to the TableRef editor to render the table reference, and delegates to the Plot editor to show the plot child.

```
<default> editor for concept  Heatmap2
node cell layout:
  [- heatmap2 % table % -> % plot % -]

inspected cell layout:
  <choose cell model>
```

Define the Editor   10.4

An MPS editor customizes how a node appears in the editor. Create an editor for Heatmap2 (see [Cam14]) and define its content as shown in Figure 10.3.

Generate R Code   10.5

In order to generate the statement to R code, we will use the MPS Generator aspect.

In the first step, we create a reduction rule, see Figure 10.4 and [Cam14] (Generator Aspect chapter). The rule indicates that nodes of the Heatmap2 concept will be transformed with the reduce_Heatmap2 template. The template was created using the **New Template** intention found on the reduction rule node. See detailed instructions in [Cam14]. When configuring the type of the output node (under content node:), use Lines, from the language *org.campagnelab.textoutput* to produce text with the MPS Generator aspect.

The simplest template we can build for the Heatmap2 concept is shown on Figure 10.5. Simply calling the heatmap.2 function is a good start, but trying to run this will fail because the function is not part of a plain R distribution. The next section explains how to install the gplot R package which provides the

Figure 10.4: **Generate R Code: Step 1.** Start by adding a reduction rule.

function and activate it.

Adding package and library support   10.5.1

MetaR makes it easy to install R packages as they are needed by an analysis. For this to work, we need to declare that the Heatmap2 concept depends on the *gplots* R package. We can do this by overriding the default Statement behavior method called dependencies(). This method is expected to return the list of package names that must be installed and loaded before the statement can execute. To override the method, navigate to the Heatmap2 concept in the editor, select the behavior tab, create the behavior and when the empty behavior is shown, select Override Behavior Method (see Figure 10.6). Replace the body of the method with

```
return new singleton<string>("gplots");
```

Figure 10.7 shows the complete method. Rebuild the language, then the solution. When you run the Analysis, you will see that the gplots package is being installed:

Figure 10.5: **Simplest template.** The template simply converts any node of the Heatmap2 concept to a line of R code that calls the heatmap.2 function with the table as an argument. Notice the use of a property macro to obtain the R name of the table variable from the node table attribute. Open the 🐣2: Inspector to see what value the macro will take when a node is generated to R code.

Figure 10.6: **Override Behavior Methods.** When the override dialog appear, choose dependencies() and click OK.

```
public sequence<string> dependencies()
  overrides StatementDependencies.dependencies {
  return new singleton<string>("gplots");
}
```

Figure 10.7: **Complete Dependencies Behavior Method.**

```
...
Loading required package: gplots
Installing package into '/Users/fac2003/.metaRlibs'
(as 'lib' is unspecified)
also installing the dependencies 'bitops', 'gtools', 'gdata',
    'caTools'

trying URL ....
```

Adjust Generator Priorities   10.5.2

Adjust the generator priorities as shown in Figure 10.8. To set priorities, you need to define a Design dependency on *org.campagnelab.metar.tables* in the generator aspect Module Properties dialog.

Redirecting the plot output   10.5.3

The next problem with the simple template is that it fails to write the image of the plot in location where MetaR can find it. This is needed to display the plot preview, or to make it possible to use plots with the multiplot statement. The strategy we use to handle both cases is to wrap the actual plotting code inside

Figure 10.8: **Adjust Generator Priorities.** Generation of the Heatmap2 concept has to happen in the same generation phase as the generation of the *org.campagnelab.metar.tables* concepts. Failure to adjust priorities may result in table names not being correctly inserted in the heatmap.2 function call.

a plot_xxx function. The function is named with the id of the statement that generates the plot, so that we can easily refer to it in other places where the plot should be reused. We start by defining this function:

```
plot_ $[id]=function(t){
  heatmap.2(as.matrix(t))
}
```

The function accepts one argument: the name of the table to plot.

R The number of arguments is up to you, since you will generate both the function and the function calls.

The next step is to add the R code for generating a PNG file with the plot to show a preview in the inspector. To this end, we redirect the plot output with png(), call the plot function, and close the graphics output (dev.off()):

```
png(file=" $[plot.png]", width= $[w], height= $[h])
plot_ $id( $[table])
ignore <- dev.off()
```

Notice how the parameters of the png function are taken from the Plot node. For instance, the $[plot.png] macro will expand to

```
new RPath(node.plot.getPath()).toString();
```

Handling errors 10.5.4

Accurate error reporting is important to the end-user. When things do not go well and the R code fails, it is useful to know precisely which MetaR statement generated the error. In MetaR, this is done by taking advantage of the tryCatch R functions (see http://mazamascience.com/WorkingWithData/?p=912).

Because tryCatch is not particularly intuitive and is rather verbose, MetaR extends the *org.campagnelab.TextOutput* language with the tryForNode statement (concept name: TryAndReport). The tryForNode statement attempts to execute some lines of R code, and reports any errors that occur in these lines with a link to the statement that produced the error or warning. The tryForNode statement is also responsible for showing the STATEMENT_EXECUTED/8289224 569309332962/ lines in the Run tool, which are hyperlinked to statements in the editor.

The statement tryForNode takes an ID, which is the result of the id() method defined for each MetaR statement. The value of the ID can be set as usual with a property macro (its value should be node.id()). Figure 10.9 shows the completed template for Heatmap2. While the statement only needs to call the heatmap.2 function, handling possible errors and producing plots that can be reused to build multi-panel figures have added quite a few lines to the output.

(R) MetaR 1.3.1.1 makes it easier to wrap *TextOutput* Lines into a tryForNode

block. Select a node of type Lines (highlighted in blue braces) and invoke the intention Wrap Lines in a TryForNode Block. Note that this intention is not available for lines already inside a tryForNode block.

Figure 10.9: **Complete Generator for Heatmap2.** This template wraps the heatmap.2 function call inside a plot function (needed when building figures with multiple panels) and inside a tryForNode statement. The tryForNode statement generates to the tryCatch R language construct and appropriately reports errors to the end-user.

Using the New Language   10.6

Using the *your.domain.heatmap2* language in MetaR analyses requires adding the language under Used Languages, as shown here:

Once the language is added, you can create statements by typing the Heatmap2 concept name. Figure 10.10 shows the result of using the new concept.

Figure 10.10: **Heatmap2 Execution.** This figure presents an Analysis using the new Heatmap2 concept and shows the resulting plot preview in the inspector (bottom right). Error handling and progress indicators can be seen on the bottom left.

This concludes this tutorial. You can find a project with the *heatmap2* language extension described in this Chapter at:

```
https://bitbucket.org/campagnelaboratory/
                        metar_extension_tutorial
```

11 — MPS Key Map

Windows or Linux	MacOS	Action
`⌥`+`0-9`	`⌥`+`0-9`	Open corresponding tool window
`ctrl`+`S`	`⌘`+`S`	Save all
`ctrl`+`⌥`+`F11`	`N` or `A`	Toggle full screen mode
`ctrl`+`⇧`+`F12`	`N` or `A`	Toggle maximizing editor
`ctrl`+`BackQuote`	`ctrl`+`BackQuote`	Quick switch current scheme
`ctrl`+`⌥`+`S`	`⌘`+`Comma`	Open Settings dialog
`ctrl`+`⌥`+`C`	`⌘`+`⌥`+`C`	Model Checker

Table 11.1: General

Windows or Linux	MacOS	Action
`⌥` + `F7`	`⌥` + `F7`	Find usages
`ctrl` + `⌥` + `⇧` + `F7`	`⌘` + `⌥` + `⇧` + `F7`	Highlight cell dependencies
`ctrl` + `⇧` + `F6`	`⌘` + `⇧` + `F6`	Highlight instances
`ctrl` + `⇧` + `F7`	`⌘` + `⇧` + `F7`	Highlight usages
`ctrl` + `F`	`⌘` + `F`	Find text
`F3`	`F3`	Find next
`⇧` + `F3`	`⇧` + `F3`	Find previous

Table 11.2: Usage and Text Search

Windows or Linux	MacOS	Action
`ctrl` + `M`	`⌘` + `M`	Import model
`ctrl` + `L`	`⌘` + `L`	Import language
`ctrl` + `R`	`⌘` + `R`	Import model by root name

Table 11.3: Import

Windows or Linux	MacOS	Action
`ctrl` + ` `	`ctrl` + ` `	Code completion
`ctrl` + `⌥` + `click`	`⌘` + `⌥` + `click`	Show descriptions of error or warning at caret
`⌥` + `↵`	`⌥` + `↵`	Show intention actions
`ctrl` + `⌥` + `T`	`⌘` + `⌥` + `T`	Surround with...
`ctrl` + `X` or `ctrl` + `⇧` + `⌦`	`⌘` + `X`	Cut current line or selected block to buffer
`ctrl` + `C` `ctrl` + `Insert`	`⌘` + `C`	Copy current line or selected block to buffer
`ctrl` + `V` `⇧` + `Insert`	`⌘` + `V`	Paste from buffer
`ctrl` + `D`	`⌘` + `D`	Up current line or selected block
`⇧` + `F5`	`⇧` + `F5`	Clone root
`ctrl` + `↑` or `↓`	`⌘` + `↑` or `↓`	Expand or Shrink block selection region
`ctrl` + `⇧` + `↑` or `↓`	`⌘` + `⇧` + `↑` or `↓`	Move statements Up or Down
`⇧` + `Arrows`	`⇧` + `Arrows`	Extend the selected region to siblings
`ctrl` + `W`	`⌘` + `W`	Select successively increasing code blocks
`ctrl` + `⇧` + `W`	`⌘` + `⇧` + `W`	Decrease current selection to previous state

Windows or Linux	MacOS	Action
ctrl + Y	⌘ + Y	Delete line
ctrl + Z	⌘ + Z	Undo
ctrl + ⇧ + Z	⌘ + ⇧ + Z	Redo
⌥ + F12	⌥ + F12	Show note in AST explorer
F5	F5	Refresh
ctrl + MINUS	⌘ + MINUS	Collapse
ctrl + ⇧ + MINUS	⌘ + ⇧ + MINUS	Collapse all
ctrl + PLUS	⌘ + PLUS	Expand
ctrl + ⇧ + PLUS	⌘ + ⇧ + PLUS	Expand all
ctrl + ⇧ + 0-9	⌘ + ⇧ + 0-9	Set bookmark
ctrl + 0-9	ctrl + 0-9	Go to bookmark
Tab	Tab	Move to the next cell
⇧ + Tab	⇧ + Tab	Move to the previous cell
Insert	ctrl + N	Create Root Node (in the Project View)

Table 11.5: Editing (Part 2/2)

Windows or Linux	MacOS	Action
`ctrl`+`B` or `ctrl`+`click`	`⌘`+`B`	Go to root node
	or `⌘`+`click`	Go to declaration
`ctrl`+`N`	`⌘`+`N`	
`ctrl`+`⇧`+`N`	`⌘`+`⇧`+`N`	Go to file
`ctrl`+`G`	`⌘`+`G`	Go to node by id
`ctrl`+`⇧`+`A`	`⌘`+`⇧`+`A`	Go to action by name
`ctrl`+`⌥`+`⇧`+`M`	`⌘`+`⌥`+`⇧`+`M`	Go to model
`ctrl`+`⌥`+`⇧`+`S`	`⌘`+`⌥`+`⇧`+`S`	Go to solution
`ctrl`+`⇧`+`S`	`⌘`+`⇧`+`S`	Go to concept declaration
`ctrl`+`⇧`+`E`	`⌘`+`⇧`+`E`	Go to concept editor declaration
`⌥`+`Left` or `Right`	`ctrl`+`Left` or `Right`	Go to next or previous editor tab
`Esc`	`Esc`	Go to editor (from tool window)
`⇧`+`Esc`	`⇧`+`Esc`	Hide active or last active window
`⇧`+`F12`	`⇧`+`F12`	Restore default window layout
`ctrl`+`⇧`+`F12`	`⌘`+`⇧`+`F12`	Hide all tool windows
`F12`	`F12`	Jump to the last tool window

Table 11.6: Navigation (Part 1/2)

Windows or Linux	MacOS	Action
`ctrl` + `E`	`⌘` + `E`	Recent nodes popup
`ctrl` + `⌥` + `Left` or `Right`	`⌘` + `⌥` + `Left` or `Right`	Navigate back or forward
`⌥` + `F1`	`⌥` + `F1`	Select current node in any view
`ctrl` + `H`	`⌘` + `H`	Concept or Class hierarchy
`F4` or `↵`	`F4` or `↵`	Edit source or View source
`ctrl` + `F4`	`⌘` + `F4`	Close active editor tab
`⌥` + `2`	`⌥` + `2`	Go to inspector
`ctrl` + `F10`	`⌘` + `F10`	Show structure
`ctrl` + `⌥` + `)`	`⌘` + `⌥` + `)`	Go to next project window
`ctrl` + `⌥` + `(`	`⌘` + `⌥` + `(`	Go to previous project window
`ctrl` + `⇧` + `Right`	`ctrl` + `⇧` + `Right`	Go to next aspect tab
`ctrl` + `⇧` + `Left`	`ctrl` + `⇧` + `Left`	Go to previous aspect tab
`ctrl` + `⌥` + `⇧` + `R`	`⌘` + `⌥` + `⇧` + `R`	Go to type-system rules
`ctrl` + `⇧` + `T`	`⌘` + `⇧` + `T`	Show type
`ctrl` + `H`	`ctrl` + `H`	Show in hierarchy view
`ctrl` + `I`	`⌘` + `I`	Inspect node

Table 11.7: Navigation (Part 2/2)

Windows or Linux	MacOS	Action
ctrl + F9	⌘ + F9	Generate current module
ctrl + ⇧ + F9	⌘ + ⇧ + F9	Generate current model
⇧ + F10	⇧ + F10	Run
ctrl + ⇧ + F10	⌘ + ⇧ + F10	Run context configuration
⌥ + ⇧ + F10	⌥ + ⇧ + F10	Select and run a configuration
ctrl + ⇧ + F9	⌘ + ⇧ + F9	Debug context configuration
⌥ + ⇧ + F9	⌥ + ⇧ + F9	Select and debug a configuration
ctrl + ⌥ + ⇧ + F9	⌘ + ⌥ + ⇧ + F9	Preview generated text
ctrl + ⇧ + X	⌘ + ⇧ + X	Show type-system trace

Table 11.8: Generation

Windows or Linux	MacOS	Action
ctrl + O	⌘ + O	Override methods
ctrl + I	⌘ + I	Implement methods
ctrl + /	⌘ + /	Comment or uncomment with block comment
ctrl + F12	⌘ + F12	Show nodes
ctrl + P	⌘ + P	Show parameters
ctrl + Q	ctrl + Q	Show node information
ctrl + Insert	ctrl + N	Create new ...
ctrl + ⌥ + B	⌘ + ⌥ + B	Go to overriding methods or Go to inherited classifiers
ctrl + U	⌘ + U	Go to overridden method

Table 11.9: BaseLanguage and Editing

Windows or Linux	MacOS	Action
`ctrl` + `K`	`⌘` + `K`	Commit project to VCS
`ctrl` + `T`	`⌘` + `T`	Update project from VCS
`ctrl` + `V`	`ctrl` + `V`	VCS operations popup
`ctrl` + `⌥` + `A`	`⌘` + `⌥` + `A`	Add to VCS
`ctrl` + `⌥` + `E`	`⌘` + `⌥` + `E`	Browse history
`ctrl` + `D`	`⌘` + `D`	Show differences

Table 11.10: Version Control System and Local History

Windows or Linux	MacOS	Action
`F6`	`F6`	Move
`⇧` + `F6`	`⇧` + `F6`	Rename
`⌥` + `⌦`	`⌥` + `⌦`	Safe Delete
`ctrl` + `⌥` + `N`	`⌘` + `⌥` + `N`	Inline
`ctrl` + `⌥` + `M`	`⌘` + `⌥` + `M`	Extract Method
`ctrl` + `⌥` + `V`	`⌘` + `⌥` + `V`	Introduce Variable
`ctrl` + `⌥` + `C`	`⌘` + `⌥` + `C`	Introduce constant
`ctrl` + `⌥` + `F`	`⌘` + `⌥` + `F`	Introduce field
`ctrl` + `⌥` + `P`	`⌘` + `⌥` + `P`	Extract parameter
`ctrl` + `⌥` + `M`	`⌘` + `⌥` + `M`	Extract method
`ctrl` + `⌥` + `N`	`⌘` + `⌥` + `N`	Inline

Table 11.11: Refactoring

Windows or Linux	MacOS	Action
F8	F8	Step over
F7	F7	Step into
⇧ + F8	⇧ + F8	Step out
F9	F9	Resume
⌥ + F8	⌥ + F8	Evaluate expression
ctrl + F8	⌘ + F8	Toggle breakpoints
ctrl + ⇧ + F8	⌘ + ⇧ + F8	View breakpoints

Table 11.12: Debugger

Bibliography

[Cam14]   Fabien Campagne. *The MPS Language Workbench: Volume I*. Volume 1. Fabien Campagne, 2014 (cited on pages 9, 11, 12, 95, 99).

[Coo+07] Dianne Cook et al. "Exploring gene expression data, using plots". In: *Journal of data science: JDS* 5 (Jan. 2007), pages 151–182. ISSN: 1683-8602. URL: http : / / www . researchgate . net / publication / 228684305 % 5C _ Exploring % 5C _ gene % 5C _ expression%5C_data%5C_using%5C_plots (cited on page 54).

[Dmi04]   Sergey Dmitriev. *Language oriented programming: The next programming paradigm.* 2004. URL: http://www.jetbrains.com/ mps/docs/Language_Oriented_Programming.pdf (cited on page 9).

[Ven80]   John Venn. "I. On the diagrammatic and mechanical representation of propositions and reasonings". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 10.59 (1880), pages 1–18 (cited on page 58).

Index