TestWorkflow × | edgeR diff exp × | GSE59364_DC_all.csv × | ColumnGroupContainer × | TableC.csv ×

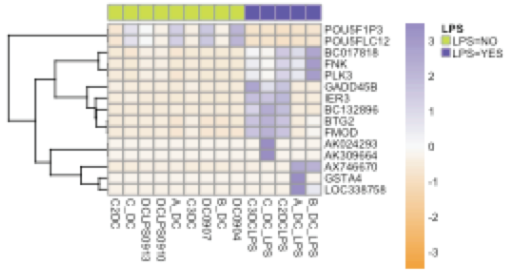
```
Analysis edgeR diff exp
{
  import table GSE59364_DC_all.csv

  subset rows GSE59364_DC_all.csv when true: $(gene) != "Total" -> filtered

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

  join ( filtered, Results ) by group ID -> MergedResults
  subset rows MergedResults when true: ($(FDR) < 0.01) & ($(logFC) > 2 | $(logFC) < -2) -> 1% FDR
  build heatmap with 1% FDR select data by one or more group LPS=YES, group LPS=NO -> plot [
    annotate with these groups: LPS
    scale values: scale by row
    cluster columns: false cluster rows: true
  ]
  Multiplot -> PreviewHeatmap [  1 cols x 1 rows ]   Hide preview
```

plot




```
  render plot as PDF named "heatmap.pdf" ...   rendering
```

edgeR diff exp | GSE59364_DC_all.csv | ColumnGroupContainer

MetaR Documentation Booklet

http://metaR.campagnelab.org

PUBLISHED BY FABIEN CAMPAGNE

HTTP://BOOKS.CAMPAGNELAB.ORG

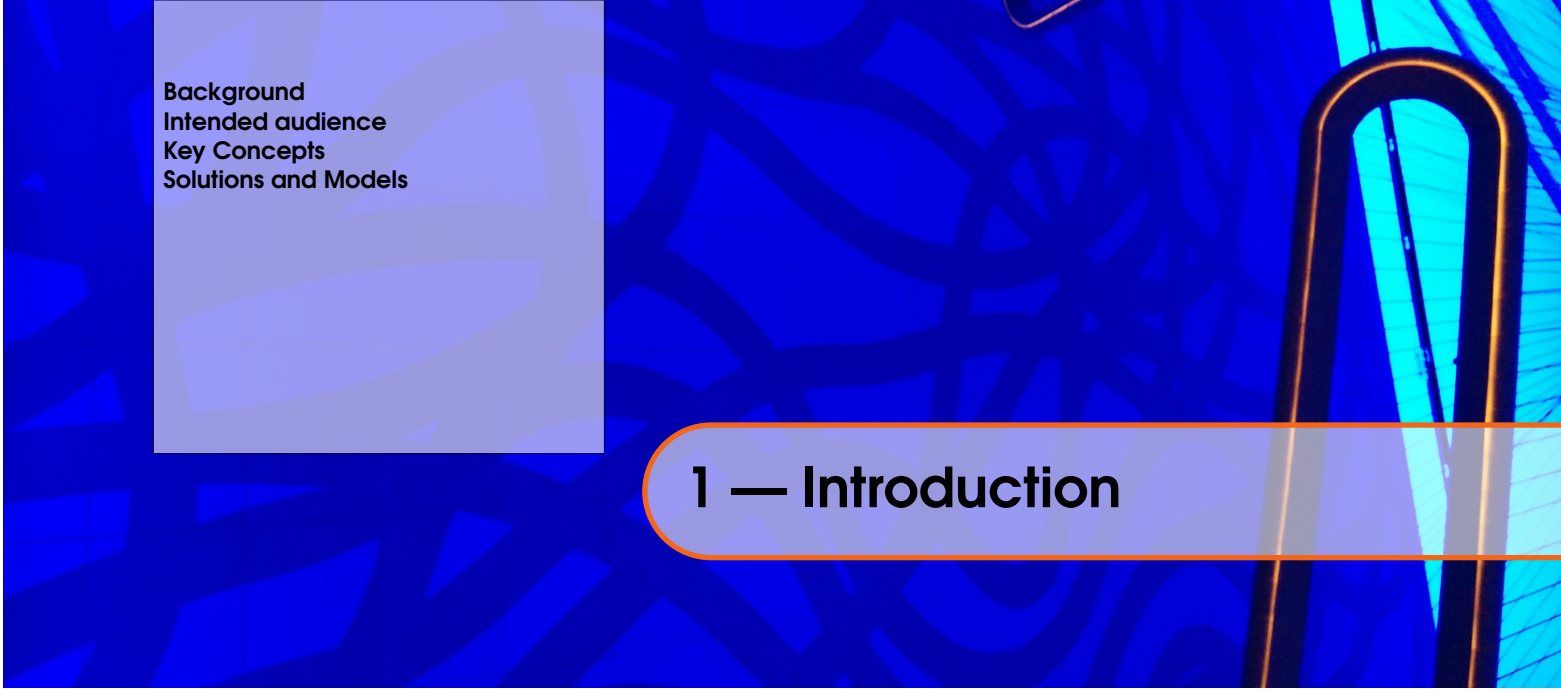
Version 1.1.2, Feb 2015

Credits

We thank the developers of the Meta Programming System, who have developed MPS since the early 2000. MetaR would not have been possible without them.
MPS Project leaders, in chronological order: Sergey Dmitriev, Igor Alshannikov, Konstantin Solomatov and Alexander Shatalin. **Current team members:** Alexander Shatalin, Fedor Isakov, Mihail Muhin, Michael Vlassiev, Václav Pech, Simon Alperovich, Daniil Elovkov, Victor Matchenko, Artem Tikhomirov, Mihail Buryakov and Alexey Pyshkin. **Earlier members of the MPS team:** Evgeny Gryaznov, Timur Abishev, Julia Beliaeva, Cyril Konopko, Ilya Lintsbah, Gleb Leonov, Evgeny Kurbatsky, Sergey Sinchuk, Timur Zambalayev, Maxim Mazin, Vadim Gurov, Evgeny Geraschenko, Darja Chembrovskaya, Vyacheslav Lukianov and Alexander Anisimov. **And these external contributors:** Sascha Lisson, Thiago Tonelli Bartolomei and Alexander Eliseyev.

We also thank the many people who have taken the MetaR training sessions at the Clinical Translational Science Center (Weill Cornell Medical College, Memorial Sloan Kettering, and Hospital for Special Surgery and Hunter College). Their feedback we received during these sessions have been instrumental in rapidly making MetaR user-friendly.

Contents



Background
Intended audience
Key Concepts
Solutions and Models

1 — Introduction

1.1 Background

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` [**Dmitriev:2004**]. MPS is a mature Language Workbench that makes it relatively easy to create new languages and tools to help users of these languages [**campagne2014mps**].

1.2 Intended audience

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.

1.3 Key Concepts

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.

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.

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 Chapter 4 to learn more about language composition in MetaR.

1.4 Solutions and Models

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/`[**campagne2014mps**], or watch the beginning of the MetaR training video at `http://campagnelab.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 [**campagne2014mps**]) 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.

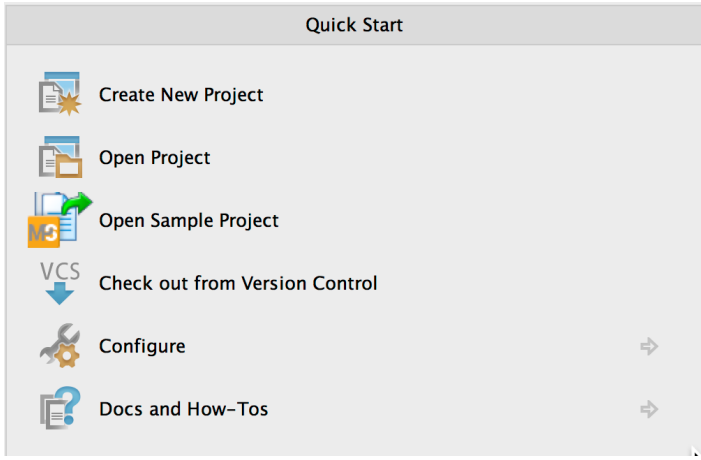


Figure 1.1: **The Quick Start menu.** This menu is displayed when you first open MPS, or when you have no project currently open in MPS. The first item in the menu is used to create a new project.

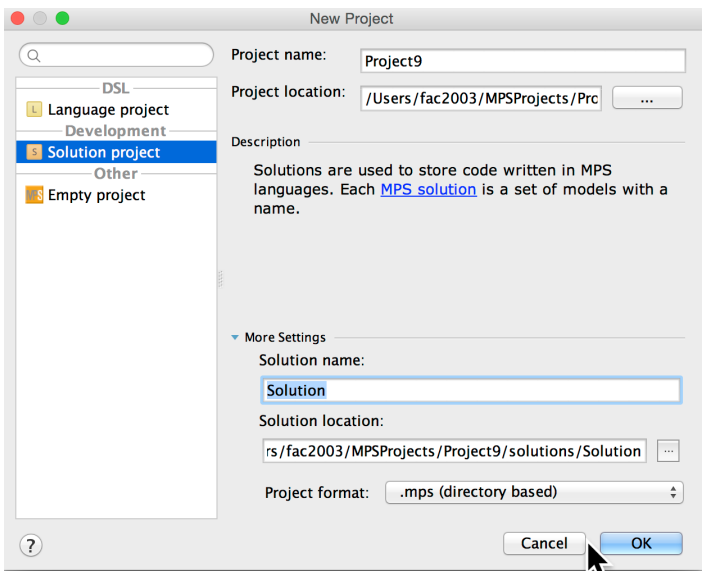


Figure 1.2: **The New Project Dialog.** Use this dialog to create a new Solution. Solutions are used to store models and express your solutions to specific analysis problems. Select the "Solution Project" project type on the left panel, name the project, and name the solution you wish to create.

2 — Tables

2.1 Overview

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.

2.2 Create a Table

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.

```
Table <no name> ...
File Path
  <no pathToResolve>
Columns
  <no name>:
```

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.

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 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.

2.3 Column Groups Container

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.

2.4 Column Groups

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 column group (group for short).

```
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.

Figure 2.3: **Empty Column Group Container.** Use a column group container to define column groups and associated group usages. Place the cursor over the « ... » symbol and press ↵ to add a new group or group usage. Name the group or usage immediately after creating it.

```
Column Groups and Usages

Define Usages:
  << ... >>

Define Groups:
  << ... >>
```

Figure 2.4: **New Group.** You should name a new group immediately after creating it.

```
<no name> used for << ... >>
```

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 (ctrl+␣ to locate the name, then pressing ⇥ 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 ↵ 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.

2.5 Column Group Usage

A Column Group Usage can be defined by pressing ↵ 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 ⇥. Keep pressing ⇥ to add more empty group usages.

2.6 Example Column Group Container

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.

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
```

3 — Analyses

3.1 The MetaR Analysis Root Node

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 `{ }`.

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

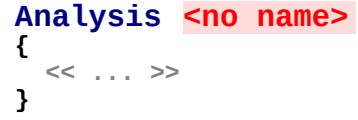


Figure 3.1: **New MetaR Analysis Root Node.** This figure presents a freshly created MetaR analysis root node. You can press ⏎ over the « ... » to add Statement nodes to the analysis. Use auto-completion to discover which types of statements are available.

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.

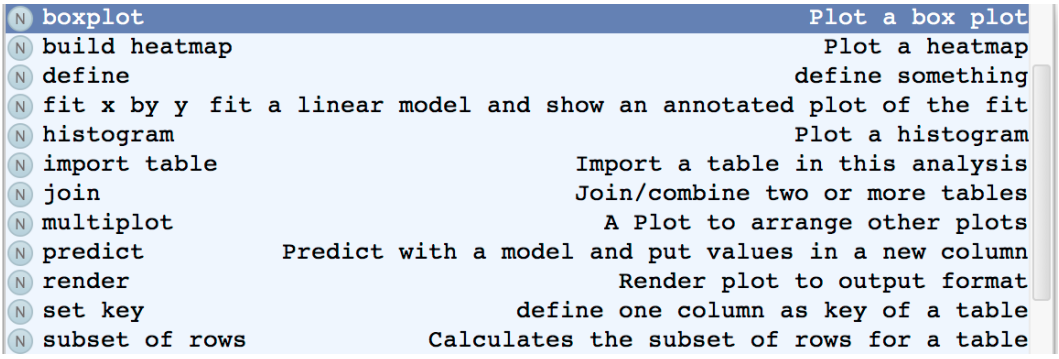


Figure 3.2: **Auto-completion Dialog for Statements.** This dialog is obtained by placing the cursor where a Statement is valid, and pressing Ctrl+␣.

```
Analysis Typing aliases tutorial
{
  import
  import table <no table>
}
```

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.

3.2 Styles

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 on the model with a given name.

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.

A Style is essentially a container for Style Items. Items are well-defined settings that are applied by MetaR to render a graphical object. Figure 3.5 shows a snapshot of the auto-completion menu when looking for items to add.

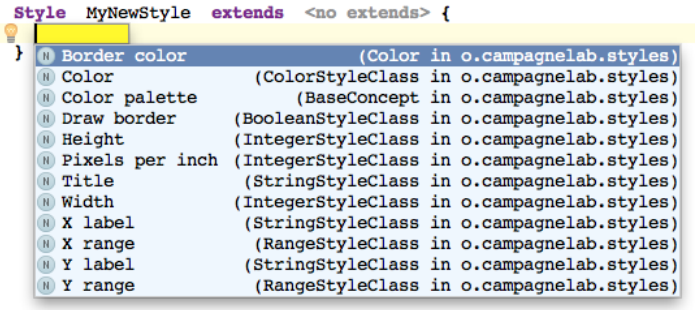


Figure 3.5: **Style with Style Items.** A Style is a container for Style Items. By using auto-completion ctrl + [ ], style items can be added to the node.

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.

3.2.1 Binding Styles to Statements

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 most simple and convenient way to create a style and bound it to a statement in a single step is to use the intention "Create New Style" available on compatible statements as shown in Figure 3.6.

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 at a later time. In this case, the list of Styles visible to a statement in the auto-completion menu is limited

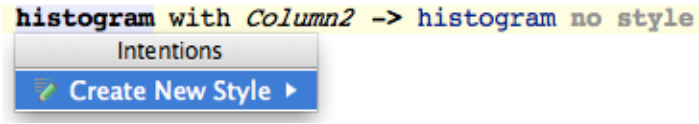


Figure 3.6: **Create New Style Intention on Statements.** The user has pressed ⎵+↵ on a statement that can be bound to a style. By selecting the *Create New Style* intention, a new default style is created in the model and bound to the statement.

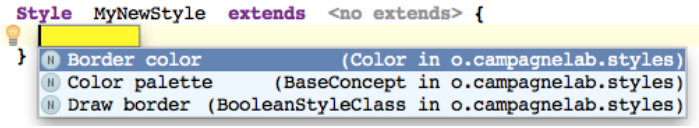


Figure 3.7: **Style with Restricted Items.** A subset of Style Items is displayed in the auto-completion menu when the Style is bound to a statement.

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.

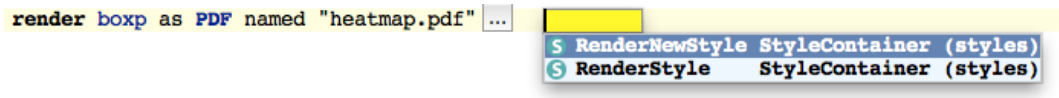


Figure 3.8: **Styles visible from a Statement.** A statement can be bound only to Styles that define items compatible with the statement.

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 minimum.

3.3 Working with Tables

3.3.1 Import Table

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 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

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

```
write <no table> to "<no path>" ...
```

Figure 3.10: **New Subset Rows Statement.** Use this statement to filter the rows of a table.

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

the table AST root node.

3.3.2 Write Table

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.

3.3.3 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.

3.4 Subset Rows

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 3.10 shows a new `subset rows` statement.

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.

subset

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

3.4.1 Example

Figure 3.11 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 3.11: **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).

3.4.2 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.

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.

Figure 3.12: **New Join Statement.** Use this statement to join two or more tables into one.

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

3.5 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 statement is `join`. Figure 3.12 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.

R The join statement will create new columns if the input tables have columns with the same name. In this case the shared columns are renamed “colname.TableName”. Place the cursor on result and look at the column preview in the inspector to see which column names are generated by a given join statement.

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.

3.5.1 How Join works

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 3.13 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 3.13: **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 3.14.

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 3.14: **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 3.13, 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 3.15. 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 3.15: **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

3.5.2 Example Join

Figure 3.16 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 3.17).

3.6 Plotting Data

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

- boxplot alias `boxplot`
- histogram alias `histogram`

[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.

Figure 3.16: **Example of Join Statement.**

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

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

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

Figure 3.18: **New Boxplot Statement.**

- 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.

3.6.1 boxplot

Figure 3.18 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 ⌤ to define more than one column. Use auto-completion to locate individual columns from the imported tables, or the tables created by prior statements.

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

Different kind of plots accept different types of styles. The boxplot accepts a `ColorPlot Style`. You can create this node in the model and bind the boxplot statement to it to customize the colors that will be used to draw this boxplot.

3.6.2 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

Different kind of plots accept different types of styles. The histogram accepts a `ColorPlot Style`. You can create this node in the model and bind the histogram statement to it to customize the colors that will be used to draw this plot.

3.6.3 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 3.19 presents a new `fit` statement.

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

Figure 3.19: **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.

```
scatter plot style <no name> {
  x label : <no xLabel>
  y label : <no yLabel>
  title : <no title>
  pixel width : <no pixelWidth>
  pixel height : <no pixelHeight>

  x range :
    <no xRange>
  y range :
    <no yRange>
}
```

Figure 3.20: **New Scatterplot Style.**

style

Different kind of plots accept different types of styles. The fit statement accepts a `ScatterPlotStyle`. You can create this node in the model and bind the fit statement to it to customize the title, axes and range of data in the scatterplot. Figure 3.20 shows the attributes of the `ScatterPlotStyle` node.

R Styles will refactored in a future version of MetaR to improve their flexibility and remove redundant style definitions. This documentation does not describe styles in detail for the moment, and will be updated to describe the new Style functionality.

3.6.4 heatmap

Use this statement (`heatmap`) to construct a heatmap. Figure 3.21 presents a new `heatmap` statement.

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

Figure 3.21: **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.

annotations

You can use the "Attach Annotations" intention (💡) when the cursor is on top of `heatmap` to customize heatmap annotations. Figure 3.22 shows a heatmap statement with annotations.

```
heatmap with <no table> select data by<no dataSelection>-> <no name> [
  annotate with these groups:<< ... >>
  scale values: <no scaling>
  cluster columns: false cluster rows: false
]
```

Figure 3.22: **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 3.23 presents a heatmap constructed with the build heatmap statement, using annotations and row clustering.

3.6.5 multiplot

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 3.24 presents a new Multiplot.

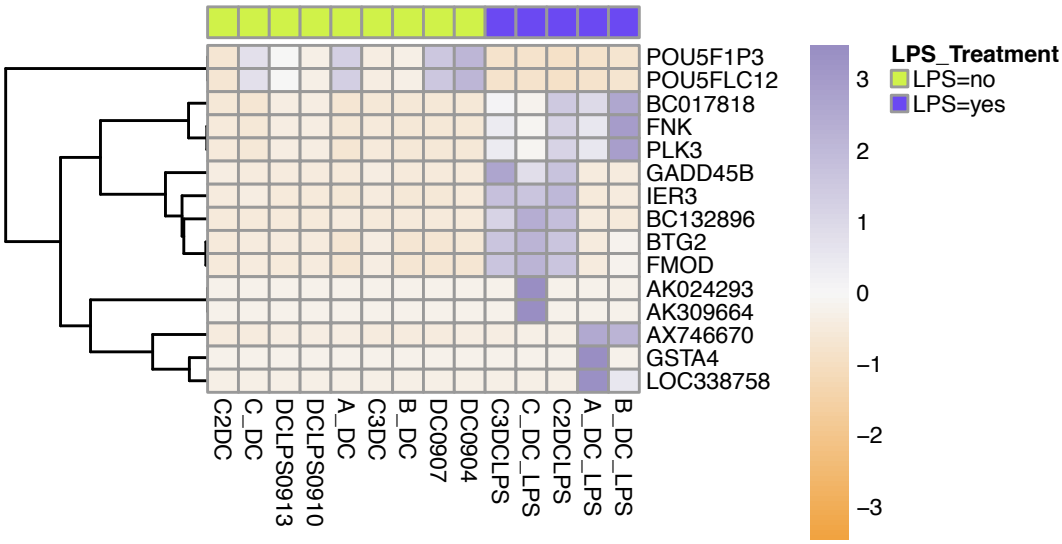


Figure 3.23: **Example of Heatmap Plot.** This heatmap has been customized with annotations. The `LPS_Treatment` group usage is shown in the legend, with two groups: `LPS=yes` and `LPS=no`. The values plotted have been scaled by rows, and the rows clustered. Data are from `http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE59364`.

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

Figure 3.24: **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 3.25 presents an example of multiplot.

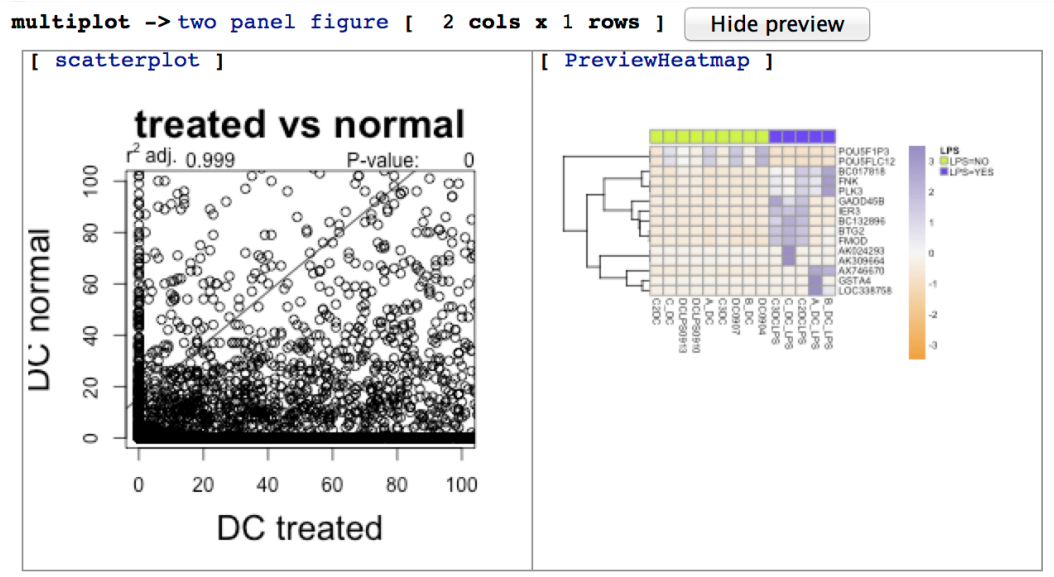


Figure 3.25: **Example of Multiplot.** This example shows a multiplot composed of two columns and one row. In the preview, a fit x by y plot is shown on the left, and a heatmap on the right.

4 — EdgeR

4.1 Understanding Language Composition

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 4.1 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 4.2. Adding 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.

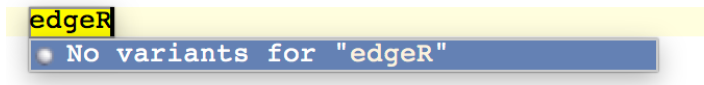


Figure 4.1: **Error When Typing the EdgeR Alias.** The `EdgeR` statement is not yet defined.

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

Figure 4.2: **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.

4.2 The edgeR Statement

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.

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 4.3 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

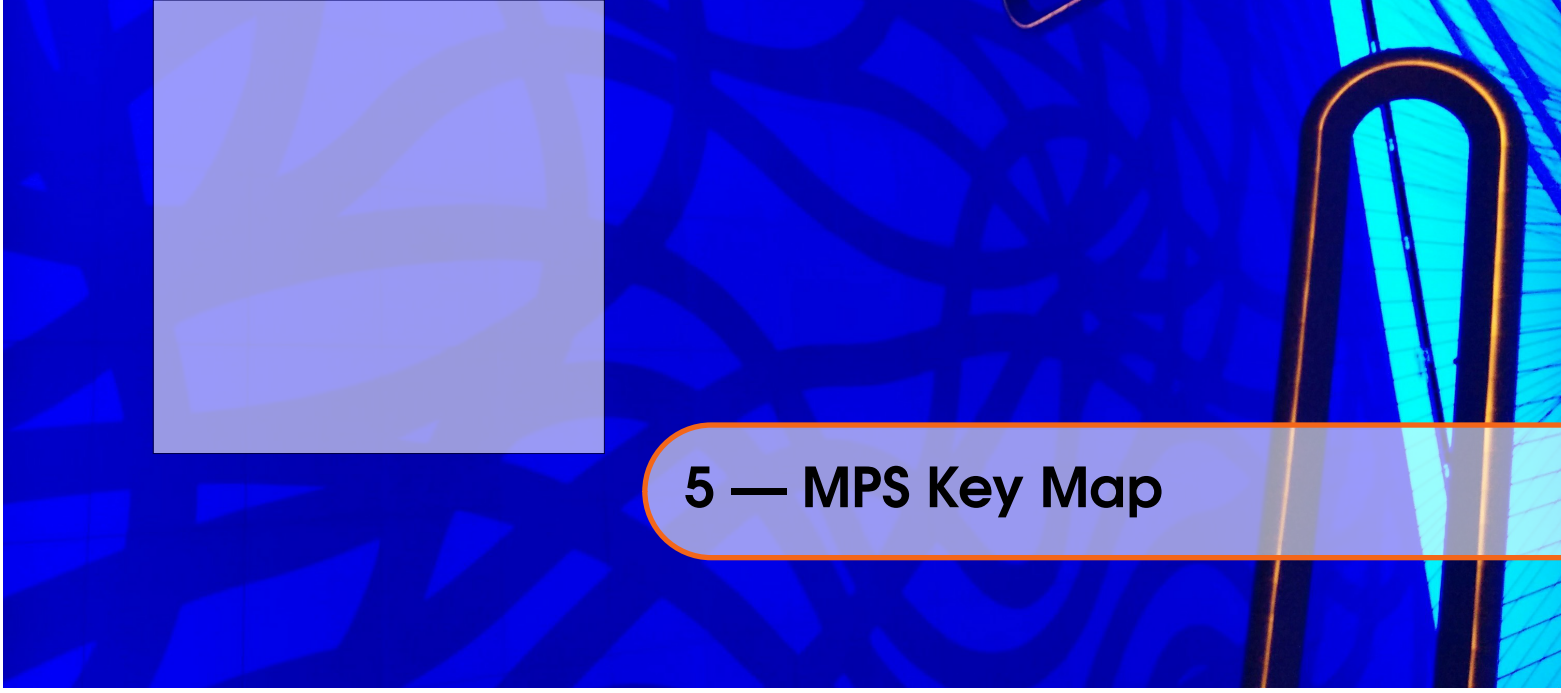
Place the cursor on the keyword following normalize with and use auto-completion to switch from one type of normalization method to another. The the EdgeR Bioconductor documentation `http://www.bioconductor.org/packages/release/bioc/html/edgeR.html` for details about these approaches.

4.3 Example

Figure 4.3 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 4.3: **EdgeR Example.**



5 — 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 5.1: 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 5.2: 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 5.3: 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

Table 5.4: Editing (Part 1/2)

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 5.5: Editing (Part 2/2)

Windows or Linux	MacOS	Action
ctrl + B or ctrl + click	⌘ + B or ⌘ + click	Go to root node / 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 5.6: 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 5.7: 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 5.8: 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 5.9: 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

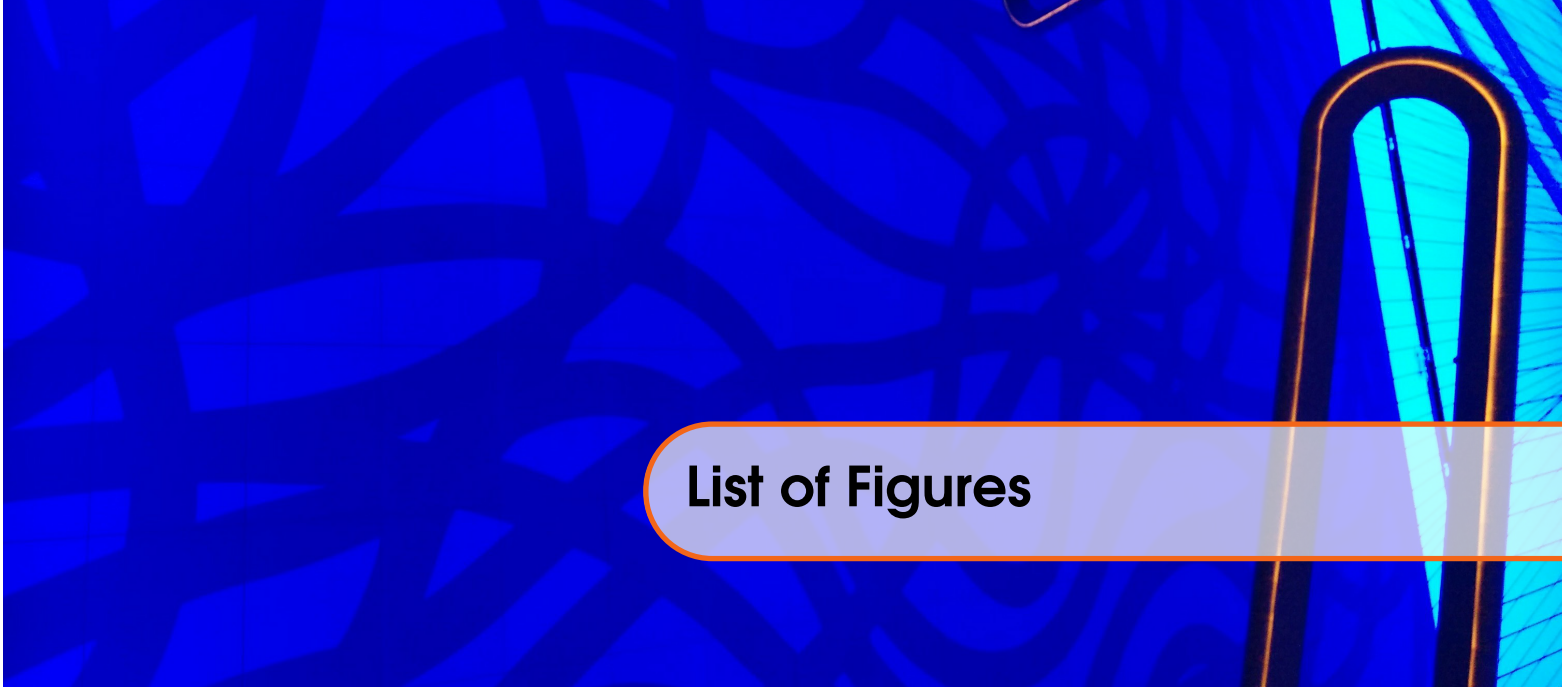
Table 5.10: Version Control System and Local History

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 5.11: 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 5.12: Debugger



List of Figures