Deltek

Deltek Maconomy®

Maconomy Reporting for Report Developers

May 14, 2021

While Deltek has attempted to verify that the information in this document is accurate and complete, some typographical or technical errors may exist. The recipient of this document is solely responsible for all decisions relating to or use of the information provided herein.

The information contained in this publication is effective as of the publication date below and is subject to change without notice.

This publication contains proprietary information that is protected by copyright. All rights are reserved. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, or translated into another language, without the prior written consent of Deltek, Inc.

This edition published May 2021.

© Deltek, Inc.

Deltek's software is also protected by copyright law and constitutes valuable confidential and proprietary information of Deltek, Inc. and its licensors. The Deltek software, and all related documentation, is provided for use only in accordance with the terms of the license agreement. Unauthorized reproduction or distribution of the program or any portion thereof could result in severe civil or criminal penalties.

All trademarks are the property of their respective owners.

Contents

MRL Language Reference

The Maconomy Report Language (MRL) is a language for developing reports based on the Maconomy system. This manual describes the syntax and usage of MRL.

This manual describes MRL version 1.3.

Concepts

The Maconomy Report Language (MRL) is a language for developing reports based on the Maconomy system. Currently, two kinds of reports are supported by MRL: universe reports and report collections.

Universe reports

A Universe Report is made up of three elements: parameters, queries, and links.

Element	Description
Parameters	Report parameters are the interface to a report. The user enters parameter values before running a report. Parameters can be used for controlling the queries, e.g. in restrictions that limit the rows retrieved by the query, or for controlling the layout.
Queries	A Universe Report consists of one or more queries using MQL syntax – see “MQL Language Reference”. A query defines all values that need to be calculated for use in the layout, i.e. row values and aggregate values. Aggregate values are used in totals and subtotals.
Links	Reports support hypertext linking. This provides the user with the ability to switch between reports, Web pages and Portal components simply by a clicking a link. Links may be parameterized, allowing data to be shared between the linked resources.

Report collections

Report collections are—as the name suggests—collections of universe reports. Report collections allow report developers to unify a group of related reports, thus reducing the need for code reuse.

Generic reporting

A report may be associated with layouts and a report M-script allowing report developers to customize its appearance and functionality.

Layout

The layout defines what the report will look like when printed (either to the screen or to a printer). The layout is defined in a separate file using MPL syntax – see the manual “Printout Design Using MPL”. If no layout file is specified, a default layout matching the structure of the defined queries is generated.

Report M-Script

By embedding references to M-scripts within the report, a report developer can give parameters generic values and validate data entered by a user. The report M-script is defined in a separate file using M-Script syntax—see the manual “M-Script Language Reference”.

Installing reports

Reports and their associated layouts and report M-scripts must be installed on the server using the MBuilder tool. See the manual “MBuilder Reference”.

Reading this manual

The formal syntax of MRL is presented in a variant of BNF (Bachus Naur Form). In the descriptions of attributes, optional attributes are listed in [square brackets].

Definitions

This section describes the syntax of the language elements in the current version of MRL. A version history of MRL is supplied in section 3 on page 24.

MRL is a tag-based language, and it consists of elements and attributes just like XML. Unlike XML, every attribute in MRL has an associated type, and can have a short form.

```
BNF          mrl ::=
             (<MRL 1.3> | MRL 1.3)
             universereport | reportcollection
```

Universe report

The `universereport` element defines a universe report.

```
BNF          universereport ::=
             <universereport (name = module|module) [title = string|string] > [help]
                    parameter* queries [links] [fieldlinks]
                    <end universereport>
```

Attributes of the `universereport` element:

Name	Type	Description
`Name`	*module*	The name of the universe report. The name is used when referring to the report
`[title]`	*string*	The title of the report.

In the following example, a very simple universe report is defined. The name is `reference::R001` and the title is "`Reference Manual Report 001`". It defines the parameter `ParEmployeeNumbers` that is used in the query for restricting rows. The special semantic of the MQL `in` operator ensures that if the user supplies the value "10..20", only employees matching this restriction are shown, and it ensures that when the user does not supply a value for the parameter, no restriction is used, and all employees are shown.

```
<MRL 1.2>
<Universereport reference::R001 "Reference Manual Report 001">
       <Parameter ParEmpolyeeNumbers type=string >
       <Queries>
              MSELECT EmployeeNumber, Name1 FROM Employee
              WHERE EmployeeNumber in ParEmployeeNumbers ORDER BY EmployeeNumber
       <End Queries>
<End Universereport>
```

Help

With the `help` element, external help information for the report can be specified. Help for a report can be specified using 3 levels:

`description` is for a very short description of what the report can be used for. The description is for use in report listings and "mouse-over" help (tooltip).

`summary` is for a short version of the help text.

`text` is for an in-depth description of what the report can be used for, and how it works.

BNF

```
help ::=
<help [(description = string|string)] > [summary]
[text]
<help end>

summary ::=
<summary> [text]
<end summary>

text ::=
<text (text = text|text) /> | text
```

The following example defines a report help element where all levels of help are defined (the short form of `text` is used).

```
...
<help "MRL Reference Manual, Report">
      <summary>
            #This Report is for the MRL Reference Manual#
      <end summary>
            #
            This Report illustrates the help element, and is used as an example in
the MRL Reference Manual.
            #
<end help>
...
```

Parameter

The `parameter` element defines a parameter of the report, which will appear in the selection criteria and might be filled with a value by the user before the report is executed.

A parameter can be used to control the queries, e.g. in restrictions that restrict the rows retrieved by the query, or to control the layout.

BNF

```
parameter ::=
  <parameter (name = id|id)
            (type = typeid|typeid)
            [(title = string|string)]
```

```
                [(basis =
               qualifiedfieldid|qualifiedfieldid)]
               [(mandatory+|mandatory-)]
               [(solitary+|solitary-)]

               [(hidden+|hidden-)]

          [(value = constExpressionShort|constExpressionShort)]

     />
```

Attributes of the `parameter` element:

Name	Type	Description
`Name`	*ID*	The name of the parameter. The name is used when referring to the parameter in the query or in the layout.
`Type`	*typeID*	The expected Maconomy type of the parameter. The type might be affected by the `solitary` attribute, see examples in this section
`[title]`	*string*	The title of the parameter.
`[basis]`	*qualifiedID*	Reference to an existing field from a Maconomy object, e.g. `employee.employeeNumber`. By referring to an existing field, the type, title, and foreign key search are inherited from the field to this parameter.
`[mandatory]`	boolean	Indicates if the parameter is mandatory. A mandatory parameter must be given a value by the user before the report is executed. Default value is `mandatory-`.
`[solitary]`	boolean	Indicates if the parameter is solitary. A solitary parameter accepts a single value, whereas a non-solitary (`solitary-`) parameter accepts a range value, e.g. "`100..200`". The type of the parameter is affected by the solitary parameter, see examples in this section
`[hidden]`	boolean	Indicates if the parameter is hidden. A hidden parameter is not available in the selection criteria. It can be used in

Name	Type	Description
		connection with the 'drill' feature of reports. Default value is `hidden-`
`[value]`	literal	Default value for the parameter. If this attribute is omitted, the Maconomy `type'null` value is assigned to the parameter.

Parameters are associated with every query in the `queries` element, as if they were defined explicitly in each query using the `using-parameters` clause in MQL – see section 2.1.3, "Queries".

In the following example, a mandatory and a solitary parameter with explicit type and title definitions are defined for use with the `eq` operator in the `mql` expression. The `solitary` option means that only a specific date can be entered, and the `mandatory`option means that a value must be entered.

```
...
<Parameter ParStartDate type=date title="Start date" solitary+ mandatory+>
...
MSELECT ...
WHERE FieldDate = ParStartDate
...
```

Example: Default value

In the following example, a default value is assigned to the parameter initializing the parameter with the current date. The value of a parameter is a literal (a constant).

However, by using M-scripts in the reports (see section 2.4 on page 18), it is possible to calculate values for parameters.

```
...
<Parameter ParStartDate type=date title="Start date" solitary+
   mandatory+ value=date'today
>
...
  MSELECT ...
  WHERE FieldDate = ParStartDate
...
```

Example: Field association and foreign key definition

In the following example, a parameter has been associated with the Maconomy field `JobHeader.CustomerNumber` in order to inherit the type, title, and foreign key definition from this field. A foreign key definition enables Ctrl+G search on the parameter in the selection criteria dialog.

```
...
<Parameter ParCustomerNumber basis=JobHeader.CustomerNumber
   solitary+ mandatory+>
```

```
...
  MSELECT ...
  WHERE CustomerNumber = ParCustomerNumber
...
```

Example: Field association and foreign key definition

In the following example, a parameter has been associated with the Maconomy field `JobHeader.CustomerNumber` in order to inherit the type, title, and foreign key definition from this field. A foreign key definition enables Ctrl+G search on the parameter in the selection criteria dialog.

```
...
<Parameter ParCustomerNumber basis=JobHeader.CustomerNumber
   solitary+ mandatory+>
...
  MSELECT ...
  WHERE CustomerNumber = ParCustomerNumber
...
```

Example: Range parameter (solitary-)

In the following example, a `solitary-` parameter with explicit type and title definition is defined for use with the `in` operator in an `mql` expression. The `solitary-` option means that an integer range can be entered by the user, e.g. “10..20”. Note that a parameter with the `solitary-` option always has the type *string*, and note the special semantics of the `in` operator in MQL, where the expression evaluates to `true` if the value of the parameter is empty, which is very different from the semantics of the `eq` operator.

```
...
<Parameter ParInterval type=integer title="Range" solitary->
...
  MSELECT ...
  WHERE FieldInteger in ParDateInterval
...
```

Example: Hiding attributes (hidden+)

Sometimes a report is used as a drill down report, meaning that another report links to it. In such cases, there may be parameters that a user should not be able to set because their value is supplied by the other report. The `hidden+` option can be used for hiding parameters from the selection criteria. In the following example, the `ParStartDate` parameter will be hidden.

```
   ...
<Parameter ParStartDate type=date value=date'today hidden+>
...
   MSELECT ...
   WHERE Date = ParStartDate
...
```

Queries

The `queries` element defines the queries of the Report. A query is an MQL query, that is, a query based on the Maconomy Universe technology. The `queries` element defines all the values, which need to be calculated for use in the layout, i.e. row values and aggregate values. Aggregate values are used in totals and subtotals. For a description of how to make an MQL query, please see the "MQL Language Reference" manual.

BNF

```
queries ::=
  <queries>
    mselect (; mselect)*
<end queries>
```

Parameters defined in the report are associated with each query in the `queries` element, as if they were defined explicitly in each query by means of the `using-parameters` clause in MQL. This means that a parameter is available in each query for use in expressions, e.g. restrictions in a `where` clause.

Besides the explicitly defined parameters, three parameters are implicitly defined, namely: `Report$name`, `Report$title`, and `Report$nameOfUser`. These parameters always make the name of the report, the title of the report and the user name avail- able in each query.

When a report is executed, the actual values for the parameters are passed to each query.

In the following example, a query is using the implicit parameter `Report$nameOfUser` in a query restriction.

```
...
  <Queries>
    MSELECT EmployeeNumber, Name, UserName
    FROM EmployeeUserUniverse
    WHERE Username = Report$NameOfUser
  <End Queries>
  ...
```

When defining multiple queries, a name has to be associated with each query in order to access the columns of a query in the layout. This is done using the cursor naming available in MQL.

In the following example, two queries are defined in the `queries` element. The first query has the name `Header`, and the second query has the name `TimeSheet`.

```
...
<Queries> MSELECT [
    EmployeeNumber, Name
    ] AS CURSOR Header FROM Employee
    WHERE EmployeeNumber = ParmEmployeeNumber;
    MSELECT [
    PeriodStart, PeriodEnd,
    sum(TimeActivity1Total) as TActivity
    ] AS CURSOR TimeSheet FROM TimeSheetHeader
    WHERE EmployeeNumber = ParmEmployeeNumber;
```

```
<End Queries>

...
```

Links

Reports may link to each other much like Web pages link to each other. As of MRL version 1.3, report links can be specified in MRL. Up to MRL 1.3, links had to be specified in the layout, but now they can be specified in both MRL and MPL (in case of conflicts, the MPL links will override the MRL links).

What are links?

Consider two reports: “Earnings Overview” displaying the earnings of the different departments, and “Employee Earnings” displaying the department and earnings of each employee.

Earnings Overview

Department	Earnings
Financial	-1
PR	5

Employee Earnings

Employee	Department	Earnings
Bob	PR	3
John	Financial	-1
Lisa	PR	2

As the reports are related to each other, we would like to link between them such that a user can switch from one report to the other at a click of the mouse. Assume a link has been successfully defined from the Department field of the “Earnings Overview” report to the “Employee Earnings” report (see how to do this in the example “Example: What are links?” on page 15). Then, when a user clicks on a particular entry, say PR, in the “Earnings Overview” report, the “Employee Earnings” report will be opened. However, links provide more functionality than merely opening new reports: As the user specifically selected PR, we can transfer the selected value to the “Employee Earnings” report such that the relevant entries are presented (in this case, the employees of the PR department):

Employee Earnings - PR Department

Employee	Department	Earnings
Bob	PR	3
Lisa	PR	2

It is an important feature of links that they can be parameterized such that data can be transferred between reports.

The links element – link definition

A link specification consists of two parts: link *definitions* and link *associations*. This section describes link definitions; link associations are described in the next section.

A link definition specifies the resource linked to as well as the parameters that should receive a value when linking. A resource need not be a report, it may be a portal component, a URL, or a dialog.

BNF `links :: =`

```
<links>
        link*
<end links>


link :: =
        <link (name = id | id)
                (report = moduleid | portalcomponent = moduleid | url
= url | dialog = moduleid )
                [tooltip = string]
                [reportaction = drill | drillaround | edit] [target =
insideportal | newwindow | currentframe] [skipparametertransfer+ |
skipparametertransfer-]
        >
                (linkparameter | linkparameterfree)*
        <end link>
```

Attributes of the `link` element:

Name	Type	Description
`Name`	*ID*	The name of the link. The name is used when referring to the link.
`Report`	*module*	The identifier of the report linked to.
`Portalcomponent`	*module*	The identifier of the Portal component linked to.
`url`	*url*	The URL linked to
`dialog`	*module*	The identifier of the dialog linked to.
`[tooltip]`	*string*	A short help description associated with the link. The tooltip will appear as a “mouse-over” effect.
`[Reportaction]`	*enum*	This attribute can be used only when linking to a report; it

Name	Type	Description
		controls what happens when the new report is opened. With `drill`, the report is simply opened. With `drillaround`, the report is opened, but the report linked from will be removed from the drill stack (the list of reports previously linked from). With `edit`, the report linked to lets the user specify selection criteria first. The default value is `drill`
`[Target]`	*enum*	This attribute specifies the target window of the link. The possible values are: `new` (show in a new browser window), `currentframe` (replace the current frame), and `insideportal` (show inside the Portal). The attribute is ignored for report links. Default value is `insideportal`
`[Skipparametertransfer]`	*boolean*	If false (–) all parameters of the report linked from are used as link parameters (hence, it is not necessary to write the parameters once again). If true (+) only parameters specified in the subsequent link parameter elements will be transferred. Subsequent link parameter elements will overwrite the transferred parameters. Default value is `Skipparametertransfer-`

Links may transfer values to the parameters of the resource linked to. The link parameters specify these parameters. When linking to a URL, the link parameters are URL parameters. When linking to a Portal component, the link parameters are key fields of the Portal component. When linking to a report, the link parameters are parameters of that report (note the subtle difference between parameters of a report and link parameters).

There are two kinds of link parameters, those called simply `parameter`, which take constant values only, and those called `parameterfree`, which take free parameters.

In the above link example, the “Employee Earnings” report has a parameter of the same type as the Department field, in the example called `parmEmpDepartment`, used for making restrictions on departments. The “Earnings Overview” report will then have a single link parameter, namely `parmEmpDepartment`. Moreover, it will be a free parameter such that the value of `parmEmpDepartment` will be equal to the value selected by the user.

```
BNF          linkparameter ::=
                    <parameter
                           (name = id | id)
                           value = constExpressionShort
                    />

             linkparameterfree ::=
                    <parameterfree (name = id | id) type = type
                    />
```

The attributes of the `parameter` element and the `parameterfree` element:

Name	Type	Description
`Name`	*ID*	The name of the parameter of the resource linked to. When linking to a report or a Portal component, this parameter must be a parameter of that resource.
`Value`	*literal*	The value of the parameter.
`Type`	*typeID*	The Maconomy type of the parameter.

Most often the report we link to shares parameters with the report we link from. In these situations it is not necessary to explicitly specify these parameters as link parameters, because they will automatically be transferred (assuming `skipparametertransfer` is false, which it is by default). This is illustrated in the section "Example: Using skipparametertransfer" on page 16.

The fieldlinks element – link association

The second part of a link specification, the link *associations*, specifies the fields we link from and the values that are to be transferred in the link

There are two kinds of field links, *universe field* links, and *query field* links. Before we explain these, recall that a field identifier may refer either to a field in a query or a field in a universe. For example, with the query.

```
MSELECT [DepName, Earnings] as cursor DepCursor FROM Department
```

The field `Department.DepName` refers to field in a universe, whereas `DepCursor.DepName` refers to a field in a query.

A query field link defines a link from a field in a query, and a universe field link defines a link from a field in a universe (hence a universe field link must be associated with a universe). As every field in a report belongs to a query, all links can be expressed using query field links. However, in practice it may be more convenient to use universe field links, for example if you rename a field or have several queries that use the same universe field (this is illustrated later in the section "Example: Using universefieldlinks" on page 15).

```
       fieldlinks ::=
              <fieldlinks
```

```
                [universe = moduleid] [interface = id]
        >
                (queryfieldlink     | universefieldlink)+
</fieldlinks>

queryfieldlink ::=
        <queryfieldlink name = qualifiedid
link = functionexpression [hidden+ | hidden-]
        />

universefieldlink ::=
        <universefieldlink name = qualifiedid
        link = functionexpression [universe = moduleid]
        [interface = id] [hidden+ | hidden-]
        />


functionexpression ::=
id ( linkargument (, linkargument)* )


linkargument ::=
        constExpressionShort | qualifiedid | functionfieldid
```

Attributes of the `fieldlinks`, `queryfieldlink`, and `universefieldlink` elements:

Name	Type	Description
`Name`	*qualifiedID*	The name of the field linked from.
`Link`	*funcExp*	Parameterized reference to a link. The link must be defined in the link definition, and the arguments must match the link parameters. The arguments determine the actual values of the link parameters.
[`Universe`]	*module*	The name of the universe to which a universe field belongs. This attribute must be specified for universe field links such that the links are associated with a universe. The attribute may be specified as an attribute of the `fieldlinks` element. If the attribute belongs to the `fieldlinks` element, every universe field link inherits the

Name	Type	Description
		value, i.e. will be associated with the universe, but the universe field links may overwrite it.
[Interface]	*ID*	Interface of the universe. If no interface is specified, the default interface will be used. If the attribute belongs to the `fieldlinks` element, every universe field link inherits the value, i.e. will be associated with the interface, but the universe field links may overwrite it.
[Hidden]	*boolean*	Specifies if the link should be hidden. A hidden link will not be available.

All references to fields in a universe field link must belong to the associated universe.

The universe and interface attributes are introduced at the `fieldlinks` element only to shorten the specification of universe field links. An example where universe field links share attributes:

```
<fieldlinks>
  <universefieldlink L1 universe=U1 link= ...>
  <unviersefieldlink L2 universe=U2 link= ...>
  <unviersefieldlink L3 universe=U1 link= ...>
<end fieldlinks>
```

Which is equivalent to

```
<fieldlinks universe=U1>
  <universefieldlink L1 link= ...>
  <unviersefieldlink L2 universe=U2 link= ...>
  <unviersefieldlink L3 link= ...>
<end fieldlinks>
```

Example: What are links?

The following two reports specify the example mentioned in the section “What are links?” on page 10.

```
<UniverseReport EarningsOverview "Earnings Overview">
   <Parameter parmDepartment type=string/>
   <Queries>
      MSELECT [DepName, Earnings] as cursor DepCursor FROM Department
      WHERE DepName IN parmDepartment;
   <End Queries>
   <links>
```

```
     <link myFirstLink report=EmployeeEarnings>
          <parameterfree parmEmpDepartment type=string/>
        <end link>
     <end links>
     <fieldlinks>
       <queryfieldlink DepCursor.DepName

  link=myFirstLink(DepCursor.DepName)/>
     <end fieldlinks>
  <end Universereport>
<UniverseReport EmployeeEarnings "Employee Earnings">
  <Parameter parmEmpDepartment type=string/>
  <Queries>
     MSELECT [Name, Department, Earnings] as cursor EmpCursor FROM Employee
     WHERE Department=parmEmpDepartment;
  <End Queries>
<end Universereport>
```

Example: Using universefieldlinks

In the above example we used a query field link, but we could just as well have used a universe field link:

```
  <universefieldlink DepName universe=Department
                                          link=myFirstLink(DepName)/>
```

This would make no difference. However, if the “Earnings Overview” report had had an additional query selecting the same field:

```
...
  <Queries>
     MSELECT [DepName, Earnings] as cursor DepCursor FROM Department
       WHERE DepName IN parmDepartment;
     MSELECT [DepName, Earnings] as cursor DepCursor2 FROM Department
       WHERE DepName IN parmDepartment;
  <End Queries>
...
```

things would be different. Using the query field link (of the section “Example: What are links?” on page 15), we would have links only in the first query; using the universe field link, we would have links in both queries.

Example: Using skipparametertransfer

The skipparametertransfer attribute of the link tag is introduced to make it easier to specify links in which parameters are automatically transferred when we link.

Assume our two reports shared parameters, i.e., that both had the parameter

```
<Parameter parmDepartment type=string/>
```

And assume we did not explicitly transfer parameters when we linking, that is the link was defined as

```
<links>
   <link my2Link report=EmployeeEarnings/>
<end links>
<fieldlinks>
   <queryfieldlink DepCursor.Name link=my2Link()/>
<end fieldlinks>
```

This would change the link functionality. As the attribute `skipparametertransfer` is by default false, we automatically transfer parameters when linking. Thus, when we open the “Employee Earnings” report, the value of its parameter will be equal to the value of the parameter in the “Earnings Overview” report. Hence, it will not be equal to the selected entry.

If we skipped parameter transfer, i.e., with the following link definition

```
<link my2Link report=EmployeeEarnings skipparametertransfer+/>
```

The value of the parameter `parmDepartment` of the Earnings Overview report would be its default value.

Example: Linking to a Web page

The following link opens the Maconomy homepage in a new window:

```
<links>
   <link my3Link url=http://www.maconomy.com target=newwindow>
      <parameterfree parmEmpDepartment type=string/>
   <end link>
<end links>
```

As this link is parameterized, the link parameter is transferred as a parameter in the URL, so we actually open the URL:

`http://www.maconomy.com/?parmEmpDepartment=PR` (assuming PR was selected).

Report collections

The `reportcollection` element defines a report collection. A report collection is essentially a list of universe reports that share parameter and link definitions.

BNF

```
reportcollection ::=
  <reportcollection (name = module|module) [(title = string|string)] >
  [help]
     parameter*
     universereport+ [links] [fieldlinks]
```

```
<end reportcollection>
```

Where `name`, `title`, `help`, `parameter`, `universereport` and `links` are defined in the previous section.

Each of the universe reports in the report collection functions as an ordinary universe report. You refer to a member by qualifying its name with the name of the collection, hence `A::B::Member1` refers to the member `Member1` of the collection `A::B`. The names of the members cannot be qualified (cannot contain `::`). The collection itself does not contain a query and cannot be executed. As the members share parameters, they are not allowed to have a parameter definition. The members will also share the links defined for the collection, but each member may additionally introduce its own links.

The members of a collection are independent of each other, thus each member has its own instances of the parameters. It is, nonetheless, easy to copy the values of the parameters when linking from one member to another—this happens per default (`skipparametertransfer` is by default false).

Example

```
<Reportcollection A::B "My first report collection">
   <parameters>
     <parameter MyParm title="My Parameter" type=string      />
</parameters>

<Report Member1 "My first report member">
   <Queries>
      ...
   <End queries>
   <Links>
      ...
   <End Links>
<End report>

<Report Member 2 "My second report member">
      ...
<End report>
<End reportcollection>
```

The first member has a link definition, meaning this report (and this report only) has links. If the links were defined for the collection (at the outermost level), both members would inherit the links.

Layouts

Both universe reports and report collections may be associated with layouts allowing report developers to control how a report should look. A universe report may be associated with at most one layout (MPL) file, making it unproblematic to associate the two. A report collection, on the other hand, may be associated with several layouts. Each of the layouts must refer to the report member it is to be associated with. This is done adding a `print` attribute to the `layout` tag of the layout. For example, if an MPL file contains

```
<layout print="Member2">
```

then the layout will be associated with the member named `Member2`. Layouts cannot be associated with the collection itself and at most, one layout may be associated with each member of the collection.

Report M-scripts

The definition of parameters is not as flexible as we would sometimes want. A significant limitation is the fact that the default value of a parameter must be a constant expression. In some cases, we want to be able to express more details, for example to say that the default value should be the user's latest entry to the database or the first day of the present month. With the introduction of M-Script in version 1.2 of MRL, M-Script can be used to update and validate parameters, which allow a flexible use of report parameters.

To use M-Script for updating and validating parameters, you have to create a file containing the M-Script code. The file can declare two functions:

`updateParametersPre`

`updateParametersPost`

The first of these functions is used to update the default value of the parameters, i.e. to update the parameters just before they are presented to the user in the selection criteria. The latter function serves two purposes. It can be used to verify that the data entered by the user in the selection criteria satisfy certain constraints. For example, if a user is required to enter the name of an employee, we may want to verify that the name is in fact the name of an employee. Moreover, the function can be used to update the values entered by a user. For example, we may want to qualify the name with the employee's position.

Script interface

The functions `updateParametersPre` and `updateParametersPost` must be public and have a single argument. The argument is an M-Script object representing the parameters of the report:

```
{ parm1 : {    type : "string",
               title :       "Surname",
               basis :       "",
               mandatory :   true,
               solitary :    true,
               solitaryType: "string",
               hidden :      false,
               value :       "Alice"
          },
 parm2 :  {    type : "string",
               ...
               value : ""
          },
 reportInfo : {      report$name   : { ..., value : "Report1" },
                     report$title  : { ..., value : "Report no. 1" }, report$nameOfUser
                     : { ..., value : "Adam" }
               }
}
```

where `parm1`, `parm2` are the names of the parameters of the report. All parameters of the report are available, and all attributes of the parameters (except the name, which is already known) are available as well. Moreover, the implicitly defined parameters (the name and title of the report and the user name, see section "2.1.3: Queries" on page 9) are available in the property `reportInfo`. The use of the `solitaryType` attribute is somewhat technical. Whenever a parameter is solitary-, its `solitaryType`

attribute is equal to the type attribute specified in the report, otherwise it is equal to "string". This attribute is useful because the type of a `solitary-` parameter is "`string`". For example, if we have a report parameter `<parameter A type=date>`, then type is "`string`" and `solitaryType` is "`date`", because the parameter by default is `solitary-`.

The `updateParametersPre` and `updateParametersPost` functions must return an object whose properties are the names of the parameters that are to be updated, more precisely, it must be on the form

```
{       parm1 : {     mandatory : false,
                      hidden : true,
                      value  : "Bob" },
        parm2 : { mandatory : false, ... }
}
```

The object is used for updating the parameters. In this case, the parameter `parm1` will be hidden, but it will not be mandatory and its value will not be "`Alice`" but "`Bob`." Only the parameters and the attributes referred to in the object will be updated.

Only the `mandatory`, hidden and value attributes of a parameter can be updated. All parameters of the report can be updated except the implicitly defined parameters.

Only parameters declared in the report can be updated; it is thus not possible to declare new parameters in the script. In case updateParameterPost is used only for verification, and not for updating parameters, it may be declared a procedure instead of a function.

Verification and failure

The `updateParametersPost` function can be used to verify that data entered in the selection criteria is valid. Whenever data is invalid, a user must reenter data. In the script, this is controlled by calling one of the following functions:

`maconomy::fail(msg)`

`maconomy::failOnParameter(parm, msg)`

both of which expect string(s) as argument(s). The functions cause failure, meaning execution of the `updateParametersPost` function stops and the user is presented with an error message, `msg`, connected to the selection criteria. The `failOnParameter` function is distinguished from fail by setting focus to a field, `parm`, in the selection criteria.

Restrictions on M-Script use

The M-Script file must be installed on the server, see section “1.4: Installing reports” on page 4. To access the update functions, the file must declare a package. It is not possible to make use of the package version framework of M-Script. All the M-Script API functions:

- maconomy::sql*
- maconomy::mql*
- maconomy::dialog*
- maconomy::fail*

are available in report M-scripts.

The manuals “Maconomy M-Script Language Reference” and “Maconomy M-Script API Reference” contain a list of permitted contexts for each M-Script command.

Commands that are permitted in report M-scripts are marked with “All” or “MRL” in the “Context” section for each command.

Example: Pre-update and verification

Suppose we have a report that shows data about a company’s future business up to a given date. The report has the parameter

```
<parameter parmFutureDate type=date solitary+ />
```

We want the default value of this parameter to be one week from the present day. However, if a user changes the value of the parameter, we require that the entered date be in the future. This way we ensure the report shows engagements of the future. These constraints are met by the following script.

```
#version 14
package myReport(1.0.0);


public function updateParametersPre(parameters)
{
   return { parmFutureDate : { value : getdate() + 7 } };
}
public procedure updateParametersPost(parameters)
{
   var enteredDate = parameters.parmFutureDate.value;
   if (enteredDate <= getdate())
      maconomy::failOnParameter("parmFutureDate",
                                "Please enter a future date");
}
```

The `updateParametersPre` function updates the parameter parmFutureDate by setting its default value to one week from today. In the `updateParametersPost` procedure, we declare a variable `enteredDate`, which is equal to the value of the parameter `parmFutureDate`, hence equal to the date entered by the user in the selection criteria. We then verify whether the entered date is in the future: if not, `maconomy::failOnParameter` is called, which causes failure (stops execution of the script), meaning the user is presented with the error message and required to reenter a date.

Example: Verification and post-update

We have a report with the following parameter

```
<parameter parmArea type=real solitary+ />
```

However, we want to use its square root. In the script, we verify that the parameter is not negative, before returning its square root:

```
public function updateParametersPost(parameters)
```

```
{
   var parmArea = parameters.parmArea.value;
   if (parmArea < 0.0)
      maconomy::fail("Please enter a non-negative number"); else
      return { parmArea : { value : sqrt(parmArea) } };
}
```

Example: Conditionally hidden parameters

In the following example, we have a report with a single parameter

```
<parameter parmEmpName basis=Employee.Name1 />
```

The report is used primarily as a drill-down report (meaning that another report links to it). If a link is successfully established, `parmEmpName` is supplied a value, otherwise its value will be its default value (the empty string). When the parameter has a non-empty value, the user should not be able to change it (hence the parameter should be hidden), otherwise the user should be able to change it (hence the parameter should not be hidden).

```
public function updateParametersPre(parameters)
{
  if (parameters.parmEmpName.value != "")
     return { parmEmpName : { hidden : true } };
  else
     return { };
}
```

Example: Accessing the database

The following example illustrates how a report M-Script may access the database using the MQL API of M-Script. Assume a report with the parameter

```
<Parameter parmDate basis=JobEntry.DateOfEntry solitary+ >
```

We want the default value of this parameter to be the date of the user's latest entry to the `JobEntry` relation of the database. This requirement is met by the M-Script function below.

```
public function updateParametersPre(parameters) {
   try {
      var userName = parameters.reportInfo["report$nameOfUser"].value; var
      userEmpNumber = "0";
      var mqlQuery1 = 'MQLQUERY1';
                        <MQL 1.3>
                        MSELECT employeenumber FROM employee
                        WHERE name1 = mqlParmEmpName
                        USING PARAMETERS mqlParmEmpName : string
                      MQLQUERY1
```

```
     var mqlBind1 = { parameters:{ mqlParmEmpName: { value: userName }
} };
     var mqlResult1 = maconomy::mql(mqlQuery1, mqlBind1);
     var mqlRows1 = mqlResult1.mqlData.rows;

   if (sizeof(mqlRows1) == 1)
     userEmpNumber = mqlRows1[0].employeenumber.value;
   else
     maconomy::fail("Internal error: illegal name");
     var mqlQuery2 = 'MQLQUERY2';
                       <MQL 1.3>
                       MSELECT DateOfEntry FROM JobEntry
                       WHERE employeenumber = mqlParmUserEmpNumber
                       ORDER BY DateOfEntry DESC
                       USING PARAMETERS mqlParmUserEmpNumber : string
                     MQLQUERY2
     var mqlBind2 = { parameters:{ mqlParmUserEmpNumber: { value: userEmpNumber } } };
     var mqlResult2 = maconomy::mql(mqlQuery2, mqlBind2);
     var mqlRows2 = mqlResult2.mqlData.rows;

     if (sizeof(mqlRows2) == 0)
        return { parmDate : { value : getdate() } };
     else {
        var latestEntry = mqlRows2[0].DateOfEntry.value;
        return { parmDate : { value : latestEntry } };
     }
   }
   catch(e) { maconomy::fail("Internal error"); }
}
```

First, we obtain the name of the report user, which is available under the `reportInfo` property. Then a query, `mqlQuery1`, is defined, which is to retrieve the employee number of the user. The query uses a parameter to which the user's name is bound. If the query does not retrieve one row, then an internal error has occurred, since the user is not identified by his or hers name. Thereafter another query, `mqlQuery2`, is defined, which is to retrieve the date of the user's latest `JobEntry` entry. If no date is retrieved, the default value of `parmDate` is set to be today, otherwise the default value is the date of the latest entry.

Example: M-Script for a report collection

A report collection may be associated with at most a single report M-script, so all the members of a report collection share pre- and post-update functions. Nevertheless, we can customize the behavior of the functions for each member. This can be done using the implicitly defined parameter Report$name.

Suppose the report collection is that of the example in section 2.2.1 on page 17. We want the default value of the `MyParm` parameter to be 1 for first member of the collection and 2 for the second member:

```
public function updateParametersPre(parameters)
{
   var reportName = parameters.reportInfo["report$name"].value;

   if(reportName == "A::B::Member1")
      return { MyParm : { value : "1" } };
   else if(reportName == "A::B::Member2")
      return { MyParm : { value : "2" } };
   ...
}
```

We refer to the members by means of their full (qualified) name.

Getting Started with Universe Reports

This manual helps you to get started with Maconomy Universe Reports. After reading this manual and working through the examples, you should be able to build a report, install it, and run it.

Please note that a number of Maconomy add-ons are required to create and run universe reports.

Introduction to universe reports

The data in your Maconomy system is only valuable to the extent by which you are able to retrieve the data. This guide describes a simple, robust way of retrieving and presenting data from the Maconomy system.

Universe reports are reports designed to retrieve relevant information quickly and presenting it in the Portal. The information can be presented as HTML (for screen use) or PDF (for printing).

Universe reports offer powerful drill down functionality. When the report results are displayed, you can drill down to explore the figures in greater detail. The drill down is done by simply clicking the links presented on the report lines. A “drill down path” is displayed at the top of the report, which lets you drill back up again to higher reporting levels. For more information, see the “Maconomy Portal User’s Guide” in the Maconomy Reference Manual.

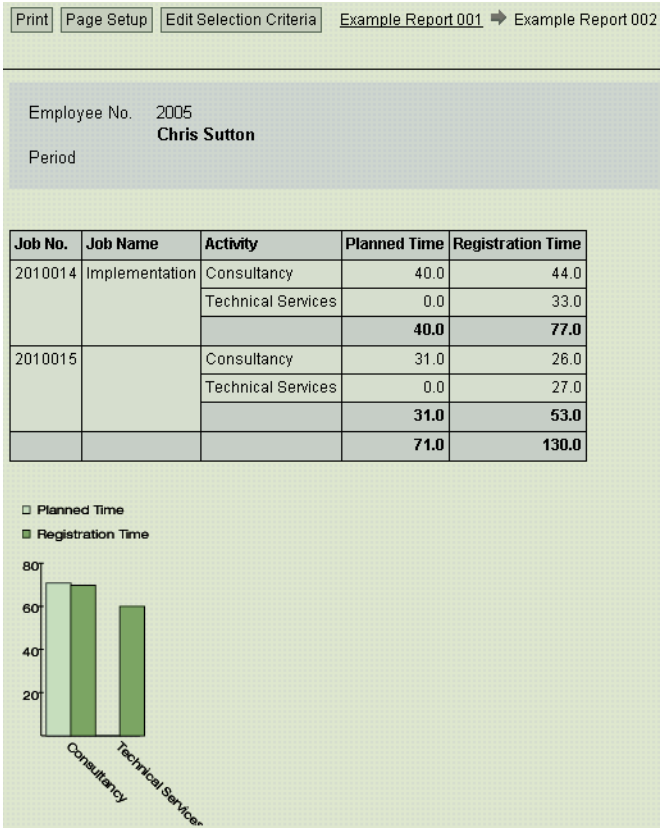
What’s in a universe?

A universe is a collection of fields from different relations in the database. When you refer to a universe instead of all the relations you need to access, Maconomy will think of the universe as a single relation with all the required fields available. For instance, you can create a “Job” universe with all information relevant to jobs. This makes reporting faster. It also makes reporting simpler, as we shall see.

However, a single relation in the database can also be defined as a universe. This is what we do in this manual.

In this manual, we will build two simple reports as an example of how to use the Maconomy universe reporting tools. Even though the reports are simple, developing them involves most of the techniques required for most universe reports.

After building the reports, we link them together. The final result will look like this:



Job No.	Job Name	Activity	Planned Time	Registration Time
2010014	Implementation	Consultancy	40.0	44.0
		Technical Services	0.0	33.0
			40.0	**77.0**
2010015		Consultancy	31.0	26.0
		Technical Services	0.0	27.0
			31.0	**53.0**
			71.0	**130.0**

The following sections take you through the creation of the report step by step. First, we will look at the basic concepts.

Concepts

A report is made up of three elements:

1. A query or a number of queries (data selection)
2. Parameters (data parameterization)
3. Layout (data presentation)

Queries

The query defines what data will be available for display in the report. The query is the mechanism by which data in the database are selected. In order to select the correct data, universe reports use a data selection language called MQL (Maconomy Query Language). Using this language, you can specify the universes (or relations) from which data should be selected, which fields in the universes should be selected, and a number of other criteria.

MQL is very similar to SQL, which is the most common data selection language. Then why use MQL? MQL has a number of advantages over SQL:

- MQL works the same way, regardless of which database is used – it is database- independent. The syntax or function of SQL differs slightly between databases, where the SQL expressions

used for an Oracle database is different from what is used for DB2 or MS SQL Server databases. This means that you can e.g. switch data- bases without having to rewrite all your reports, and you can transfer a good report from one Maconomy system to another without having to change it.

- MQL is specially designed to make use of Maconomy universes. This means that MQL is easier and faster than regular SQL.
- Maconomy validates the MQL statements for syntax, field typing, and semantic errors before they are issued to the database, making it easier to find errors in your statements.
- Finally, MQL offers features not available in standard SQL, such as data aggregation.

It is possible to have multiple MQL expressions in the same report.

The MQL query is included as part of a file written using MRL - Maconomy Reporting Language. The MRL file is the report file as such, in that it defines the name, query, and parameters of the report. We will return to the contents of this file later.

Parameters

Often you will want to be able to modify your report while running it ("at runtime"). For instance, you may want to limit the selection of jobs to jobs created by a specific employee or created for a specific range of customers.

In order to achieve this, you need to set *report parameters*. The parameters in effect define the report's interface by specifying which fields the user can use to limit the data selection. They can also control aspects of the report layout, e.g. by specifying that a report header should or should not be printed.

The parameters can be embedded in the report file to control data selection options or in the layout file (see below) to control layout selection options.

Layout

The layout defines what the report will look like when printed (either to the screen or to a printer). The layout is defined in a layout definition file by means of MPL – Maconomy Print Language. This is the same language, which is used for creating print layouts in the standard Maconomy client. It has a wealth of features, which are described in the manual "Printout Design using MPL". Features, which are only relevant for use in universe reports, are described in a section of the same manual.

The layout is specified in a separate file. The advantage of this is that you can change the layout without having to change the actual report file.

If you do not specify a layout, the report uses a standard, built-in layout. This layout is available for inspiration, i.e. you can copy the default layout and change it to suit your needs as described in "Viewing the default layout" on page 11.

Some of the advantages of MPL are the following:

- The MPL code is validated, and syntax and semantic errors can be found before the report is run.
- The layout specification is target independent. This means that you use the same layout file, regardless of whether you output the report in HTML format or in PDF.
- MPL controls the Maconomy formatting engine. The data selected from the data- base is passed through the formatting engine with the specifications defined in the layout file. Maconomy then produces the report as specified. The same formatting engine is used for producing HTML, PDF, SVG, etc.

Report example 1

In this section, we will create a report showing a list of employees. The example shown below is annotated, i.e. comments to the individual line numbers are added in the section “Annotations” below.

Note that the file is built using `tags`, similar to an HTML file. Some tags must have both an opening tag and a closing tag. The code between the tags then pertains to the command or instruction indicated by the tag, for example, `<Queries>...<End Queries>`.

Some tags are single tags with attributes, for example, `<MRL 1.0>`.

Create the report file

Open a text editor, and type the following:

```
1   <MRL 1.0>
2   <Report name=Examples::R001 title="Example Report 001">

3       <Parameter parmEmployeeNumbers type=string title="Employees" mandatory-
        solitary- >

4       <Queries>
5            MSELECT EmployeeNumber, Name1
6            FROM Employee
7            WHERE EmployeeNumber in parmEmployeeNumbers
8            ORDER BY EmployeeNumber
9       <End Queries>

10  <End Report>
```

Close the file, and save it as `R001.mrl`. Please note that if you are using Windows Notepad, you may have to enclose the filename in quotes when you save it to prevent Notepad from adding `.txt` to the filename.

Annotations

The following comments are intended to help you understand the example above. They are not a complete explanation of each MRL tag.

1. All MRL files start with an indication of the current MRL version number. This tells the Maconomy server what to expect.
2. Next, the single tag `<Report>` defines the current MRL file as a report definition file. Furthermore, the tag has two attributes:
   a. `name` specifies the name by which the Maconomy server will know the report. The server places the report in a separate namespace indicated by the colon- separated names. In this case, the report belongs to the “Examples” collection and is entitled “R001”.
   b. `title` is the external name of the report, i.e. the name printed at the top of the report when it is run.
   c. An additional attribute, `defaultLayout`, can be used to force the report to use the default layout, no matter if a valid layout file exists. This attribute is not used here. Leaving it out means that the default layout will not be used if a valid layout file exists.
3. The Parameter tag defines the parameter selection dialog of the report. When you run the report in the Portal, this is what you see first. In this example, a parameter section called `parmEmployeeNumbers` is defined, in which the user can select an employee or a range of employees. This parameter section is referenced by MQL in item 7 (see below).

The expected input into the parameter section is a string (`type=string`).

The label (external name) of the field in which the user selects a range of employees is "Employees" (`title="Employees"`).

The user is not forced to enter employees (`mandatory-`). To force the completion of a field, enter `mandatory+`.

The user can enter a single employee or a range (e.g. `1047..2109`) of employees (`solitary-`). If you only want the user to enter a single value, enter `solitary+`.

Please note that you can enter multiple `Parameter` tags to include several restriction fields.

4. The `Queries` tag specifies the beginning of the data selection part of the report file. This part contains MQL code, which will be explained below.
5. Select the values of the database fields `EmployeeNumber` and Name1. These fields contain the employee's number and full name, respectively.
6. The fields mentioned in item 6 are to be found in the `Employee` relation (or universe) of the Maconomy database.
7. However, do not select all the fields - only select those which match the criteria entered by the user in the parameter dialog section called `parmEmployeeNumbers`, which was defined in item 3 above. When the report is run, only those employees that are matched by the criteria entered by the user will be selected.
8. Sort the selected data in ascending order by employee number. This concludes the MQL data selection statement.

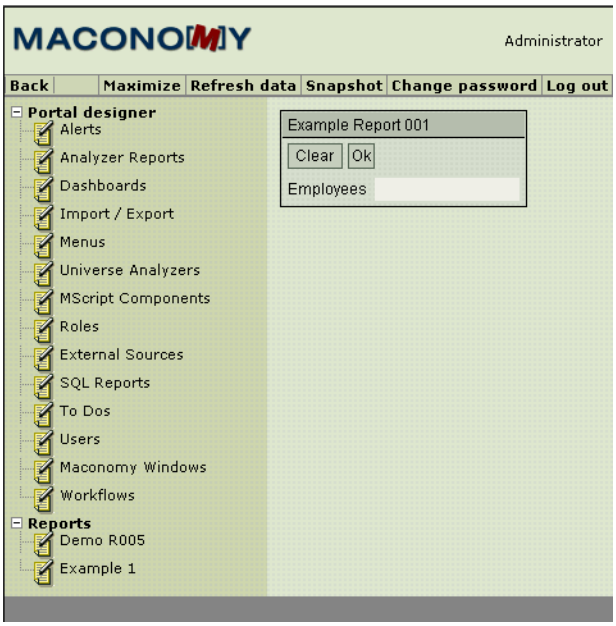
   If you want to add additional queries, you can repeat items 5-8 above with different values. Only remember to place a semicolon (;) to separate the MQL statements.
9. The `End Queries` tag specifies the end of the data selection part of the report file.
10. The `End Report` tag specifies the end of the report file. Any text or commands after this tag will be ignored.

Testing the report

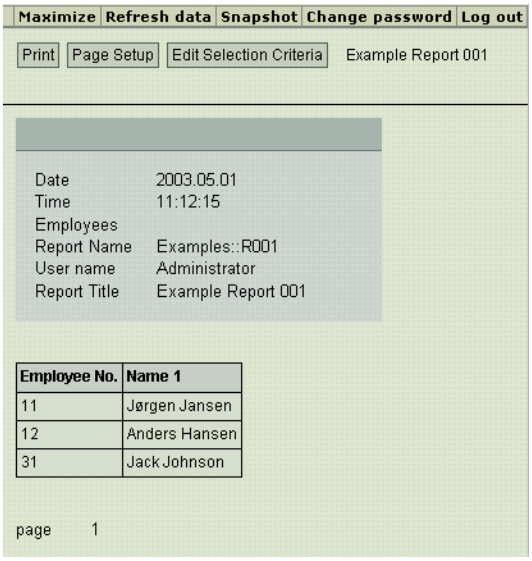
In order to test the report, the report must be installed on the server and made available in the Portal. This is a three-step process of stamping the report file, installing it on the Maconomy server, and granting access to it using the Portal Designer.

This process is described in the section "Enabling the report" on page 18. Note that you must have access to the Maconomy server and a file stamping tool to place the file in the Portal framework, and you must have Portal Designer rights to add the report to a Portal role.

After completing the steps outlined in the section "Enabling the report", you can run the report. It should look like this:



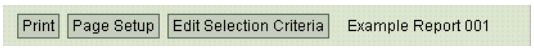
The report is launched showing the field defined in the Parameter section of the report file. After entering criteria in the field “Employees” (or leaving the field empty), clicking OK will produce a result similar to the following (the Portal menu will not be included in screen shots from now on):



Remember that we did not add a specific layout to this report, so the report uses the standard layout with a header supplying details about the report, the results in a standard table, and a page number.

Note that the external names of the fields selected in the MQL statement are used as headings in the result table. One effect of this is that if you are using dynamic translation, these field names will be displayed in the language, which you are set up to use.

With the buttons along the top of the report:



You can choose to print the report, set up page size and orientation, and return to the parameter selection dialog to rerun the report.

Adding a layout

Maconomy identifies a layout file by its name. If a layout file with the same name as the report file (except extension) is placed in the same folder as the report file, and the report file does not specifically say that it should use the default layout, then Maconomy will use that layout file to format the report.

Now we would like to remove the header information and the page number from the report.

Create the layout file

In the same folder as the report file (R001.mrl), create a file called R001.mpl using a text editor. Enter the following in the file:

```
1  <mpl 2>
2  <layout>
3  <paper fontsize=10>

4     <table cursor=query>
5          <fields>
6               *
7          <end fields>
8     <end table>

9  <end paper>
```

Save the file.

Annotations

The following comments are intended to help you understand the very simple example above. They are not a complete explanation of each MPL tag. The “MPL Language Reference” contains complete information.

1. All MPL files start with an indication of the current MPL version number. This tells the Maconomy server what to expect.

   Please note that the tags are case-sensitive – you must type all MPL tags using lowercase only.

2. Next, the single tag `<layout>` specifies that the purpose of the following code is to define a layout.

3. The paper tag defines the general options of the current layout, much like the `<BODY>` tag in HTML. The only attribute used here is `font size`, which is set to 10.

   Many of the attributes, which can be used in the paper tag, are not relevant for layouts intended for display in a browser. The browser automatically takes care of most of the formatting.

4. The `table` tag specifies the beginning of a table in the layout. The attribute `cursor` specifies the origin of the content of the table. The content is the result of an MQL query, and by default MQL queries have the name query. The table is laid out automatically by the columns and rows selected by the MQL query.

   It is possible to give the MQL query another name than query in the report file, using the as cursor attribute. For more information, please see the MQL documentation and the example in the section “Create the second report” on page 12.

5. The `fields` tag relates to the `table` tag above and tells Maconomy that the contents of the table begins here – the processing of the query result can begin.

6. The `*` means that all selected fields should be included in the table. Instead, you could have selected among the fields in the query by entering internal field names, preceded by a dot (`.`). You can also use this technique to reorder the columns from the query. For instance, if you write:

`.Name1 .EmployeeNumber` (note: no separator other than a space between the field names) instead of `*`, the same table is displayed in the result, but the columns are switched around.

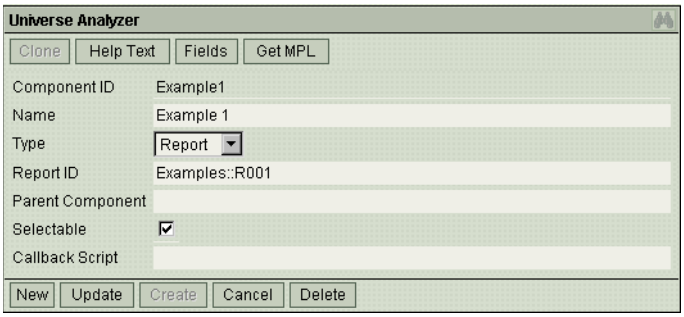
7. The `end fields` tag specifies the end of the fields of the table generation part of the layout file.
8. The `end table` tag specifies the end of the table generation part of the layout file. Note that you can generate multiple tables in a layout by repeating the table tags.
9. The `end paper` tag specifies the end of the layout file. Any text or commands after this tag will be ignored.

Now install the layout file on the server by following the steps outlined in the section “Enabling the report” on page 18.

Viewing the default layout

If you want to see how the default layout is designed, you can dump the contents of the default layout to a text file in the following way:

1. In the Maconomy Portal, click “Universe Analyzers” in the Portal menu.
2. Find the report in the list of components, and click its name.
3. This opens the “Universe Analyzer” edit window.



4. In this window, click the button “Get MPL”.
5. This opens a new browser window, which reveals the contents of the MPL layout file of the current universe report. You can now copy this information and paste it into a text editor for use as inspiration for your own layouts.

Note that it is always the default layout that is shown, not any customized layout which was installed together with the report.

Report example 2

You have now built a report, which shows a list of employees, and you have added a custom layout to the report.

We will now add a second report showing time sheet information about the individual employees. Then we will create a link between the first and the second reports, so that when you click on an employee in the first report, the second report will launch with information about that employee.

This example is rather more complex than the first one. It is used to generate a list of the jobs and activities on which an employee has been working within a range of dates.

Create the second report

Open a text editor, and type the following:

```
    <MRL 1.0>
    <Report name=Examples::R002 title="Example Report 002">
1       <Parameter parmEmployeeNumber basis=employee.employeenumber mandatory+
        solitary+ >
2       <Parameter parmPeriod type=date title="Period" mandatory- solitary-
        >

3       <Queries>
4            MSELECT [EmployeeNumber, Name1] as cursor
                 EmployeeCursor
             FROM Employee
             WHERE EmployeeNumber = parmEmployeeNumber;

5            MSELECT [Job.JobNumber, Job.JobName,
                 [Activity.ActivityText, PlanOfWeekSUM,
                 NumberOfWeekSUM] as cursor activityCursor]
                 as cursor JobCursor aggregate sum
6            FROM JobUniverse
7            WHERE Employee.EmployeeNumber = parmEmployeeNumber
8            AND TimeSheet.PeriodStart in parmPeriod ORDER BY Job.JobNumber;

9            MSELECT [Activity.ActivityText, PlanOfWeekSUM,
                 NumberOfWeekSUM] as cursor activityCursor
             FROM JobUniverse
             WHERE Employee.EmployeeNumber = parmEmployeeNumber AND
             TimeSheet.PeriodStart in parmPeriod
             ORDER BY Activity.ActivityText
10      <End Queries>

11  <End Report>
```

Close the file, and save it as `R002.mrl`.

Annotations

The annotations below do not repeat what you already know from the first report example.

1. This report contains two parameter tags. The attribute basis is a short way of specifying the `type` and `title` of the current field in the parameter selection window. By specifying a database field as the basis of the current parameter selection field, the selection field will inherit the database type and the external name of the field. In this case, it corresponds to typing `type=string title="Employee No."`.

   Note that this parameter selection field is both mandatory and solitary, i.e. the user must enter a single employee number.

2. This parameter selection field is of type `date`. This is a special Maconomy data type. The user will use this field to enter a range of dates for which he or she wants to see what the employee has spent time on.

3. In this report, the `Queries` section contains three MQL queries. Please note the use of the semicolon (`;`) to separate the queries.

4. The first query selects the employee number and the full name, as in the first example report. However, the default name of the query, `query`, is not used in this case, but the query is given the name `EmployeeCursor` using the syntax `as cursor`. By naming the queries in this way, the MPL layout file can distinguish between the three queries in the report file.

   Note that when `as cursor` is used, you must enclose the database field names in square brackets `[]`.

5. This `MSelect` statement is a query, which defines two blocks of data, each with a separate cursor name. The fields `ActivityText`, `PlanOfWeekSUM`, and `NumberOfWeekSUM` are defined as one cursor, and that cursor along with the fields `JobNumber` and `JobName` are combined into another cursor.

   This construct allows the report to iterate through all selected jobs (`jobCursor`), and for each job print the activities used on the job, the registered number of hours, and the week number (`activityCursor`). Furthermore, a sum of the fields within both the cursor `activityCursor` and `JobCursor` are generated using `aggregate sum`, adding a subtotal per job to the report and a grand total for all jobs.

   Please see page 17 for an illustration of the finished report.

6. The fields mentioned in the `MSelect` statement are all to be selected from the universe `JobUniverse`. This universe is a combination of several relations (`Job` and `Activity`), and it also contains calculated fields that do not exist in the database – `PlanOfWeekSUM` and `NumberOfWeekSUM`.
7. The fields are selected for the employee specified in item 1...
8. ...and for the dates specified in item 2.
9. This query defines the fields, which the report will use to build a bar chart to sum up the report.
10. This ends the query definition section of the report file.

Create the layout file

In order to display the second report in a nice way, you must design a layout. Open a text editor, and type the following layout:

```
   <mpl 2>
   <layout>
   <paper fontsize=10>
1  -- island on Employee
2     <repeating cursor=EmployeeCursor>
3        <island justification=left leftmargin=10pt topmargin=10pt
         bottommargin=10pt rightmargin=10pt>
           {
4        [.EmployeeNumber] .EmployeeNumber;
5        ""       .Name1:bold=true; [.parmPeriod]   .parmPeriod;
           }
         <end island>
6     <end repeating>
7     <skip 10pt>

8     <table JobCursor>
        <fields>
          .Job.JobNumber
          .Job.JobName
        <end fields>
        <table activityCursor>
          <fields>
            .Activity.ActivityText
            .PlanOfWeekSUM
            .NumberOfWeekSUM
          <end fields>
        <end table>
      <end table>
      <skip 10pt>
```

```
9     <barchart cursor=ActivityCursor height=140pt >
10        <legend>
          .Activity.ActivityText
         <end legend>
11        <fields>
            .PlanOfWeekSUM
            .NumberOfWeekSUM
          <end fields>
       <end barchart>
       <skip 10pt>

   <end paper>
```

Close the file, and save it as `R002.mpl`.

Annotations

The annotations below do not repeat what you already know from the first layout example.

1. You can enter one-line comments in MPL files by inserting a pair of hyphens before the comment.
2. `repeating cursor=EmployeeCursor` means that the current layout (paper) should be repeated for every item within the query cursor `EmployeeCursor`. Remember that in the report file, we specified that it should only be possible to select one employee, meaning that the current cursor will only run once. However, if you later change the parameter selection in the report file, the layout file is already prepared for this.
3. The island tag defines an area of text, just like an island in a window in the Maconomy client. In this case, the island is placed along the left edge of the paper area, with a margin of 10 points all around.
4. In the island, show the parameter selection criteria.
5. The field Name1 should be in a boldface font.
6. Ends the repeating part of the layout.
7. With the skip tag, you can jump a certain amount of points down the page.
8. This table tag is nested, i.e. it contains another table tag. The implication of this is that for every job in `JobCursor`, the fields in the `activityCursor` are printed. Note also that the attribute name query does not need to be present; i.e. `query=JobCursor` and simply `JobCursor` are equally valid.
9. The `barchart` tag draws a barchart based on the cursor `ActivityCursor`. The activity number is used as legend, and the planned hours per week and the week number are the bars in the chart. That way, you can easily compare the planned and the available hours per week.

Transfer the report file (and the layout file) to the server as outlined in the section “Enabling the report” on page 18. However, do not display this report in the Portal menu.

Linking the reports

Even though the second report could run as a stand-alone report, we would like to link it to the first report we created.

Linking reports is done in the layout file alone, and there is no need to rewrite anything in the reports as such.

Therefore, we need to open the first layout file, `R001.mpl`, in a text editor. Change the lines concerning the table tag to read the following:

```
<table cursor=query>
```

```
1           <fields>
2                .EmployeeNumber:report="Examples::R002"
                 :parameters=
                      [parmEmployeeNumber=.EmployeeNumber]
                 :reportsetup=[rpAction="drill"]
3       *
                 <end fields>
        <end table>
```

Save the files.

Annotations

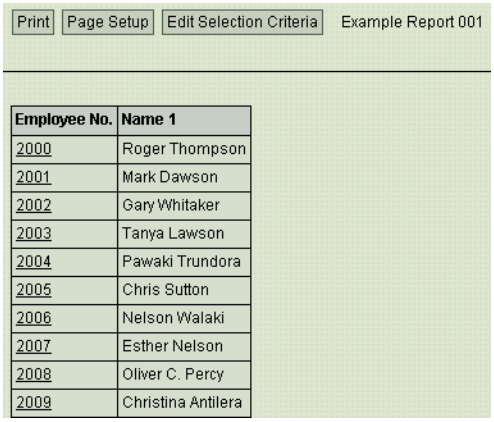
1. The `fields` tag has been expanded. Before, it simply showed an `*`, denoting that all fields should be displayed.
2. The field `.EmployeeNumber` (which is one of the fields selected by the MQL query in the report file) is specifically added to the layout, with a number of parameters:

   `report`: This attribute means that a report should be attached to the field name. In this case, we attach the second report we made, and which we called `Examples::R002`.

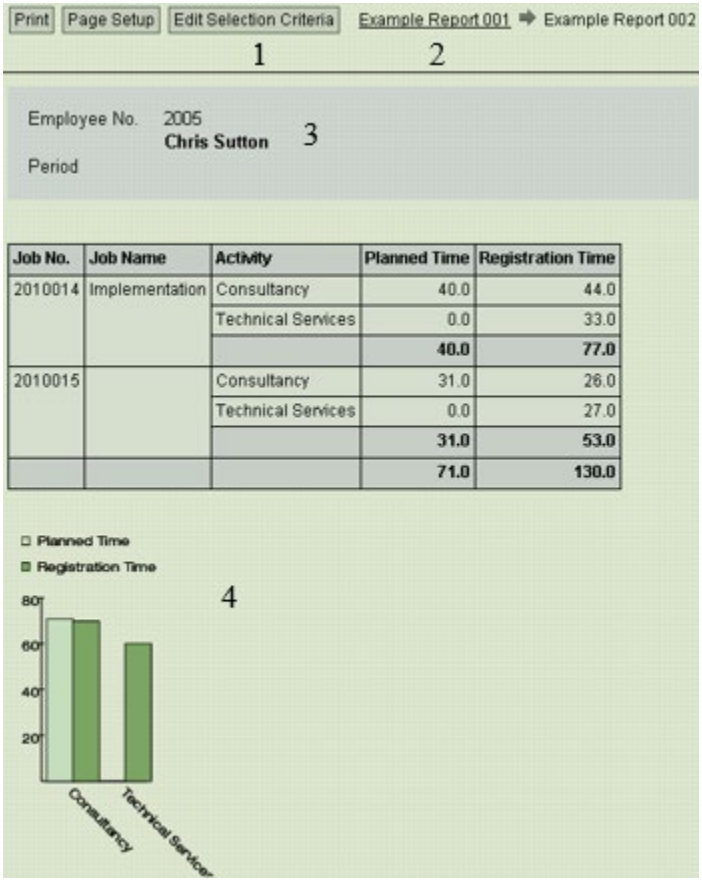
   `parameters`: This is a signal to the target report (`R002`) that its parameter section with the name `parmEmployeeNumber` should be prefilled with the value of this field (i.e. an employee number).

   `reportsetup`: With this attribute, you can specify an action to be performed automatically when the second report is launched. In this case, the report action “drill” (`[rpAction="drill"]`) is selected. This means that the second report is a drill down report – it is run instantly, without displaying the parameter selection options.
3. The remaining fields are then displayed as normal.

Now transfer the report and layout files to the server as outlined in the section “Enabling the report” on page 18. The report will now look like this (after choosing to see employees “200?” in the parameter selection window):



Note that the employee numbers are now hyperlinks. Clicking a link launches the second report, using the clicked employee as input. The report looks like this when clicking employee 2005:



Notes:

1. If you click “Edit Selection Criteria”, you can choose to see information about a certain period.
2. Maconomy automatically adds drill down/drill up functionality. Clicking “Employee Report 001” lets you select another employee.
3. This area shows information about the current employee.
4. The barchart shows the activity text and amounts of planned hours compared with registered hours. If nothing was registered, no barchart would be displayed.

Enabling the report

In order to install a report on the Maconomy server, a number of requirements must be met: You must have access to the Maconomy report stamping tool, your company must have a valid license file, you must have access to installing the file on the Maconomy server, and you must have access to the Portal Designer role in the Maconomy Portal. All these requirements are further described in the following.

Please note that this is not a manual for using either the stamper tool or the installer tool. You can find more information about these tools by exploring their help options and reading the “MStamper Reference” and “MBuilder Reference” manuals.

Both tools must be run on the computer hosting the Maconomy Server.

Report stamper (MStamper)

The Maconomy stamper tool is called `mstamper.exe` and is included in the Maconomy TPU (Tools Packing Unit). Using it requires the presence of a special license file, usually called `license.tac`, which you receive from Maconomy when purchasing the license to create universe reports.

When installing a recent TPU, the stamper tool is installed as well. On an NT machine, it is installed in the `MaconomyNT\bin` directory. This directory is included in your system path, which means that the stamper tool can be run from any directory on the computer.

After creating a report and a layout file, open a system prompt and navigate to the folder where the report and layout files are placed. Then run the stamper tool. To stamp the first report we made, you can type the following:

```
mstamper --readable --output r001.stamped.mrl r001.mrl
```

The report file R001.mrl is kept readable (as opposed to “scrambled”), and the stamped file is called `r001.stamped.mrl`.

Now do exactly the same for the layout file. The example below uses the short form of attributes:

```
mstamper -r -o r001.stamped.mpl r001.mpl
```

Both files are now stamped and ready to be installed on the server.

Report installer (MBuilder)

To install the file, use another tool included in the Maconony TPU, `mbuilder.exe`. This tool verifies the report files and installs the files correctly on the Maconomy server.

Example:

```
mbuilder --MaconomyDir c:\maconomyNT\MaconomyHomes\W_8_0
--Customization custom --Report r001.stamped.mrl
```

This command points out the location of the Maconomy server and the name of the report file stamped in the previous example. The command also installs any layout file with the same name as the report file except file extension, so the layout file `r001.stamped.mpl` from the example above will be installed at the same time. If you for some reason do not wish to do that, append the attribute `-MRLOnly` to the command line above.

The report is now ready to be made available in the Portal.

Portal Designer

After stamping and installing the report, it must be made available to the users in the Portal. A number of steps are involved, all of which require that you log into the Portal as a Portal Designer.

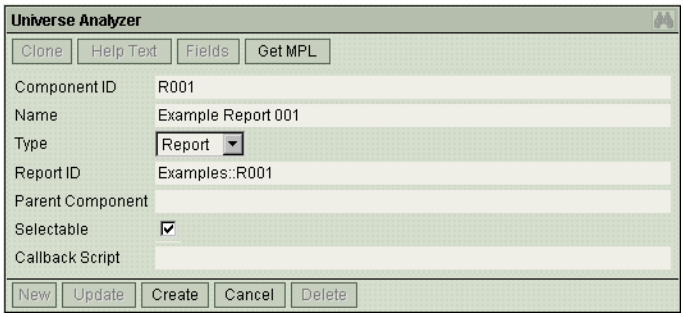
For more information about the Portal Designer, please see the chapter “Maconomy Portal Designer” in the Maconomy Reference Manual.

Include the Report

In order for the report to be visible in the Portal, you must create the report as a Portal component. To do so, open the “Universe Analyzers” list in the Portal Designer, and do the following:

1. Click “New” to create a new component.

2. Enter a component ID in the field “Component ID”. This is the component identifier, that is the name by which it is referenced in the Portal. You may choose any name as an ID (but do not use the backslash character).
3. Enter a name in the field “Name”. This is the external name for the report, e.g. in the Portal menu.
4. Select “Report” in the field “Type”.
5. Enter the path to the report in the field “Report ID”. This is the path as defined in the attribute name in the Report of the report file, e.g. `Examples::R001`.



The “Universe Analyzer” list might look as above.

Click “Create” to add the report as a report component.

The report is now made ready for inclusion in roles and menus in the Portal.

Assign the report to a role

Now you need to assign the report to one or more roles. Portal users assigned to those roles will have access to use the report.

Open the “Roles” list in the Portal Designer, and do the following:

1. Click the role to which you want to add access to the report. Please note that if you are using one of the Maconomy standard roles, you may need to clone the role first as described in the “Maconomy Portal Designer” manual.
2. Click the “Components” button to see a large window of all the components available in the Portal.
3. Scroll right to find the “Universe Analyzers” column.
4. Scroll down to find the component ID you just added, and mark the field.
5. Click “Update”.

Users assigned to the current role now have access to the report. However, the report cannot be found in any menu yet.

Include the report in a menu

If you want the report to be visible in a menu, you must assign the report component to a menu. You may not want to do this if the report is linked from another report, as is the case with report example 2 in this guide. That report should only be available after the user has selected an employee in the first report.

To assign a report component to a menu, open the “Roles” list in the Portal Designer, and do the following:

1. Click the role to which you want to add menu access to the report.
2. Click “Menu Items”.

3. Click the name of the menu to which you want to add the report for users of the current role, e.g. “Universe Reports”.
4. To insert the report on a new line, click the “Insert line”
5. Enter the report’s component ID (e.g. `R001`), and click “Update”.

The report has now been added to the “Universe Reports” menu for the role in question, and can be launched by any user with access to that role.

Where to go from here

Additional information can be found in the following manuals:

- “MRL Language Reference”
- “MStamper Reference”
- “MBuilder Reference”
- “Maconomy Portal Designer” (Maconomy Portal Configuration and Administration manual)
- “Printout Design using MPL” (including “MPL for Universe Reports”, in the System Administrator’s Guide)

Maconomy Report Designer

This handbook is a guide to Maconomy's Report Designer. The report designer is a program, which translates written reports into reports, which can be printed in Maconomy by the end user.

The language used for developing the reports is called RGL – Report Generator Language.

You need some knowledge of Maconomy's database in order to write these reports. It is also beneficial to have some previous experience from other report designers or general knowledge about programming.

Please note: The information in this document is primarily meant for maintaining existing RGL reports. The development of new RGL reports is discouraged because the RGL technology has been replaced by universe reporting as described in the other chapters of this book.

User's guide

This manual is divided into two parts: an overview and a reference section. The concepts described in this book are fully illustrated by examples to get you started writing simple reports and develop them as you go along and as the need arises.

This section is an overview, not a reference manual. Consult the reference section whenever you need a precise and comprehensive explanation of a concept.

This section of the handbook contains the following:

- Subsection 2, "Procedure", explains how to write and translate a report, describes which aids you have at hand etc.
- Subsection 3, "SQL", gives an introduction to SQL, which is the standard Maconomy uses to manipulate the information in the database.
- Subsection 4, "Levels", explains how reports are designed in levels, which are the building blocks of the reports. Your understanding of report levels is of vital importance to being able to write reports.
- Subsection 5, "Basic RGL elements", explains what is understood by variables and expressions, which is the mechanism used to make calculations.
- Subsection 6, "Selection criteria", explains how to design the window in which the user enters data for the report.
- Subsection 7, "Hints", has some advice and hints to make it easier to write (good) reports.

Procedure

This section explains the procedure used for making reports in Maconomy.

Using the Report Designer

When you write or modify a report, it is important to define precisely what is desired of the report. Not before you establish the purpose of the report can you begin writing it or adjusting it.

The function of the report is basically to retrieve data from the database and present it (or data calculated from it) in a way sensible for the user. It is therefore necessary to know which data is located in which relations in the database. Consult the Database Description, where you will find descriptions of all relations and their fields. You can make most adjustments to existing reports without this knowledge.

Entering reports

When you have found out where the desired data are situated in the database, or how to calculate them, you start to write the report. This is done by entering the report with a suitable editor (e.g. Microsoft Word® or TeachText). The result is a file, which contains the report. It is sensible to follow the convention that names of files containing reports end in `.rg`, e.g. `MyReport.rg`.

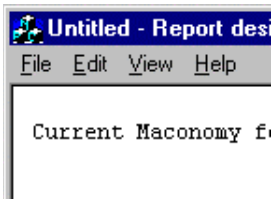
Translation

When you have finished entering the report, it must be translated. This is done with the report designer program.

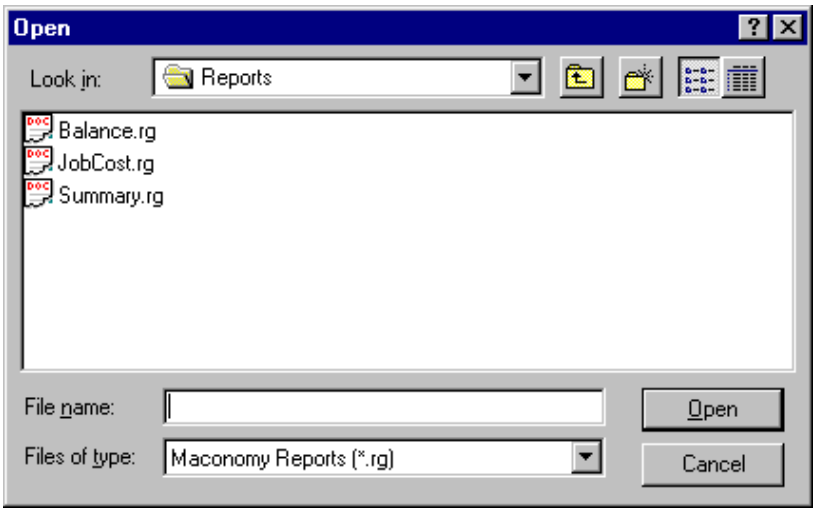
The Report Designer comes in two versions: one for Windows and one for the Macintosh. The translation of reports is a little different on the two platforms, and therefore the description of this procedure is divided into a Windows and a Macintosh section.

Windows

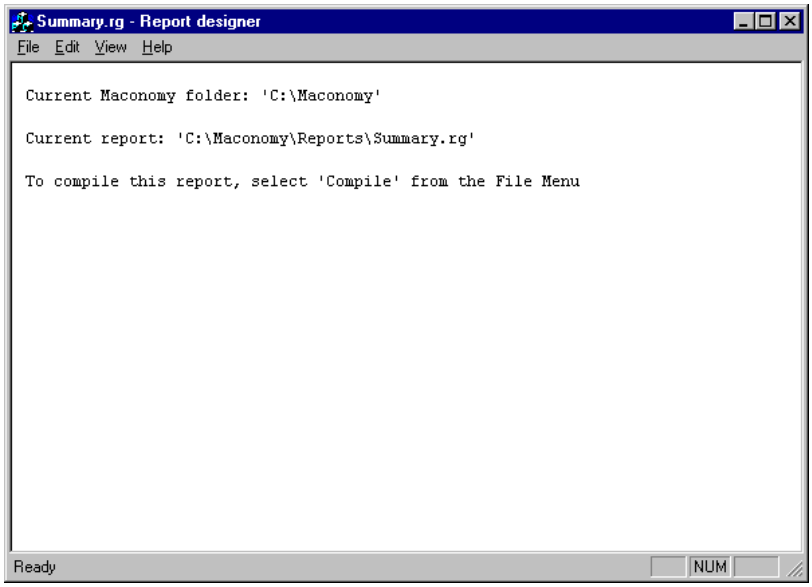
Start the Maconomy Report Designer for Windows. After you perhaps have selected a Maconomy folder, the following window is displayed:



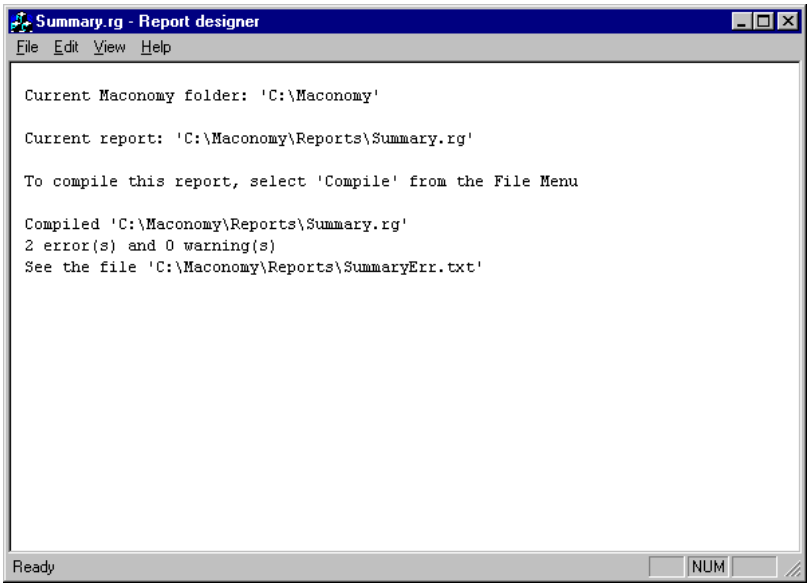
The window shows which Maconomy folder the Report Designer is working with. In order to be able to translate the report, select “Open...” from the File menu. This opens a window in which you choose which report to translate.



Here you can e.g. select the report `Summary.rg` by clicking the file name and then “Open”.

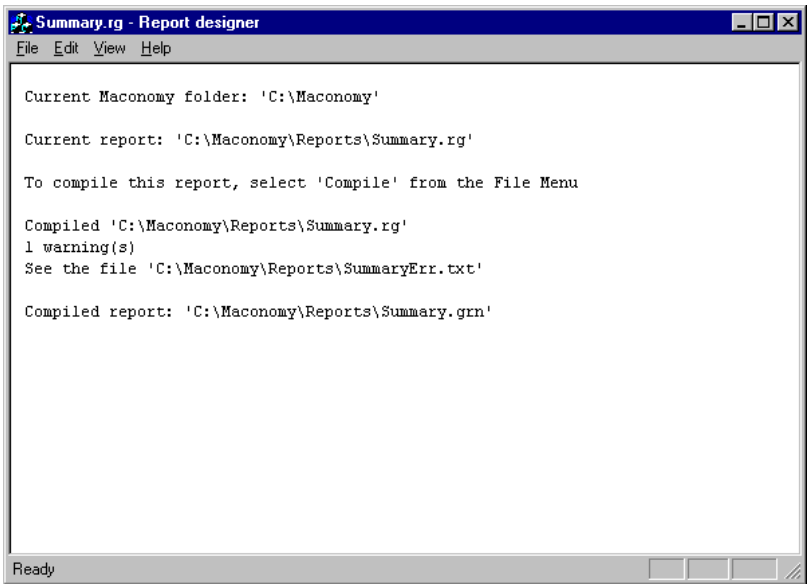


You can now translate the report by selecting “Compile” from the File menu.



In this example, the report is not correct (and can therefore not be translated). The Report Designer has created a file called “`SummaryErr.txt`” in the same folder as the source text of the report. This file contains a description and the location of the error. You must now correct the error and attempt another translation.

The Report Designer can also display a message such as this one:



Here, no actual errors were found in the report, and the report was translated. However, the Report Designer has found a suspicious-looking construct, which means that the report may not function, is the way it was intended. The warning is again described in the file `SummaryErr.txt`, and you are recommended to read the warning and decide if the report needs to be changed.

If the Report Designer finds no errors and no reason to issue a warning, a message such as the one below is shown:

The Report Designer has now created a file, which contains the report in a format, which enables Maconomy to process it. The file is called `Summary.grn` and is located in the same folder as the source text. The report can be renamed after translation.

Macintosh

Start the Maconomy Report Designer program. After perhaps selecting the Maconomy folder, select “Open...”. Note that from System 7 and up you can drag one or more reports to the Report Designer program, and the program will then translate them.

This opens a window in which you choose which report to translate.

The Report Designer will create a file as described above under 2.1.1, “Windows”, which may contain error messages. Follow the guidelines above to translate the report until it is translated correctly. The correctly translated file has the string `.grn` appended to its file name and is saved in the same folder as the report source text. The file can later be renamed.

Test

When this is done, the report must be tested. You do this by selecting the “Report...” function in Maconomy and then selecting the correctly translated file (e.g.

`MyReport.grn`). It is good practice to select “Print to screen” to save paper. If you discover that the report does not produce the expected results, you can adjust it and translate it again.

SQL

SQL is the language Maconomy uses to communicate with its database. SQL is the most widely used standard for database processing.

It is the use of SQL that ensures that you can change the database server without changing Maconomy's functionality.

The language in which you write reports is inspired by SQL. SQL introduces various concepts, some of which are described in this chapter and used to define reports.

Important concepts

Relations

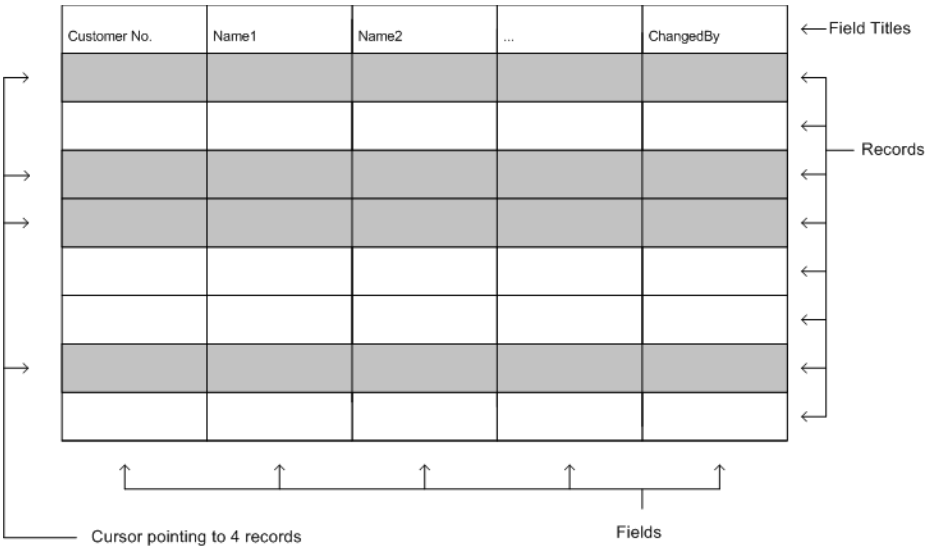
Relations are data tables

Relations are groups of data in the database. For the purpose of reports for Maconomy, we can define relations as tables of data. These tables consist of a fixed number of columns and a variable number of rows. Each row has data in each column. Each column has a name, e.g. “CustomerNumber” or “Name1”, which indicates which kind of data is listed.

Fields, Records

In SQL terminology, columns are called fields and rows are called records. This terminology is used from here on.

To recapitulate, a relation is a group of records, each with a number of named fields, which contain data. This is illustrated below.



Maconomy is designed from a number of different relations. There is a relation for customers, a relation for vendors etc. All the relations used by Maconomy are described in a separate document: Maconomy Database Description. This document lists the name of every relation in Maconomy as well as which field is contained in it.

Cursors

Access to individual records in a relation can only be obtained by `cursors`. These are conceptual pointers to the records in the relation to which the cursor belongs. The figure above illustrates a cursor pointing to four records in the relation.

Cursor declarations

You define a cursor using a cursor declaration. The cursor declaration is used to specify the relation from which you wish to read records, any conditions tied to the records and the order in which you wish to sort the records. Every cursor has a name.

Field selection

In the cursor declaration, you specify which fields you want access to. You write a list of field names, separating the names by commas. Normally all fields are selected, specified by the keyword `ALL`.

WHERE part

You can associate a condition to a cursor. The records you wish to retrieve with the cursor must meet your condition. A condition is specified in the cursor's optional `WHERE` part. If you do not specify a `WHERE` part, this is interpreted as a condition which is always true, meaning that all records are retrieved. The `WHERE` part is a logical expression. It is described later.

ORDER BY part

You can also specify how the records are to be sorted when they are retrieved. This is done using an `ORDER BY` part. The `ORDER BY` part consists of a list of field names in a decreasing order of priority. If you do not specify an `ORDER BY` part, the records will be sorted arbitrarily.

Examples

```
cursor CustomerCursor is
   select all from Customer
```

The cursor called `CustomerCursor` retrieves all records in the `Customer` relation. No sorting is done.

```
cursor CustomerCursor is
   Select all from Customer
   Where Name2 = "Jones"
```

`CustomerCursor` retrieves those records in the `Customer` relation, which have the value “Jones”. No sorting is done.

```
cursor CustomerCursor is
   Select all from Customer
   Where Name2 = "Jones"
   order by Name1
```

`CustomerCursor` retrieves the Jones records sorted by `Name1` – that is, “Jones, Andy” comes before “Jones, Bill”.

Levels

After being introduced to the necessary SQL concepts, you are now ready to use them to make simple reports. This chapter goes through the basic building blocks of your reports, the so-called levels.

Report levels

Essentially, a report consists of a number of levels. Normally, a cursor is assigned to a level. For every level, the program runs through and prints all the records pointed to by the cursor assigned.

Example

We will now write our first report: A report which will print all purchase lines from purchase order number “800001”:

```
-- A cursor to process purchase lines for
-- purchase order number 800001


cursor PurchaseLines is
   select all from purchaseLine
   where PurchaseOrderNumber = "800001"


-- For every record in purchase line with
-- order number 800001, print item number, item text
-- and quantity of ordered items. level 1 is Purchase lines
lines
   ItemNumber: 30mm Itemtext1: 80mm QuantityOrdered: 30mm
```

This report prints a number of lines, one line for every line in purchase order number 800001. The line contains the item number in a field width of 30 mm, then the item text in a field width of 80 mm, and finally the quantity ordered in a field width of 30 mm.

Hierarchy

Levels are sorted in a hierarchy. This hierarchy expresses both the sorting order and nesting. Every level has a number. This number defines the hierarchy. The rules are:

- The highest level must have level number 1.
- A level with level number N is nested in the previous level with level number N-1.
- The order in which the levels are sorted in the report is the order in which the records from the levels will be printed.
- Every time one record is printed on a level, the program prints all records on the next level (i.e. level number 1 higher). If there are several levels with the same level number, the records are printed in the order in which they are written in the report.

Example

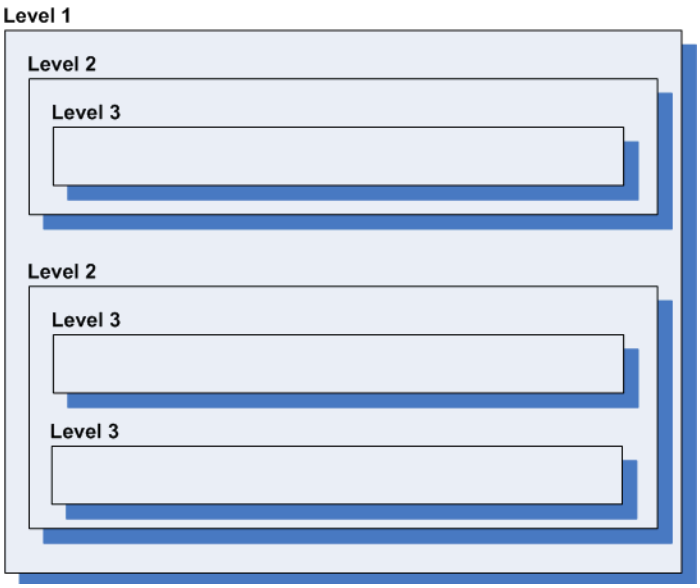
Study the following level structure (the structure is emphasized by indentation):

```
level 1 is C1
   level 2 is C2
     level 3 is C3
   level 2 is C4
     level 3 is C5
     level 3 is C6
```

The report is printed in the following order:

```
One record from C1 is processed.
   One record from C2 is processed.
     All records from C3 are processed.
   The next record from C2 is processed.
     All records from C3 are processed.
   :
   (No more records in C2)
   One record from C4 is processed.
     All records from C5 are processed.
     All records from C6 are processed.
   The next record from C4 is processed.
     All records from C5 are processed.
     All records from C6 are processed.
   :
   (No more records in C4)
The next record from C1 is processed.
:
(No more records in C1)
```

To get a clear picture of hierarchies, it is useful to draw the structure as boxes nested in each other. The example from above would look like this:



Example

We want to print all purchase orders and for each purchase order print all purchase lines pertaining to the purchase order. This is easily done by writing two levels: The first level (level 1) prints the purchase order, and level 2 prints all order lines. As you only want the purchase order lines, which pertain to the current purchase order, you make the `PurchaseLines` cursor select only the entries where the field `PurchaseOrderNumber` is the same as the `PurchaseOrderNumber` of the purchase order. If you do not do this, the report will print all purchase order lines for each purchase order.

```
-- Cursor declarations
cursor PurchaseHeaders is
   select all from PurchaseHeader
cursor PurchaseLines is
   select all from PurchaseLine where
   PurchaseOrderNumber = PurchaseHeaders.PurchaseOrderNumber
   order by LineNumber
-- Levels
level 1 is PurchaseHeaders
   lines
   PurchaseOrderNumber Name1: 80mm Name2: 80mm
level 2 is PurchaseLines
   lines
   " " -- Indentation
   ItemNumber : 30mm
   ItemText   : 80mm
   NumberOrdered     : 30mm
```

Each level can contain 3 parts: A header part, a lines part, and a trailer part.

Part	Description
Header	The header part is printed before the records are processed. It is normally used to print headers and to clear variables (described later).
Lines	Records are printed individually from the lines part.
Trailer	The trailer part is printed after the records have been processed. Typically, totals computed in the lines part are printed here.
Group by	You can specify a group by part to a level to which you have assigned a cursor. The group by part means that your data are grouped according to the values of specific fields.

Example

You need a list of your customers, sorted by country. For every country, the list should show the names of customers in this country, one on each line, indented by a few spaces. The report is written like this:

```
cursor Customers is
   select all from Customer
   order by Country
```

level 1 is Customers group by Country

```
header
   "Customers per
   Country":400:justification=center:size=15
lines
Country newline
```

level 2 is Customers

```
lines
   " ":50 Name1
```

The lines part in level 1 is generated once for each country and level 2 runs through all the customers in the given country. The report will show:

Customers per Country

```
USA
              InLiving Inc.
              Manson Mobile Homes
              Summer Domiciles Inc.
              DanFurniture Inc.
              Liva Leather Rooms Inc.
              Home Decorations Inc.
```

```
            Design House Group
            Flex Decorations
            Frank's Fancy Furniture

England
            Nice Homes Ltd.
            Water Beds Ltd.
```

Basic RGL elements

The previous sections described how you use levels to process records in the database. This, however, is not enough to write interesting reports. This requires several new concepts: variables, expressions, statements, and functions.

So far, we have introduced a method for processing records in relations (using cursors) and a method for printing data from these records. However, this is seldom enough to write useful reports. For this, you need a mechanism to make calculations.

Variables

Item	Description
Variables	Variables are named objects, which contain data of a specific kind, or type. Variables are introduced when they need to be used, and they “live” throughout the report. The variable type (string, integer (whole number), date etc.) is determined exclusively from the context in which it is used. This is why variables do not need to be declared.
Types	A given variable can contain data of just one type. This type is called the variable's type. The report designer deduces a variable's type from the function it performs. Therefore, you rarely need to concern yourself with variable types.
Assignments	Variables are assigned values in assignment statements. These statements must be written before and clearly separated from the actual texts in the header, lines or trailer parts. Assignment statements look like this: `VariableName := Expression` Variables can also be assigned values when they are used as arguments for certain predefined functions.
Standard values	When a variable's value is used before the variable has explicitly been assigned a value, the system will supply sensible standard values, determined from the variable's type. A date value, for example, has a value, which corresponds to the day the report is run, before the first explicit assignment is made for the variable.

Expressions

Item	Description
Expressions	Expressions are formulae, which represent a value when the report is run. Expressions can contain references to cursor fields, variables, and built-in functions (described later).

Example

```
Cursor CustomerCursor is
   select all from Customer
```

level 1 is `CustomerCursor`

```
header
   BalanceSum := 0.00 -- Initialize BalanceSum variable
   "Sum of Balances for Customers"
lines
   -- BalanceSum is increased by this customer's balance
   BalanceSum := BalanceSum + DebitBalanceBasics
trailer
   -- Print the sum
   BalanceSum
```

The above report sums up the balance for all customers. The `BalanceSum` variable is set to `0.00` in the header part of the level. This is only done once. The lines part is processed once for every record in the `CustomerCursor` cursor and the `BalanceSum` variable is updated for every record. The summed up value is printed in the trailer part.

Arithmetic expressions

Expressions, which return a number value, are called arithmetic expressions. These are built from sub-expressions, which are joined by the following:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
<	Less than
>	Greater than
<=	Less than or equal to

Operator	Description
>=	Greater than or equal to
=	Equal to
<>	Different from

The usual mathematical rules apply. Therefore, `1+2*3` has the value `7`, whereas `(1+2)*3` has the value `9`.

Logical expressions

Logical expressions can only have one of two values: `true` or `false`. This is the type of expression you use in a cursor's `WHERE` part to specify which records to retrieve. Logical expressions are constructed by logical sub-expressions and the following operators:

Operator	Description
`and`	Logical conjunction. Both sub-expressions must be true for the result to be true.
`or`	Logical disjunction. The result is true if one or both sub-expressions are true.
`not`	Logical negation. The result is false if the sub-expression is true.

Examples

```
-- True for customers for which customer number is within
-- the range FromCustomer to ToCustomer (inclusive).
Customer number >= FromCustomer and Customer number <= ToCustomer
-- Select only manual or subledger accounts AccountType = Manual or AccountType =
Subledger
-- True for customers not called Jones Name2 <> "Jones"
-- The above expression may also be written as not (Name2 = "Jones")
```

Statements

There are four types of statements in the Report Designer language (RGL): The assignment statement, the `NEWPAGE` statement, structured statements, and print statements.

Statement	Description
Assignment	An assignment statement causes the value on the right side of `:=` to be calculated and assigned to the variable on the left side. Example: `a := 2`
`NEWPAGE`	The `NEWPAGE` statement causes a page break in the report.
Structured statements	Structured statements enable loops and conditional execution. Three statements exist: `WHILE`, `FOR`, and `IF`.
Print statements	Print statements are used for determining the appearance of the report. You can specify fonts, justification, and font face.

Example

```
WHILE  i      <      50     AND    a [ i ]       <      37 DO
   b :=b +    2 * a [ i ]
   i := i + 1
END    WHILE
FOR    i              :=            3      i      <=     30     i := i + 1 DO
   SUM :=     SUM    +      a [ i ]
END FOR
IF     i      > 0    THEN
   a   :=     b / i
   ELSIF      i < 0  THEN
   a   :=     - b / i
ELSE
   a   :=     b
END    IF
```

Functions

Function arguments

Predefined functions are used in a variety of operations. There are five obvious groups of functions: *mathematical* functions, *date* functions, *time* functions, *finance* functions, and *miscellaneous* functions.

Functions are invoked by writing the symbol `@`, followed by the function's name. Every function has a number of arguments (in parentheses, separated by commas). Type and number of arguments is specific for each function. Some functions have an arbitrary number of arguments, but you can never state more than 30.

All functions and their application are described in detail under “21: Predefined functions” on page 60 in the reference section of this manual.

Selection criteria

The reports, which have been illustrated so far, all have the flaw that the user is unable to determine which customers, vendors etc. to include in the report. Therefore, the next step is to introduce the selection criteria dialog. This is the dialog the user sees when the report is run in Maconomy.

Selection criteria specification

Below is an example of the window the user sees when the report is generated. The user can enter values on the dotted lines. When the selection criteria have been specified, the user clicks the OK button (with the mouse or Return key).

The dialog works the way the user is used to from Maconomy. You cannot close the window or move to a new field if a given value is invalid.

Item	Description
Target	You write a specification of the selection criteria dialog in the report's `Target` specification. The (optional) part must be written before anything else.

Example

```
Target ("Customer")

  "Customer No. Range"      :20mm FromCustomer"-"ToCustomer ;
```

```
  "Customer Group"          :20mm CustomerGroup ;
  "Sales Rep."              :20mm SalesRep
```

Note that a semicolon separates the lines.

The text in parenthesis after the keyword `Target` is the dialog's title. If you do not specify a title, the window title is "Enter Values".

The names of the variables are written in the lines of the `Target` specification. These contain the values the user enters for the report. The variables are typically used in cursor declarations to limit the number of records to retrieve.

If you specify logical variables in the target group specification, the corresponding item will be check box. If the variable is an enumerated field (such as `CurrencyTypeType` etc.), the field will be a popup with the options that exist for that enumerated type.

After writing the variable, you can supply a standard value in parenthesis. The field is initialized with this value when the window is opened, and it is used for the report unless the user changes it.

Example

```
Target
  "From Year" FromYear (2005) ;
  "To Year"   ToYear (2006)
```

You can also state that the user cannot be allowed to leave the field blank. You do this by writing the keyword `MANDATORY` in parenthesis after the variable.

Hints

You now have some idea of what reports look like in Maconomy. There are probably a lot of unanswered questions – you will find some answers in the reference section. This chapter rounds up the experience you have gained from making your first reports, and it concludes this section with some helpful hints on how to design viable reports.

Style

Reports are not static structures, never changed once they have seen the light of day. On the contrary, it is only natural that reports are continually adjusted and extended to meet the requirements and wishes of the user. It is therefore important to always write reports in a style, which makes them easy to maintain.

You are recommended to supply indentation to illustrate the nesting of the levels. This is easy to do with an editing program suitable for the purpose, for example, the MPW editor for the Macintosh.

Example

```
level 1 is OrderCursor
header
lines
trailer
   level 2 is OrderLineCursor
   header
   lines
```

```
trailer
```

If you do not follow a specific standard (not necessarily the one in this book), you will encounter the same kind of problem as when you try to add numbers, where the digits are placed in the wrong columns.

Use comments as often as you like. Comments start with two dashes and extend to the end of the line on which they begin. Comments are also useful when you experiment. You can remove and insert printouts by removing and inserting "--".

Use long and descriptive names for variables and cursors. The maximum permitted length of names is 255 characters, so there is no need to economize.

Make sure that the report structure is correct before starting any calculations. Insert printouts to print interesting fields from the records in the report. This lets you monitor whether the cursor's `WHERE` parts are behaving, as they should.

Construct parameter dialogs to look like the ones used by Maconomy. Use the Geneva font (which is standard, if nothing else is specified), and use the dash, "-", to separate entry fields that appear on the same line. Make sure that the most frequently used values are the ones the user selects merely by clicking OK.

Report designer reference

This section describes the language used for report writing. The section is for reference only. The section "User's guide" above describes the procedure of making reports from Maconomy. The "Database Description" section of the Maconomy Reference lists the necessary database field names.

Audience

To make the most of this section, you must be familiar with the following concepts: cursor, relations, and levels. Consult the section "User's guide" above for understanding basic concepts.

Notation

In natural languages (e.g. English or Danish), there is a distinction between correct constructions (the syntax) and valid constructions (semantics). This handbook also distinguishes between syntax and semantics for the various constructions described.

The syntax is described in a notation, called Backus Naur Form (abbreviated to BNF), which is a precise notation used for describing the syntax. The following symbols are used:

Symbol	Description
`[…]`	Text between the two brackets can occur once or not at all.
`{…}`	Text between the two curly brackets can occur zero times or any number of times.
`<Name>`	< and > surround a symbol which you define (if it is written to the left of ::=) or a symbol which is defined elsewhere.
… | …	This symbol appears between two alternatives. You either get the value to the left or to the right of the | character.

Symbol	Description
::=	This symbol means “is defined as”. The element to the left of the symbol is equal to the expression to the right side of the symbol.

By placing two symbols together, the two symbols (or what they define) appear in the order the symbols are written.

Examples

```
<TargetLines> ::=
   <TargetLine> { ;  <TargetLine> }
```

Read as: “`TargetLines` is defined as a target line, followed by 0 or more occurrences of: semicolon, followed by a target line”, or, in other words, target lines is a row of target lines, separated by a semicolon. Target lines are defined elsewhere.

```
<digit> ::=
   0|1|2|3|4|5|6|7|8|9
```

Read as: A digit is defined as the character `0`, or the character `1`, or `2`, or `3`, or ... the character `9`.

Lexical items

Lexical items are the basic building blocks of report definitions. This chapter describes what symbols look like.

Characters

```
Letter> ::=
   A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|
   P|Q|R|S|T|U|V|W|X|Y|Z|
   a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|
   p|q|r|s|t|u|v|w|x|y|z|
<Digit> ::=
   0|1|2|3|4|5|6|7|8|9
<Underline> ::= _
<Other characters> ::=
   Any character you can enter on the computer
<Characters> ::=
   <Letter>| <digit> | <Underline> |
   <Other characters>
<Character sequence> ::= <Character> {<Character>}
```

A letter is alphabetical. A digit is a character from `0` to `9`. A character sequence is a sequence of at least one character.

The report designer never distinguishes between lower case and upper case.

Comments

A comment is your own explanatory text. The report designer ignores your comments.

Comments start with two dashes, “--”. All the characters following the dashes and up to the end of the line are ignored by the report designer. If your comments spread over more than one line you must start every comment line with two dashes, “--”.

```
<Comment> ::= -- <Character sequence>
```

Example

```
-- Report: BalanceTop10
-- 11 Dec. 1999
-- Version 1.1
```

Special symbols and reserved words

Special symbols and reserved words are symbols with fixed meanings. When you attempt to use them in a way, which does not harmonize with their fixed meaning, the report designer issues an error message.

The following individual characters are special symbols:

```
+ | - | * | / | \ | _ | < | >
. | : | ( | ) | " | ' | @ | ,
```

The following character pairs are special symbols:

```
-- | <= | >= | <> | :=
```

Below is the list of reserved words:

AMOUNT	ALL	AND
ARRAY	ASC	BOLD
BOOLEAN	BY	CENTER
CONDENSED	COUNT	CURSOR
DATE	DECLARATIONS	DESC
DIV	DO	ELSE
ELSIF	END	EXTENDED
FOR	FROM	GROUP
HAVING	HEADER	INTEGER
IS	ITALIC	JUSTIFICATION
LEFT	LEVEL	LINES
MANDATORY	MAX	MIN
MM	MOD	NEWLINE
NEWPAGE	NOT	OF
OR	ORDER	OUTLINE
PLAIN	PREFERENCES	PRINT
REAL	REPORTLANGUAGE	RIGHT
SELECT	SHADOW	STRING

```
STYLE          |      SUM                  |      TARGET
TIME           |      TRAILER              |      UNDERLINE
VAR            |      VARDECL              |      WHERE
WHILE
```

Literals

A literal is used to specify the value of a given type. For example, `12` is an integer (whole number) literal, and `12.22.2006` is a date literal.

Literals, syntax

```
<Literal> ::=
   <IntegerLiteral>  |      <RealLiteral>|
   <TimeLiteral>     | <DateLiteral>     |
   <StringLiteral>   | <Identifier>
<TimeLiteral>::= "<DigitSequence>:<DigitSequence>:<DigitSequence>"
<DateLiteral>::= "<DigitSequence>.<DigitSequence>.<DigitSequence>"
<DigitSequence> ::= <Digit> { <Digit> | <ThousandsSeparator>}
<ThousandsSeparator>       ::= ,
<Integer literal>   ::= <DigitSequence>
<RealLiteral>::=
   <IntegerLiteral>.<DigitSequence> [E <IntegerLiteral>] |
   <IntegerLiteral> [E <IntegerLiteral>]
<StringLiteral> ::= "<CharacterSequence>"
<Identifier> ::=    <CharacterSequence>
```

A literal can be an integer literal, a real number literal, a time literal, a date literal, a string literal or an identifier. Real number literals contain an integer literal and a decimal part, and an optional exponent part.

String literals are defined by quotation marks ("). If the string itself contains quotation marks, you must write them twice. That is, a string containing only quotation marks is written as: """".

Numerical literals (i.e. integer literals and real number literals) can contain one thousand separators. These punctuation marks (comma) can only occur three digits behind the last comma, decimal sign (period) or at the end of the number. That is, `1,250` is a valid integer literal with the value `1250`, whereas `12,50` is not valid. The decimal figure twelve and a half is written as `12.5`.

Literals, semantics

There is no syntactical difference between string literals, date literals and time literals; they are all character sequences surrounded by quotation marks. The report designer determines the type of the literal by looking at the contents of the string.

When a string has the form `tt:mm:ss`, and `tt` in the range between 0 and 23 and `mm` and `ss` are in the range 0 to 59, the string is regarded as a time value. If the string has the form `mm.dd.yy` or `mm.dd.yyyy` and `dd` lies in the range 1 to 31, `mm` lies in the range 1 to 12 and `yy` lies in the range 0 to 99 (or `yyyy` lies in the range 1941 to 2040), and if the date is correct with regard to leap years etc., the string is regarded as a date literal.

Logical truth values (`true` and `false`) are regarded as predefined constants.

String literals cannot be longer than 255 characters, and they cannot contain a page break.

Identifiers can have any length, but are only distinguished by the first 255 characters (letters, digits, and underscores). They cannot consist of digits alone.

Separators

Blanks, tabs, new line, and comments are regarded as separators. You can have any number of separators between two consecutive symbols.

Types

A value has just one type. The type determines which operations are permitted for the specific value. This subsection introduces types and the legal operations of their values.

An expression has just one type. The possible types are: INTEGER, REAL, AMOUNT, STRING, DATE, TIME, LOGIC and ENUMERATION. INTEGER, REAL and

AMOUNT types are called “numerical types”.

A literal’s type is specified by the literal. Therefore, an integer literal is an INTEGER, a date literal is a DATE, etc.

Integer type

Integers are numbers without a decimal part.

The following arithmetic operators yield integer results when applied to an integer argument:

Operator	Description
*	Multiplication
+	Addition
-	Subtraction
DIV	Integer division. For example, i1 DIV i2. Division by 0 is not allowed.
MOD	Modulus. Returns the remainder of i1 DIV i2. Division by 0 is not allowed. That is: (i1 DIV i2) * i2 + (i1 MOD i2) = i1

When evaluating expressions, both DIV and MOD take precedence over the multiplication operator. That is:

```
i1    DIV    i2    +     i3    =     ( i1   DIV    i2 )   +     i3
i1    +      i2    DIV   i3    =     i1     +      ( i2   DIV   i3 )
i1    MOD    i2    +     i3    =     ( i1   MOD    i2 )   +     i3
i1    +      i2    MOD   i3    =     i1     +      ( i2   MOD   i3 )
```

They associate left to right:

```
i1    DIV    i2    MOD   i3    =     ( i1   DIV    i2     )     MOD    i3
i1    MOD    i2    DIV   i3    =     ( i1   MOD    i2     )     DIV    i3
```

Note further:

```
( -i1 )      DIV i2 = i1   DIV    ( -i2 ) = - ( i1    DIV    i2 )
```

and:

```
( -i1 )      MOD i2 =      - ( i1 MOD i2 )
```

by convention.

```
i1    MOD    ( -i2 )       =      i1     MOD    i2
```

(however `i2 < 0` is not very meaningful).

Examples

```
5 DIV 3 = 1
2 DIV 3 = 0
5 MOD 3 = 2
```

The following operators (called relational operators) yield a logical value when they are applied to numerical values:

Operator	Description
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Different from

Integer values have to be within the range -2^{31} to $2^{31}-1$, i.e. between -2,147,483,648 and 2,147,483,647, inclusive.

The following attributes apply to integers:

Attribute	Result
'First	Returns -231 i.e. -2,147,483,648.
'Last	Returns 231-1 i.e. 2,147,483,647.

Real type

If one or both of the constituent expressions are REAL, the following operators give real results:

Operator	Description
*	Multiplication

Operator	Description
/	Division. The result is real, even if both arguments are integers.
+	Addition
-	Subtraction

The following relational operators return logical values when applied to numerical operands:

Operator	Description
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Different from

Real values lie in the range $-(2-2^{-23})*2^{127}$ to $(2-2^{-23})*2^{127}$, i.e. approx. $-3.402823872*10^{38}$ to $3.402823872*10^{38}$. The values are stored in IEEE 64 bit format.

The following attributes apply to real numbers:

Attribute	Result
'First	Returns $-(2-2^{-23})*2^{127}$, i.e. approx. $-3.402823872*10^{38}$
'Last	Returns $(2-2^{-23})*2^{127}$, i.e. approx. $3.402823872*10^{38}$

Amount type

Amounts always have exactly two decimals.

If one or both operands to the following operators are amounts, the result is an amount.

Operator	Description
*	Multiplication
/	Division
+	Addition
-	Subtraction

The following relational operators return logical values when applied to numerical operands:

Operator	Description
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Different from

Amount values must lie in the range $-2^{-63}-1$ to $-2^{-63}-1$, i.e. -92,233,720,368,547,758.07 to 92,233,720,368,547,758.07, inclusive.

The following attributes apply to amounts:

Attribute	Result
'First	Returns $-2^{-63}-1$, i.e. -92,233,720,368,547,758.07
'Last	Returns-2^{-63}, i.e. 92,233,720,368,547,758.07

String type

Strings are character sequences. The report designer does not interpret the characters in a string. A string can be empty, that is, have no characters. A string can only be as long as 255 characters.

The following relational operators return logical results when applied to string values:

Operator	Description
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Different from

Relational operators use a normal, lexicographic sorting order. The empty string, i.e. the one without any characters, is regarded as smaller than all other strings.

The following attributes apply to strings:

Attribute	Result
'First	Returns the empty string.
'Last	Returns a string, which is bigger than all other strings that can occur in Maconomy.

Date type

The smallest difference between two date values is one day.

The following relational operators return logical values when applied to dates:

Operator	Description
<	Before
>	After
>=	On the same date or after
<=	Before or on the same date
=	On the same date
<>	Not on the same date

The following attributes apply to dates:

Attribute	Result
'First	Returns January 1, 1904
'Last	Returns January 1, 2040

Time type

Time values denote hour, minute and seconds. The smallest difference between two time values is one second. Time values can express times between 00:00:00 (midnight) and 23:59:59 (one second before midnight).

The following relational operators yield logical results when applied to time values:

Operator	Description
<	Before
>	After
>=	On the same time or after

Operator	Description
<=	Before or on the same time
=	On the same time
<>	Not on the same time

The following attributes apply to time:

Attribute	Result
'First	Returns 00:00:00 (midnight)
'Last	Returns 23:59:59

Boolean type

Logical values can be `true` or `false`.

The following relational operators yield logical results when applied to logical values:

Operator	Description
AND	Logical conjunction
OR	Logical disjunctive
NOT	Logical negation
=	Equality
<>	Inequality

Truth values for logical operators are given below:

```
FALSE AND FALSE = FALSE
FALSE AND TRUE = FALSE
TRUE AND FALSE = FALSE
TRUE AND TRUE = TRUE
```

Both constituent values must be true to give a true result of an AND expression. In other words, if one or both constituent values are false, the result is false.

```
FALSE OR FALSE = FALSE
FALSE OR TRUE = TRUE
TRUE OR FALSE = TRUE
TRUE OR TRUE = TRUE
```

Both constituent values must be false to give a false result of an OR expression. In other words, if one or both values are true, the result is true.

```
NOT FALSE = TRUE
```

```
NOT TRUE = FALSE
```

If an expression is true, its negation (NOT) is false, and vice versa. The following attributes apply to logical types:

Attribute	Result
'First	Returns false
'Last	Returns true

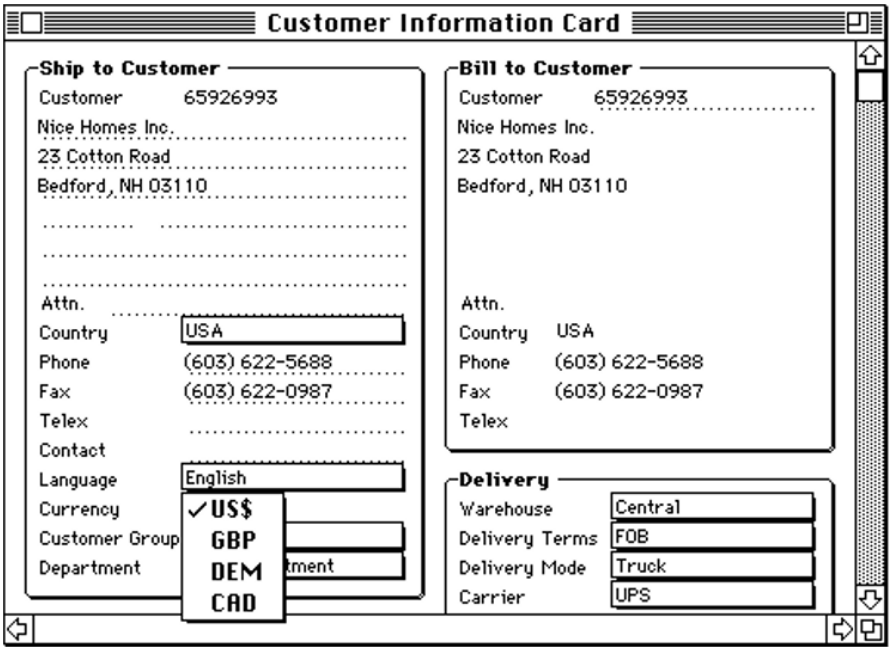
Enumeration types

Maconomy has several different enumeration types. These are represented to the user as popup fields. There is one enumeration type for every popup field. Popup fields are, for the purposes of this handbook, called enumeration literals.

There is an ordering of popup options. The first enumeration literal in a given enumeration type has an ordinal value of 0 and the following are numbered in consecutive order.

Some enumerations have fixed values, whereas others, called dynamic enumerations, have values which are maintained by the Maconomy user (in the window Popup Fields in the Set-Up module). Please note that because the Maconomy user can rename, adjust, and delete these dynamic enumeration literals freely, it is not safe to refer to them directly in your report.

Example



The `CurrencyType` type has the enumeration literals `US$`, `GBP`, `DEM`, and `CAD`. `US$` precedes all the others, and has an ordinal value of 0. `CAD` comes last and has an ordinal value of 3.

The following relational operators yield logical results when applied to enumerations:

Operator	Description
<	Before
>	After
>=	Same place or after
<=	Same place or before
=	Same place
<>	Not the same place

The following attributes apply to enumerations:

Attribute	Result
'First	Returns the enumeration value with an ordinal value of 0
'Last	Returns the enumeration value with the last ordinal value
‘Pos	Returns the ordinal value of the enumeration value
‘Val	Returns the enumeration value with the given ordinal value

Example

CurrencyType is the enumeration, which lists currency codes.

Currency Type	Value
CurrencyType 'pos(GBP)	Has the value 1
CurrencyType 'First	Has the value `US$`
CurrencyType 'Last	Has the value `CAD`
CurrencyType 'val(0)	Has the value `US$`

The values of the attributes are calculated when the report is generated, not at the time of its translation.

Array type

An array type variable is an indexed set of simple variables. An array variable must be declared in the variable declaration part of the report.

The syntax for specifying the type of an array variable declaration is:

```
<array specification> ::= ARRAY [ <lower bound> .. <upper bound> ] OF <simple type>
<lower bound> ::=    <integer literal>
<upper bound> ::=    <integer literal>
```

```
<simple type> ::=    INTEGER | AMOUNT | REAL | STRING |
                     BOOLEAN | DATE   | TIME | <enumeration type name>
```

Note the following:

- The indexing type must be INTEGER, and lower bound must be less than or equal to upper bound.
- The element type must be a simple type, i.e. arrays of arrays are not supported.
- Arrays are one-dimensional. This means that multi-dimensional arrays, declared as e.g. ARRAY [1 .. 8, 0 .. 100] OF STRING, are not supported.

An array element may be referenced in an expression by an indexed factor:

```
<factor>     ::=    ...    | <array variable>  [ <integer expression> ]
```

Similarly, an array element may be assigned a value in an assignment statement:

```
<assignment variable>      ::=    <simple variable |
<array variable> [ <integer expression> ]
```

Note that assignment to an entire array is not supported.

Examples

```
var
a      :      ARRAY  [      10 .. 99      ]      OF REAL
.
.
.
level 1 header i := 5
a [ 1 ]       :=     10.000
a [ i + 1 ]   := a [ 1 ]    +      4
```

Variables

Variables are data items whose values are maintained while the report is being generated. Variables are mainly assigned values using assignment statements.

Variables, syntax

```
<Variable> ::=
   <Identifier>
```

The name of the variable is chosen by the report writer. The name should be chosen carefully to match the function the variable has in the report. The maximum permissible length of a variable is 255 characters.

Variables, semantics

A variable is an item, which has a certain given value when the report is being generated. This value is called the variable's type.

All variables are global, which means that the variable can be used throughout the report, and that it exists throughout the time the report is being generated. Variables are defined when they are used for the first time. User-defined variables obtain their values in assignment statements, or as output parameters in finance functions.

The variable's type can be assigned in two ways: It can be declared in the beginning of the report, or it can be deduced from the context in which the variable is used.

Declared variable types

The type of the variables used in the report can be declared in the beginning of the report, before selection criteria specifications and cursor declarations. The syntax of the variable declaration is as follows:

```
<variable declaration part>      ::=    <empty>      |      VAR    {      <variable
declaration> }

<variable declaration> ::= <identifier> : <type> [ := <literal> ]

<type> ::=    INTEGER      |      AMOUNT |      REAL    |      STRING |

              BOOLEAN      |      DATE    |      TIME    |

              <enumeration type name>    | <array specification>
```

The variable may be assigned an initial value by appending `:= <literal>` to the `<type>`. The literal must follow the syntax rules for literals of the specified type. Enumeration type variables cannot be assigned an initial value.

When a variable has been declared, it cannot later be deduced. This means that if a declared variable appears out of context according to its declared type, no other type will be deduced. Instead, an error message will appear.

Example

```
VAR
    a1  :      AMOUNT := 1.00
    a2  :      AMOUNT
    r1  :      REAL := 1.000 r2    :      REAL
   .
   .
   .
LEVEL 1 HEADER
    a2  :=     r1
    r2  :=     a1
```

The four variables will preserve their declared types, regardless of the assignments.

As the default – and to ensure backward compatibility – variable declarations are optional. However, variable declarations may be set to mandatory by preceding the variable declaration part by a preference setting:

```
PREFERENCES   VARDECL MANDATORY
```

Deduced from context

When, for example, a variable is used in a place where the only meaningful value is a date, the variable is tied to a date value. The report designer issues an error message if you specify contradictory demands on the variable's type. If the report designer cannot deduce the variable type (from its context), it alerts

you of the problem and assumes that the variable has the string value. If there is a risk of this happening, you are recommended to type declare the variable at the beginning of the report.

Example

```
-- Here the BalanceSum variable becomes an
-- amount type, because the right side of the
-- assignment is an amount expression.
BalanceSum := 0.00
```

This is to illustrate the context-derived variable typing.

Predefined variables

Besides the variables inserted by the report writer, there are variables defined by Maconomy, the predefined variables. These are given a value when the report is started.

You can use the values of the following predefined variables in the same way as you use user-defined variables. However, you cannot assign values to the predefined variables; their values are maintained exclusively by Maconomy.

Predefined Variable	Type	Contains
TodaysDate / DagsDato	STRING	The date a report is started – as a string, not as date.
TheTime / Tidspunkt	STRING	The time of day a report is started – as a string, not as a date.
NameOfUser / Brugernavn	STRING	The name of the computer (the name shown in the Access Control window when you start Maconomy).

For more information about language dependence in the report, please see the subsection “Report structure”.

Variables for output of tab separated files

The format of tab separated output files can be controlled by a number of global variables. These variables are predefined, but are different from the other predefined variables in that their value can be changed. The following variables exist:

Variable and Type	Application

`TabFileFieldDelimiter : STRING`	`-- Specifies the field separator for the output file. The value must be one character. Special characters are specified in the following way:`
	`\t  Tab Character`
	`\\  \`

Variable and Type	Application
	\n New Line \xYY The ASCII character YY, where YY is a hexadecimal value.
TabFileDOSFormat : BOOLEAN	Only relevant for Windows. If true, output will be in the DOS character set, instead of the Windows character set (the character sets differ e.g. for the special language characters of Danish).
DontGeneratePrint : BOOLEAN	If true, no print will be generated.
DontGenerateTabFile : BOOLEAN	If true, no Tab separated file will be generated.

To have effect, these variables must either be assigned an initial value in the variable declarations part or appear in the target specification of the report, and they should not be assigned later.

Standard values

If a variable is used before it has been assigned a value (e.g. in an assignment statement), the report designer uses a standard value, determined by the variable's type.

The standard values are:

Type	Standard Value
INTEGER	0
REAL	0.0
AMOUNT	0.00
LOGICAL	false
TIME	Now
DATE	Today

Now and Today refer to the time and date the report is started.

Statements

There are four kinds of statements in the report designer: The assignment statement, the NEWPAGE statement, structured statements, and print statements.

Statements, syntax

```
<Statement> ::=
<Assignment Statement> | <NewPage Statement> |
```

```
<Structured Statement> | <Print Statement>
<AssignmentStatement> ::=
<Variable> := <Expression>
<NewPageStatement> ::= NEWPAGE
<Structured Statement> ::=
<While Statement> | <For Statement> | <If Statement>
```

An assignment statement consists of a variable followed by the assignment operator (`:=`) followed by an expression.

A `NEWPAGE` statement consists of the reserved word `NEWPAGE`.

The syntax of the structured statements (`WHILE`, `FOR`, and `IF`) and print statements is further defined below.

Statements, semantics

An assignment statement means that the value to the right of the := is calculated and assigned to the variable on the left. You cannot assign values to predefined variables.

A NEWPAGE statement causes a page break in the report.

Structured statements enable loops and conditional execution. Three statements exist: WHILE, FOR and IF.

Structured statements

While Statements

The syntax for the `WHILE` statement is:

```
<while statement>   ::=          WHILE  <boolean expression> DO    { <statement> }
      END    WHILE
```

As long as the `<boolean expression>` evaluates to `true`, the statements in the list `{ <statement> }` are executed.

For Statements

The syntax for the `FOR` statement is:

```
<for statement> ::= FOR <assignment statement 1> <boolean expression>
   <assignment statement 2>DO
   {   <statement>   }      END FOR
```

This is equivalent to:

```
   <assignment statement 1>
   WHILE      <boolean expression>       DO
   {   <statement>   }
   <assignment statement 2>
   END WHILE
```

The elements in the head of a FOR statement may be separated by ;:

```
<for statement>     ::=    FOR <assignment statement 1> ; <boolean expression> ;
    <assignment statement 2>      DO
    {  <statement>   }      END FOR
```

If Statements

The syntax for the `IF` statement is:

```
<if statement> := IF <boolean expression>      THEN   {      <statement>   }
  {    ELSIF  <boolean expression>
     THEN    {      <statement>   }      }
  [    ELSE   {      <statement>   }
  END  IF
```

The `<boolean expressions>` are evaluated in sequence. For the first expression that evaluates to `true`, the corresponding statement list is executed; none of the other statement lists are executed. If all expressions evaluate to `false`, the statement list after `ELSE` is executed (if any).

Example

```
  WHILE      i      <      50     AND    a [ i ]       <      37 DO
     b :=    b +    2 *    a [ i ]
     i := i + 1
  END WHILE
  FOR i             :=           3      i      <=     30     i := i + 1 DO
     SUM     :=     SUM    +      a [ i ]
  END FOR
  IF  i      > 0    THEN
     a :=    b / i
     ELSIF   i < 0  THEN
        a    :=     - b / i
  ELSE
     a :=    b
  END IF
```

Structured statements are primarily intended for calculations, not for output generation. If you use structured statements for generating output anyway, note the following:

`WHILE` and `FOR` statements should not be used for repeated output of elements on the same line as e.g. in:

```
  FOR  i := 1;  i <= 10;  i := i + 1  DO  a [ i ] :20  END FOR
```

This will not work as intended, while the following will work:

```
  FOR  i := 1;  i <= 10;  i := i + 1 DO
    i :20    a [ i ] :80   newline
  END FOR
```

After execution of a structured statement that generates output, the output line position is undefined, e.g.

```
  IF         a < b THEN    a :50  ELSE b :50    END IF
  c    :120
```

Here `c` will not be printed properly. Instead execute a 'newline' before printing c (which will then be printed at the beginning of a new line).

Print statements

Print statements are specifications of the printed report. You choose which values are to print and specify the format of the printed value (center alignment, bold text etc.).

The report definition language of the report has various print statements, which are used to define the appearance of the report.

Print Statements, syntax

```
<PrintStatement> ::=
   <PrintValue> <Attributes>      |
   NEWLINE                        |
   FONT <CharacterSequence>       |
   SIZE <IntegerLiteral>          |
   JUSTIFICATION <Alignment>
<PrintValue> ::=
   <Identifier>                   |
   <Literal>                      |
   <Identifier>.<Identifier>
<Attributes>    ::=
   { : <Attribute> }
<Attribute>    ::=
   <WidthSpecification>           |
   <FontSpecification>            |
   <SizeSpecification>            |
   <StyleSpecification>           |
   <AlignmentSpecification>
<WidthSpecification>  ::=
   [ WIDTH = ] <IntegerLiteral> [ mm ]
<FontSpecification> ::=
   FONT = <CharacterSequence>
<SizeSpecification>      ::=
   SIZE = <IntegerLiteral>
<StyleSpecification>       ::=
   STYLE = <StyleListe>
<StyleList>  ::=
   <Style> { , <Style> }
<Style> ::=
   PLAIN      | BOLD        | ITALIC      | UNDERLINE |
   OUTLINE    | SHADOW      | CONDENSED   | EXTENDED
<AlignmentSpecification>   ::=
   JUSTIFICATION = <Alignment>
<Alignment> ::= LEFT | RIGHT | CENTER
```

From a syntax perspective, the print statements are seen as literals, variables, or cursor fields, sometimes followed by a list of attributes, which determine the appearance of the printout. All attribute specifications are preceded by a colon. You can assign several attributes to every print value. The style lists consist of named styles, separated by a comma. You can leave out `WIDTH` = in width specifications.

Print statements, semantics

A print statement has the value, which is going to be printed. The list of attributes has specifications of what the printout is going to look like.

For every print statement, you can specify the width of the field in which the value is going to be printed (`WIDTH`), the alignment within the field (`JUSTIFICATION`), which font to use (`FONT`), which character size (`SIZE`) and which style to apply (`STYLE`).

`NEWLINE` is used to print an empty line. Note, the report designer inserts a new line automatically after the `HEADER`, `LINES`, and `TRAILER` of the level.

A global specification of font, size, and alignment is applied repeatedly until you enter a new global specification. Attributes, which are assigned to individual print items, always have higher priority than the global values.

The character sequence after `FONT` must be the name of one of the fonts on the computer on which the report is generated. On a Macintosh, you can usually use `Geneva`, `Courier`, `Helvetica`, `Palatino`, and `Chicago`, all of which are standard fonts. If there is no font specification in the report definition, the program uses `Geneva` by default. On a Windows computer, you can usually use `Arial`, `Times New Roman`, `Courier New`. Default is `Arial`.

Font size is measured in points. If nothing else is specified, the program uses font size `9`.

Field width is measured either in mm (1/1000 meter) or in points. There are 72 points to an English inch. The highest resolution is therefore 25.4 / 72, corresponding to 0.35 mm/point.

If you have not specified a width, the program uses 100 points, which is approx. 35 mm. If you enter a literal without explicit width specification, the program uses a value, which is precisely large enough to contain the value with its attributes.

If nothing else is stated, the program applies left alignment.

The styles you can use are: `PLAIN` (none of the others), `BOLD`, `ITALIC`, `UNDERLINE`, `OUTLINE`, `SHADOW`, `CONDENSED`, and `EXTENDED`. Some of these styles only apply to the Macintosh. You combine styles by separating them by commas.

Example

```
"The rain in Spain stays mainly in the plains":Style=bold :Size=14
```

is printed as

The rain in Spain stays mainly in the plains

The three print items

```
"> " "This is a heading":140 " <"
```

are printed as

> This is a heading <

```
"> " "This is a heading" :
   Justification = Center
   :140 " <"
```

is printed as

> This is a heading <

```
"> " "This is a heading" :
   Justification = Right
   :140 " <"
```

is printed as

> This is a heading <

```
"Italic and Underline" :Style=underline,italic
```

is printed as

Italic and Underline

```
"Size 6 " : Size =6 newline
"Size 9 " : Size =9 newline
"Size 12 ": Size =12 newline
"Size 15 ": Size =15 newline
"Size 18 ": Size =18 newline
```

is printed as

Size 6

Size 9

Size 12

Size 15

Size 18

Expressions

An expression is a formula, which expresses a value when the report is generated.

Expressions, syntax

```
<Expression> ::=
   <logical term> { OR <logical term> }
<logical term> ::=
   <logical factor> { AND <logical factor> }
<logical factor> ::=
   [ NOT ] <Simple logical expression>
<Simple logical expression> ::=
   <Simple expression>
   [<Relational operator> <Simple expression>]
<Relational operator>::=
   = | ¦ | < | □   | > | □
<Simple expression> ::=
   [ <Additive operator> ] <Term>
   { <Additive operator> <Term> }
<Additive operator>::=
   + | -
<Term> ::=
   <Factor> { <Multiplication operator> <Factor> }
<Multiplication operator>::=
   * | /
<Factor> ::=
<Literal>     |
   <Function call>   |
   <Attribute Spec> |
   ( <Expression> )
<Function call>::=
   @ <Identifier> ( <Expression list> )
```

```
<Attribute Spec> ::=
   <Identifier> ' <Identifier>
<Expression list> ::=
   <Expression > { , <Expression > } |
```

The usual rules for operator priority apply: Hence, 1 + 2 * 3 is 1 + (2*3). Brackets are used to change the order in which the expressions are evaluated.

Function calls consist of the symbol @, followed by the name of the function, followed by a parameter list in parentheses.

Expressions, semantics

The same limitations apply to expressions as to other types. Arithmetic operations can be applied to any numerical type, and numerical types can be mixed within arithmetic expressions.

If an error occurs during the evaluation of an expression value (e.g. by dividing by 0), the result is an undefined value. Such a value is written as `NaN` (abbreviation of Not A Number).

Example

```
(1 + tax/100) * Price
```

Calculates the price with tax.

```
CustomerNumber ≥  From and CustomerNumber ≤ To
```

This expression is `true` if the `CustomerNumber` is within the range between the values `To` and `From`, inclusive.

```
@AddDays (UndefinedDate, 1)
```

Calculates the date tomorrow. Note the use of the standard value for an undefined variable; this variable is not assigned a value anywhere in the report and consequently contains its default value, namely today.

```
@IF ( SoldTotal <> 0,
      SoldUS * 100 / SoldTotal,
      0.0)
```

Calculates how large a percentage of your total sales were made in the United States. Note how you have taken the precaution against dividing by 0 in the event sales are down to 0.

```
@concat ( "Balance for other ",
          @image(RecordsToSkip),
          " customers")
```

This is a string containing “Balance for other 123 customers”: `RecordsToSkip` has the value `123`. Here, one of the parameters to a predefined function (`@concat`) is itself a predefined function (`@image`).

Preference specification

At the top of a report, you can specify preference settings, which apply to the whole report.

Preferences, syntax

```
<preferences part> ::=
   <Empty> | "PREFERENCES" { <Preference specification> }
<Preference specification> ::=
   { "REPORTLANGUAGE DANISH" | "VARDECL MANDATORY" }
```

Preferences, semantics

Preferences are specified at the start of the report. See the subsection “19: Report structure” on page 56 for more information.

You can specify two kinds of variable settings: Language, and whether variables are to be type declared in the beginning of the report. For more information, please see the subsections 19.3, “Language” and 12.2, “Variables, semantics”.

Cursor declarations

A cursor is a pointer to a specific relation. A relation is a table of data in the database. Maconomy is designed from relations. The relations are described in the document “Database Description”.

Cursors, syntax

```
<Cursor declaration> ::=
  CURSOR <CursorName> IS
  <Query expression> [<Order-by-part>]
<CursorName> ::=    <Identifier>
<Query expression>  ::=    <Query specification>
<Query specification>      ::=
  SELECT <Selection> <Table expression>
<Selection>   ::=
  ALL | * | <SelectionsList>
<SelectionsList>    ::=
  <selection> {, <selection> }
<selection>   ::=    <FieldName>
<FieldName>   ::=    <Identifier>
<Table expression>  ::=
<From clause>
  [<Where clause>]
  [<Group-by-clause>]
  [<Having clause>]
<From clause>::=    FROM <RelationName>
<RelationName>      ::=    <Identifier>
<Where clause>      ::=    WHERE <SearchCondition>
<Group-by-clause>   ::=
  GROUP BY <FieldNameList>
<FieldNameList>     ::=
  <FieldName> { , <FieldName> }
<Having clause>     ::=
  HAVING <SearchCondition>
<Order-by-clause>   ::=
  ORDER BY <OrderList>
<OrderList>   ::=
  <Order> { , <Order> }
<Order>       ::=
<FieldName> [ASC | DESC]
<SearchCondition>   ::=
  <SQL Logical term> |
  <SearchCondition> OR <SQL Logical term>
<SQL Logical term>  ::=
  <SQL Logical factor> |
  <SQL Logical term> AND <SQL Logical factor>
<SQL Logical factor>       ::=
  [NOT] <SQL Logical primer>
<SQL Logical primer>       ::=
  <SQL Predicate> | ( <SearchCondition> )
```

```
<SQL Predicate>     ::=
  <SQL Comparison>
<SQL Comparison>    ::=
  <SQL Scaler>
  [ <RelationOperator> <SQL Scaler>]
<SQL Scaler>  ::=
  <SQL Term>    |
  <SQL Scaler> <Additive Operator> <SQL Term>
<SQL Term>    ::=
  <SQL Factor> |
  <SQL Term>
  <Multiplication Operator> <SQL Factor>
<SQL Factor>  ::=
  <Addition Operator> <SQL Primer>
<SQL Primer>  ::=
  <SQL Atom>  |
  <SQL FunctionReference>  |
  ( <SQL Scaler expression> )     |
  <CursorName> . <FieldName>
<SQL Atom>    ::=
  <Variable>     |
  <Attribute>    |
  <Literal>
<SQL FunctionReference>::=
  COUNT ( * )      AS <Identifier>| MIN
  ( <FieldName> ) AS <Identifier>| MAX
  ( <FieldName> ) AS <Identifier>| SUM
  ( <FieldName> ) AS <Identifier>| AVG
  ( <FieldName> ) AS <Identifier>
<SQL AccessControlSpecification> ::=
  DIRECTACCESSCONTROL |
  INDIRECTACCESSCONTROL ( <FieldName> , <RelationName> )
```

`ALL` is synonymous with the symbol * in field selections.

Cursors, semantics

The search direction (`ASC` or `DESC`) is ignored. All sorting is `ASC` (abbreviation of ascending), i.e. smaller field values go before larger field values. `HAVING` and `GROUP BY` clauses are ignored.

The cursor name (after `CURSOR`) must be unique for the cursor. Most importantly, a cursor cannot have the same name as a relation.

All cursors are ‘read only’, which means that you cannot update or add records in the database.

The relation name in the `FROM` part must name a relation in the database.

A selection is a list of fields in the relation given in the `FROM` part. Individual field names can only occur once in this list. If you specify `ALL` (or `*`), you select all the fields in the relation. In this case, the cursor's fields are given the same names as the relation's fields.

SQL search expressions look like any other expressions. However, there are restrictions: It is not permitted to state references to predefined functions. The semantics of SQL search expressions are the same as for ordinary logical expressions.

The `WHERE` clause limits the number of records traversed in a relation. The report only retrieves the records which meet the conditions set in the `WHERE` clause. If you do not specify a `WHERE` clause, you retrieve all the records in the relation, i.e. the expression is true.

There are certain limitations to which cursor fields you can refer to in the WHERE and

HAVING clauses. These are described in the chapter on scopes and visibility.

All the fields in the GROUP BY part must be included in the selection. The fields can only appear once in the GROUP BY part.

The ORDER BY part specifies the sorting order of the returned records. If you do not give an ORDER BY part, the sorting order is arbitrary, and can vary from report to report. All the fields in the ORDER BY must exist as fields in the relation stated in the FROM part, but they do not need to appear in the selection. Write the most important sorting field at the top of the list.

It is possible to specify function references in your selections. SQL defines the following functions: SUM, COUNT, AVG, MIN, and MAX. The identifier after AS is the name the function value is read from. These identifiers are called the function aliases.

Every one of these identifiers can only appear once.

COUNT (*) counts the number of records. SUM (field name) calculates the sum of the values for every field name. AVG (field name) calculates the average of the field values in a given field. MIN (field name) calculates the smallest value and MAX (field name) calculates the largest value. The report only takes into consideration the records specified in the WHERE part.

A cursor with a selection containing function references always returns just one record – the result of the calculation. This returned record does not, in other words, exist in the database, but is the result of the calculation of the selected records in the database. The calculated values are identified by the names given after AS. The types of these "fields" (which are not relation fields) depends on their function and on the types of the argument fields. COUNT returns an integer value, AVG returns a real number value. SUM, MIN, and MAX return a value of the same type as the argument.

For Maconomy systems on which the add-on "Extend Access Control" has been installed, extended access control can be activated in reports by specifying the constructs DIRECTACCESSCONTROL and INDIRECTACCESSCONTROL, which are inserted in the WHERE part of a cursor declaration. DIRECTACCESSCONTROL means that only entries in the relation in question to which the current user has access are returned, whereas INDIRECTACCESSCONTROL (<FieldName> , <RelationName>) means that only entries in the relation in question are returned if the field <FieldName> refers to entries in the relation <RelationName> to which the current user has access.

Examples

```
Cursor CustomerCursor is
   select all from Customer
```

Retrieves the records in the Customer relation, in unspecified order.

```
Cursor CustomerCursor is
   select all from Customer
   order by CustomerNumber
```

Retrieves the records in the Customer relation, sorted in ascending order by customer number.

```
cursor CountCursor is
   select count(*) as theCount,
     sum(DebitBalanceBase) as theSum
   from Customer
   where
     (CustomerNumber >= FromCustomerVar and
     CustomerNumber <= ToCustomerVar)
```

Returns just one record stating how many records meet the condition that customer number is within the range `FromCustomerVar` to `ToCustomerVar`. It also returns the sum of the `DebitBalanceBase` field for the same records. The number of records is retrieved with `CountCursor.theCount`, and the sum with `CountCursor.theSum`.

```
Cursor JobCursor is
   select all from JobHeader where DirectAccessControl
```

Returns the entries in the `JobHeader` relation to which the current user has access.

```
Cursor RequisitionCursor is
   select all from InternalRequisitionHeader
   where

     IndirectAccessControl ( JobNumber, JobHeader )
```

Returns the entries in the `InternalRequisitionHeader` relation where the field `JobNumber` refers to an entry in the `JobHeader` relation to which the current user has access.

Target group specification

The target group (or target specification) is used to describe the Selection Criteria parameter dialog, which you see when the report is generated. The parameter dialog is used to specify which data to include in the report.

The target group dialog is specified by using an (optional) target specification.

Target group, syntax

```
<Target Specification> ::=
   TARGET [ ( <StringLiteral> ) ]
   <TargetGroupLine> { ; <TargetGroupLine> }
<TargetGroupLine> ::=
   <PromptString> [ <Attributes> ]
   { <TargetGroupElement> }
<TargetGroupElement> ::=
   { <VariableReference> | <String> }
<Variablereference>::=
   <VarRef> [ <Attributes> ]
[ ( <Descriptor> [ , <Descriptor> ] ) ]
<Descriptor>         ::=    MANDATORY | <Literal>
<String>             ::=    <Stringliteral> [ <Attributes> ]
<PromptString>       ::=    <StringLiteral>
<VarRef>             ::=    <Identifier>
```

Note that the target group lines are separated by, not terminated by, a semicolon. Every line must start with a string literal, serving as a prompt string. Every string literal and variable reference can be assigned an attribute list, which describes what the string or input looks like. Variables can furthermore be assigned one or two descriptors in parenthesis. A descriptor is either the word `MANDATORY` or a literal.

Target group, semantics

The string literal specified between the parentheses after `TARGET` is the title of the parameter dialog. If nothing is specified, the window title is “Enter Values”.

If no width attribute (`:WIDTH=...`) is specified on any of the prompt strings (i.e. the strings at the start of the target group lines), the lengths of all the prompt strings are given the length of the longest prompt

string, thus aligning the fields following the prompt strings. You can also use width attributes to determine the width of these fields, which is where the user enters data specifications.

The target group dialog has data fields wherever the target group lines contain variables. If the variable is logical, the program creates a check box. If the variable is an enumerated type, the selection is done in a popup field.

You can specify one or two descriptors after a variable in a target specification line. A descriptor can be either the word `MANDATORY` or a literal. `MANDATORY` means that you cannot close the target group dialog before entering valid data. The literal is a standard value, which is already inserted when the dialog opens. You cannot give two standard values or two `MANDATORY` specifications on the same variable.

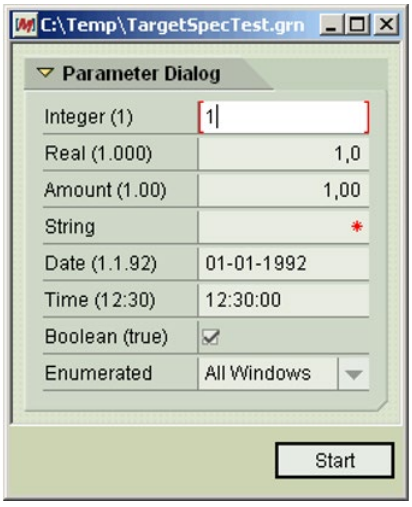
You are recommended to keep the appearance of the dialogs consistent with the design of the Maconomy dialogs.

When the user clicks “OK” in the parameter dialog, the variables are given the values specified by the user.

Example

```
TARGET ("Parameter Dialog")
   "Integer (1)":100        iVar (1) ;
   "Real (1.000)":100       rVar (1.000) ;
   "Amount (1.00)":100      aVar (1.00) ;
   "String":100             sVar (MANDATORY);
   "Date (1.1.92)":100      dVar ("1.1.92") ;
   "Time (12:30)":100       tVar ("12:30") ;
   "Boolean (true)":100     bVar (true) ;
   "Enumerated":100         eVar -- No default allowed
```

Yields the window below, which also demonstrates the different types in RGL:



Levels

Levels express the structure of the data included in the report. They are the basic building blocks with which you design reports.

Levels, syntax

```
<LevelSpecification> ::=
   LEVEL <IntegerLiteral>
   [ IS <Identifier> [ GROUP BY <FieldList> ]]
   [ <Header Spec> ]
   [ <Lines Spec> ]
   [ <Trailer Spec> ]
<Header Spec> ::=
   HEADER
   { <Statement> }
<Lines Spec> ::=
   LINES
   { <Statement> }
<Trailer Spec> ::=
   TRAILER
   { <Statement> }
<FieldList> ::=
   <Identifier> { , <Identifier> }
```

Levels consist of the word `LEVEL`, an integer determining the level's place in the hierarchy, an optional cursor attachment and optional `HEADER`, `LINES`, and `TRAILER` specifications. If the level has a cursor, you can specify a `GROUP BY` part, which consists of a list of fields, separated by commas.

The `HEADER`, `LINES`, and `TRAILER` parts consist of an optional statement list.

Levels, semantics

The level numbers reflect the hierarchy. The first level always has number 1. The next level has number 2 etc. Every level with level number N can contain a number of nested levels, which all have level number N + 1.

Example

```
(A)    level 1 is CursorA
(B)       level 2 is CursorB
(C)       level 2 is CursorC
(D)         level 3 is CursorD
```

Level 1 has two nested levels -- level 2 (B) and level 2 (C). The second level, 2 (C), also has a nested level, level 3 (D). The structure is emphasized with indentation.

Here it is appropriate to introduce the concept of level path. A level path is a list of levels from level 1 (called the root) to the given level. In the above example, level B has the path (A, B) and level D has the path (A, C, D).

In the following, it is assumed that all levels have a cursor. When the report is run, the records are retrieved like this: When the report retrieves one record from a given level, it retrieves all the records in the nested levels.

In the example above, the records are fetched in the following order:

```
A1
  B1
  C1
```

```
    D1
    D2
    :
    Dn
  C2
    D1
    :
    Dn
  C3
  :
  Cn
A2
  B1
  C1
:
An
```

First, the report retrieves the first record in `CursorA`. It then gets the first record in `CursorB` and `CursorC`. It then gets all the records under `CursorD`. It then gets the second record in `CursorC`, after which it examines all the records under `CursorD`. This continues until there are no more records in `CursorC`. Then it gets the next record in `CursorA`, and the whole procedure starts again, continuing until there are no more records in `CursorA`.

HEADER, TRAILER, and LINES

Statements are executed in the `HEADER` part, before any records are fetched. Statements are executed in the `LINES` part once for every record in the cursor. If the level does not have a cursor, the statements processed just once in the `LINES` part.

The cursor after `IS` must be declared in a cursor declaration.

GROUP BY

The identifiers in the `GROUP BY` part must be fields in the corresponding cursor's selection. If a given level in the path has a group by part, the next level in the path must have either

- no `GROUP BY` part, or
- a `GROUP BY` part formed by adding one or more fields behind the fields in the level's `GROUP BY` part. All the levels in such a sequence (from the first `GROUP BY` part to the first level without a `GROUP BY` part) must have the same cursor.

The cursor in the levels, which have a GROUP BY part, must have an `ORDER BY` part which includes the longest group by list.

Examples

```
Cursor1 is
   select all from Relation
   order by Field1, Field2
```

```
Level 1 is Cursor1
   Level 2 is Cursor2 group by Field1
      Level 3 is Cursor2 group by Field1, Field2
         Level 4 is Cursor2
```

This is a legal structure. The group by list at level 3 is produced by adding fields (`Field2`) to the previous level's `GROUP BY` list. Furthermore, `Cursor1` has an `ORDER BY` part, which includes Field1 and Field2.

```
Level 1 is OrderCursor
   Level 2 is OrderCursor Group by Field1, Field2
      Level 3 is OrderCursor group by Field1
```

This is not legal, since the `GROUP BY` list at level 3 is not an addition of fields from the `GROUP BY` list from the surrounding level (2).

```
Level 1 is OrderCursor group by Field1
   Level 2 is OtherCursor
      Level 3 is OrderCursor group by Field1, Field2
         Level 4 is OrderCursor
```

This is not legal, since level 2 with the `OtherCursor` cursor comes between the `GROUP BY` levels with the `OrderCursor` cursor.

```
Level 1 is OrderCursor group by Field1
   Level 2 is OrderCursor group by Field1, Field2
```

This is not legal, since there is not a level 3 with an `OrderCursor` without a `GROUP BY` part.

The `GROUP BY` part causes the data to be grouped in such a way that a level's `LINES` part is processed once for every different combination of the group by fields in the cursor.

Example

You wish to print all the orders in the system, and all the items in individual orders. The `OrderLines` relation has the information. You write two levels: Level 1 is grouped according to order number. All the different order numbers are processed in the `LINES` part of Level 1, taking one record for every different order number. All orders with the same order number are processed in the `LINES` part of level 2.

```
cursor OrderLines is
   select all from OrderLines
   order by OrderNumber
level 1 is OrderLines group by OrderNumber
header
   -- Print suitable heading
   "OrdersList":400
      :justification=center:Size=15
lines
   "Order number " OrderLines.orderNumber
trailer
```

```
   -- Mark that print is finished
   "End of OrdersList":400
      :justification=center:Size=15
level 2 is OrderLines
header
   "":60 -- Indentation
   "ItemNumber":60 " " "ItemText":200
lines
   "":60 -- Indentation
   OrderLines.ItemNumber:60 " "
   OrderLines.ExternalItemText:200
```

Orders List

Order number 200001

Item number	Item text
1240	Sofa, 3-seater, leather
1234	Sofa, 2-seater, leather
1272	Dining chair, oak
1262	Dining table, oak
1260	Dining table, beech

Order number 200002

Item number	Item text
1281	Bed, white
1280	Bed, black
1301	Bedside table, white
1300	Bedside table, black
1292	Mattress, water
1291	Mattress, plain
1290	Mattress, deluxe

Order number 200003

Item number	Item text
1251	Sofa table, palisander
1241	Sofa, 3-seater, canvas
1235	Sofa, 2-seater, canvas

Order number 200004

Item number	Item text
1252	Sofa table, oak
1242	Sofa, 3-seater, cotton
1236	Sofa, 2-seater, cotton

Order number 200005

Item number	Item text
1290	Mattress, deluxe
1302	Bedside table, natural
1282	Bed, natural
1252	Coffee table, oak
1236	Sofa, 2-seater, cotton

End of Orders List

Report structure

This subsection describes the basic structure of the report definition.

Report structure, syntax

```
<report>     ::=
   <preferences part>
   <variable declaration part>
   <target specification>
   <cursor declaration part>
   <level specification part>
```

A report consists of an optional target group specification, an optional cursor declaration list, and an optional list of levels.

Report structure, semantics

The first thing that happens when you run the report is that the target group window opens. When you leave this window (by clicking OK), the variables (if there are any) in the target group are assigned their values.

The levels are then processed as described in the subsection 4, “Levels”.

Language

Previously, a Danish and an English version of the Report Designer existed. This is no longer the case. This means that you can use either Danish or English language predefined variables (see the subsection 12.3, “Predefined variables”) and that names of the days in the week and month names by default will be in English.

To use Danish names for weekdays and months, insert a preference `REPORTLANGUAGE DANISH` in the preferences part of the report, i.e.

```
PREFERENCES ::=     REPORTLANGUAGE DANISH
```

Please note that:

- In practice, this only affects reports using weekday names or month names in Danish.
- Weekday names and month names in other languages than English or Danish are easily implemented using arrays of strings.

Visibility and scope

The structure of a report definition implies that it is not always permitted to refer to the name of a field or variable. The visibility rules, i.e. the rules which govern where field references and variable references are allowed, are described in this section.

The general rule is that a name must be unique. You cannot therefore have a variable with the same name as other variables, cursors, fields, relations or enumeration literals.

However, by using an escape character (“\”) you can refer to fields where the name is the same as a reserved name in RGL (the Report Generator Language), e.g. the field “`Header`” in the relation `FinanceReportLine`. Note that `name` and `\name` are equivalent spellings of the same item, e.g. a variable.

Example

```
HEADER
LINES
   IF \Header = TRUE THEN
   ...
```

Variables

Variables are visible throughout the report. If a variable exists in a target specification, it is assigned its values when the report is generated. If it appears elsewhere, it represents a value. A variable represents one and only one type of value. This value is always defined; if the user has not assigned it, it takes on a standard (default) value, determined by its type.

Cursor fields

In most cases, you can specify a cursor field by its full name, for instance `MyCursor.FieldName`, or by an abbreviated name, which does not include the name of the cursor. This abbreviated name should be used when there is no risk of ambiguity.

In cursor declarations

A cursor field can occur in a cursor declaration, if the cursor defining the field has been declared before the cursor declaration in which it is used. If you only supply the field name (without the prefixed cursor name), the report uses the nearest cursor with a selection which incorporates the field. If there is no such cursor, the report designer issues an error message. The field can be referred to as `MyCursor.FieldName`, if the `MyCursor` selection includes the field called `FieldName`. In this way, the field is clearly identified.

In levels

A cursor attached to a level is called the level's primary cursor. You can refer to primary cursor fields and to fields from all nested levels in the `LINES` part of a given level. In the `HEADER` and `TRAILER` parts, you can refer to cursors from all the levels in where the level is nested, but not to fields in the primary cursor. It is in the `LINES` part that the primary cursor records are processed. It is therefore meaningless to refer to this cursor in the `HEADER` and `TRAILER` parts; the cursor is not defined before the first record has been retrieved (the `HEADER` part) or before the last record has been retrieved (the `TRAILER` part).

If the cursor name is not given with the field name, the program uses the nearest cursor, whose selection includes this field.

Example

```
cursor PurchaseHeader is
   select all from PurchaseHeader
cursor PurchaseLines is
   select all from PurchaseLines where
   PurchaseOrderNumber =
      PurchaseHeader.PurchaseOrderNumber
   order by LineNumber
-- Levels
level 1 is PurchaseHeader
lines
   -- here we use PurchaseHeader.PurchaseOrderNumber
   PurchaseOrderNumber Name1: 80mm Name2: 80mm
level 2 is PurchaseLines
lines
   " " :30mm -- indent
   ItemText          :30mm -- PurchaseLines.ItemNumber
   ItemText1         :80mm -- PurchaseLines.ItemText1
   QuantityOrdered   :30mm
```

The report has two levels. Level 1 prints the purchase order and level 2 prints the order lines. In the lines part of level 2 you can refer to the primary cursor in level 1, just as you can refer to level 2's own cursor.

Note also that the `PurchaseLines` cursor's `WHERE` part has references to `PurchaseOrderNumber` fields both in the `PurchaseHeaders` cursor and the `PurchaseLines` cursor.

You cannot refer to a field from a cursor, which is not assigned a surrounding field - either in levels or in cursor declarations.

Example

```
cursor A is
   select a1, a2 from Relation1
cursor B is
   select b1, b2 from Relation2
   where b1 = a1
level 1 is B - ILLEGAL
```

You cannot refer to `cursor B` in level 1, because of the assumption that, in `B's WHERE` part, a1's value belongs to a cursor (namely `A`), which does not have a value in level 1.

Note that the cursor declaration is itself legal; it is the way it is used which gives an error in the cursor declaration.

The records belonging to a given cursor are retrieved when the matching `LINES` part is processed. If the cursor's `WHERE` part has variables, it is the values of the variables just prior to the lines part, which determine which records will be retrieved.

Predefined functions

Expressions can contain calls to predefined functions. The predefined functions are described in this subsection.

Predefined functions fall within the following groups: Mathematical functions, date functions, time functions, string functions, and miscellaneous functions.

Mathematical functions

Arguments for mathematical functions must be numerical.

Function	Description

`Abs (argument:integer) : integer`
`Abs (argument:amount) : amount`
`Abs (argument:real) : real`
	Returns the absolute value of the argument, i.e. the resulting value is always positive or 0. The result is the same type as the argument.

`Exp (argument:integer) : real`
`Exp (argument:amount) : real`
`Exp (argument:real) : real`
	Returns the natural anti-logarithm of the argument. `ab` can be written as
	`@EXP (b*@LN (a)).`
	The result is real, irrespective of the type of argument.

Function	Description
`Ln (argument:integerExpr) : real Ln (argument:amount) : real Ln (argument:real) : real`	Returns the natural logarithm of the value of the argument, which must be greater than 0.

Date functions

Date functions are used to calculate dates.

Function	Description
`DayOf (the date:date) : integer`	Retrieves the day's number in the month, using the date defined by the argument. Thus, `@DayOf("22.12.61") = 22.`
`MonthOf (the date:date) : integer`	Retrieves the month using the date defined by the argument. Thus, `@MonthOf ("22.12.61") = 12.`
`YearOf (the date:date) : integer`	Retrieves the year from the date defined by the argument. Thus, `@YearOf ("22.12.61") = 1961.`
`Date (day : integer, month : integer, year : integer) : date`	Constructs the date from the three arguments. Thus, `@Date(22,12,1961) = "22.12.61".`
`AddDays (the date : date, days : integer) : date`	Calculates the date by adding the number of days in argument 1 to the date in argument 2.
`AddWeeks (the date : date, weeks : integer) : date`	Calculates the date by adding the number of weeks in the second argument to the date in the first argument.
`AddYears (the date : date, year : integer) : date`	Calculates the date by adding number of years in the second argument to the date in the first argument.
`WeekDay (the date : date) : integer`	Returns the weekday as an integer between 1 (Sunday) and 7 (Saturday).
`Week (the date : date) : integer`	Returns the week number as an integer between 1 and 53.
`MonthName (the month : integer) : string`	Returns a string containing the month's name. E.g. January, when the value of the argument is one. If the month's number is not between 1 and 12, the program returns the string "`*****`".
`DayName (the date : integer)`	Returns a string containing the day's name. E.g. Sunday, when the value of the argument is one. If the

Function	Description
`: string`	days' number is not between 1 and 7, the program returns the string "*****".
`NoOfDays(StartDate : date EndDate : date) : integer`	Calculates how many days separate the two dates.
`NoOfWeeks (StartDate : date EndDate : date) : integer`	Calculates how many (whole) weeks separate the two dates.
`NoOfYears (StartDate : date EndDate : date) : integer`	Calculates how many (whole) years separate the two dates.

Examples

Calculate (in the variable `NoDaysLeft`) how many days are left of this quarter year.

```
Quarter2 := @date (1, 4,   @yearof(ThisDay))
Quarter3 := @date (1, 7,   @yearof(ThisDay))
Quarter4 := @date (1, 10, @yearof(ThisDay))
Quarter5 := @date (1, 1,   @yearof(ThisDay)+1)
-- find out which quarter starts after
-- today's date.
NextQuarter :=
   @if (ThisDay □ Quarter4,
      Quarter5 ,
         @if (ThisDay □ Quarter3,
            Quarter4,
               @if (ThisDay □ Quarter2,
                  Quarter3,
                     Quarter2)))
   No.DaysLeft :=
   @NoOfDays (NextQuarter, ThisDay)
```

Calculate (in the variable `LastDay`) the last day in this month.

```
ThisYear := @YearOf()
ThisMonthStart:= @date(1, @monthOf(ThisDay), ThisYear)
   -- If we are in December, then next month starts
   -- next year.
NextMonthStart :=
   @if (@monthOf(ThisDay) = 12,
      @date(1, 1, @yearof(ThisDay)+1),
      @date(1, @monthOf(ThisDay) + 1, ThisYear)
   LastDay := @AddDays (NextMonth'sStart, -1)
```

Time functions

Time functions are used to calculate at the time of day within 24 hours.

Function	Description
`Time (hours : integer, minutes : integer, seconds : integer) : time`	Constructs the time from the value of the three arguments. Thus, `@Time(14, 43, 11) = "14:43:11"`
`AddSeconds( TheTime : time seconds : integer) : time`	Calculates the time by adding the number of seconds in the second argument to the time of the first argument.
`AddMinutes( TheTime : time minutes : integer) : time`	Calculates the time by adding the number of minutes in the second argument to the time of the first argument.
`AddHours( TheTime : time hours : integer) : time`	Calculates the time by adding the number of hours in the second argument to the time of the first argument.
`SecondOf( TheTime : time) : integer`	Returns the number seconds in the time specification. Thus, `@SecondOf ("14:43:11") = 11.`
`MinuteOf( TheTime : time) : integer`	Returns the number minutes in the time specification. Thus, `@MinuteOf("14:43:11") = 43.`
`HourOf( TheTime : time) : integer`	Returns the number hours in the time specification. Thus, `@HourOf("14:43:11") = 14.`
`NoOfSeconds( End : time, Start : time) : integer`	Calculates the number of seconds between the two times.
`NoOfMinutes( End : time, Start : time) : integer`	Calculates the number of minutes between the two times.
`NoOfHours( End : time, Start : time) : integer`	Calculates the number of hours between the two times.

String functions

String functions process string arguments.

Function	Description
`Image ( Arg : ? ) : STRING`	Returns the argument as a string. The argument can be of any type.
`Concat ( Arg1 : ?, ... ArgN : ? ) : STRING`	The result is calculated by placing together the string representations of the arguments. There can be as many as 30 arguments, separated by commas. The arguments can be of any type.

Function	Description
SubStr (Str : STRING, From : INTEGER, Number : INTEGER) : STRING	The result is the string, which is produced by taking Number characters out of the string Str, from and to the position from. Thus, SubStr ("123456", 2, 3) = "234". If From or Number are 0 or negative, the empty string is returned.
StringToInteger (Str : STRING) : INTEGER	Returns the integer type value of Str. The value of Str must follow the syntax of an INTEGER literal.
StringToAmount (Str : STRING) : AMOUNT	Returns the amount type value of Str. The value of Str must follow the syntax of an AMOUNT literal.
StringToReal (Str : STRING) : REAL	Returns the real type value of Str. The value of Str must follow the syntax of an REAL literal.
StringLength (Str : STRING) : INTEGER	Returns the length, i.e. the number of characters, of Str.
StringPos (Str1 : STRING, Str2 : STRING, CaseSensitive : BOOLEAN) : INTEGER	Returns the position, i.e. the character number, of the first occurrence of Str1 within Str2. The CaseSensitive parameter controls whether the matching is case sensitive. If Str1 does not occur within Str2, the result is 0.
StringDelete (Str : STRING, Position : INTEGER, Number : INTEGER) : STRING	Returns the STRING value obtained by deleting Number characters, beginning at position Position, from Str. If the deletion specified exceeds the end of the string, the result is Str truncated from Position.
UpperCase (Str : STRING) : STRING	Returns the upper case image of Str.

Examples

```
S := @concat ("There are " ,
   @Image(NumberOfEntries),
   " customer entries")
```

If NumberOfEntries has the value 124, the string variable S will contain the string:

```
  "There are 124 customer entries"
```

The following expressions yield the following results:

```
@StringToInteger ( "234" ) 234
@StringToInteger ( "-345" )       -345
@StringToAmount ( "12.45" )       12.45
@StringToAmount ( "33.126,75" )   33126.75
@StringToReal ( "627.5" )  627.5
@StringLength ( "abcde" )  5
@StringPos ( "DE", "abcdefDEF", FALSE ) 4
@StringPos ( "DE", "abcdefDEF", TRUE )  7
@StringPos ( "BC", "abcdefDEF", TRUE )  0
@StringDelete ( "abcdef", 3, 2 )  "abef"
@UpperCase ( "aBcDeF" )    "ABCDEF"
```

Miscellaneous functions

The following describes the functions, which cannot naturally be grouped under the other functions.

Function	Description
`If ( condition : logic, IfValue : ?, ElseValue : ?) : ?`	If the condition is true, the value of the second argument is returned. Otherwise, the value of the third argument is returned. The second and third arguments must be of the same arbitrary type. The result of the function is also this type.
`Minimum ( Arg1 : ?, ..., ArgN : ?) : ?`	Returns the smallest value of the arguments. You can have up to 30 arguments, separated by commas. The arguments must be of the same arbitrary type. The result of the function is also this type.
`Maximum ( Arg1 : ?, ..., ArgN : ?) : ?`	Returns the largest value of the arguments. You can have up to 30 arguments, separated by commas. The arguments must be of the same arbitrary type. The result of the function is also this type.

Example

```
@if (v1 = 0.00, 0.00, v2 / v1)
```

If v1's value is 0, 0 is returned. Otherwise, v2 / v1 is returned. This means you avoid the undefined value, which is the result of dividing by 0.

Error Messages

As you develop your reports, you will sometimes find that the report designer finds an error in the report definition. It then writes a file with an error message. This subsection explains all the possible error messages.

Often the message will contain a reference to a line number in the report text. The text within the < > contains information from the report definition.

Message	Description
String exceeds maximum allowed length.	You have written a string longer than the maximum permitted 255 characters. This is often due to the absence of a string terminator (").
Incompatible types for operation. Illegal type assignment to variable <name>.	Expressions have been used in a context in which the types are not compatible. For instance, you cannot assign a string value to an integer variable. Neither can you use the `AND` and `OR` operators on numeric expressions.
Unknown type <name>.	A type is expected. Shown in connection with attributes. The error can e.g. be caused by a misspelling of an enumerated type.
Field <cursorname>.<fieldname> not found.	The specified field name does not occur in the selection for the cursor specified in `<cursorname>`. Check the list of relations and their fields (the Database Description document).
Cursor <name> not found.	Displayed if the name of `IS` at a level is not a cursor.
Attribute VAL only applies to enumeration types Attribute POS only applies to enumeration types	The `VAL` and `POS` attributes are only permitted for enumeration types.
First level not 1 Level < 1 Level > previous + 1	A level number, which is wrong in this context, was specified. Level numbers may only be one greater than the previous level number. First level number must be 1.
Level specification: level no. not an integer.	Level numbers must be integers.
ORDER BY field list not part of selection.	In the `ORDER BY` part in a cursor declaration some fields are specified which do not belong to the cursor selection.
Level GROUP BY part not included in ORDER BY part of cursor <cursorname>.	Shown when translating the `GROUP BY` parts of levels. All fields specified here must also occur in the `ORDER BY` part of the primary cursor. Furthermore, the fields in the `ORDER BY` part of the primary cursor must be in the sequence of the fields in (the longest) `GROUP BY` part.
Function call: Function <name> not found.	`@ <name>` was specified, but `<name>` is not a predefined function. See the list of functions in the subsection 21, “Predefined functions”.

Message	Description
Attribute "<name>" not found.	The specified attribute is not defined.
Function call: Wrong no. of arguments.	The number of parameters in the function call does not correspond to the definition of the function. See the list of functions in the subsection 21, “Predefined functions”.
Attribute call: Wrong no. of arguments.	The number of parameters in attribute call does not correspond to the definition.
Relation <name> not found.	A name was specified after `FROM`, which is not the name of a database relation. Check the list of relations and their fields (the Database Description document).
Descending ordering not supported.	All ordering specifications must have the same sort order, which is ascending order (`ASC`).
ORDER BY field list not unique.	A field may only occur once in the `ORDER BY` part of a cursor.
Selections must be either all simple or all functions.	It is not permitted to mix simple selections, i.e. selections which only consist of field names, `ALL` or *, with SQL functions (`SUM`, `COUNT`, `MIN`, `MAX`, or `AVG`).
GROUP BY field list not unique.	The field of the specified name occurs more than once in the `GROUP BY` part of a cursor declaration.
Warning: HAVING clauses are not supported. Warning: GROUP BY clauses in cursor declarations are not supported.	`HAVING` and `GROUP BY` for cursors are ignored and should be removed.
SUM may only be applied to INTEGER, AMOUNT or REAL fields AVG may only be applied to INTEGER, AMOUNT or REAL fields	The calculation of sums and averages is only permitted for arithmetic types.
Cursor <name> already declared. Variable <name> already declared	The name has already been used. Use another name for this cursor or variable.
Variable <name> declared twice.	Variables can only be assigned a value once in a target group specification. Introduce a new variable.

Message	Description
Variable <name> not declared	The `PREFERENCES` part specifies that variables must be type declared (`VARDECL MANDATORY`). Declare the specified variable.
Default value specification <value> not allowed.	You cannot assign a default value for popup fields (i.e. enumeration values) in the target group dialog.
Two default value specifications for variable <name>.	Remove one default value specification.
Target dialog title not a string.	The title of a target group dialog must be entered as a string literal.
Target dialog prompt string not a string.	Each target group line must begin with a string literal, not a variable.
Target dialog element must be string or variable.	Each target group element must be a string literal or a variable.
Warning: Could not infer type for variable <name>, assuming STRING.	Not enough information is available to deduce the type of the variable. The variable typically appears only in a print statement and is not used in any other context. Consider declaring the variable.
Warning: Variable '<name>' is never initialized.	The variable is never assigned a value. This may of course be intentional, if the defined default value (e.g. 1 for integer variables or today's date for a date variable) makes sense in the calculations. However, the warning can also be due to an error such as a misspelled variable name.
"HEADER", "LINES", or "TRAILER" expected.	After `LEVEL...`, a `HEADER`, `LINES`, or `TRAILER` must be specified before any statements.
"HEADER" unexpected. "LINES" unexpected. "TRAILER" unexpected.	`HEADER`, `LINES`, and `TRAILER` parts are specified in the wrong order.
';' should not be used between variable declarations (remove it) ';' should not be used between cursor declarations (remove it)	In many programming languages, such as C and Pascal, the character '`;`' is used as the separator. In the Report Designer language (RGL), no separators are used.

Message	Description
';' should not be used between statements (remove it).	
'=' not allowed for assignment, use ':='.	In many programming languages, such as C, the character '=' is used for assigning values. In the Report Designer language, the symbol ':=' is used.
Parse error, expecting <symbol>, read <symbol>.	A construction was specified with the wrong syntax.
Array lower bound greater than upper bound	The lower bound for an array must be less than or equal to the upper bound.
Array index type must be integer	The index of an array must be of the type `INTEGER`.
Array variable <navn> not declared	Variables of the type `ARRAY` must be declared in the report's variable declaration part.
Incompatible array types for assignment	The declared array type does not correspond to the type of the expression assigned to the array.
Array-to-array assignment not supported	An array cannot be assigned the value of another array.
Unrecognized report language specification <navn>	The expression `REPORTLANGUAGE` takes one of three values: `Danish`, `English`, or `<empty>`. If no value is specified, `English` is assumed.
DirectAccessControl: Argument list not allowed	This expression takes no arguments.
IndirectAccessControl: wrong no of. arguments <n> - should be 2	This expression takes two arguments: `FieldName` and `RelationName`.

About Deltek

Better software means better projects. Deltek is the leading global provider of enterprise software and information solutions for project-based businesses. More than 23,000 organizations and millions of users in over 80 countries around the world rely on Deltek for superior levels of project intelligence, management and collaboration. Our industry-focused expertise powers project success by helping firms achieve performance that maximizes productivity and revenue. www.deltek.com