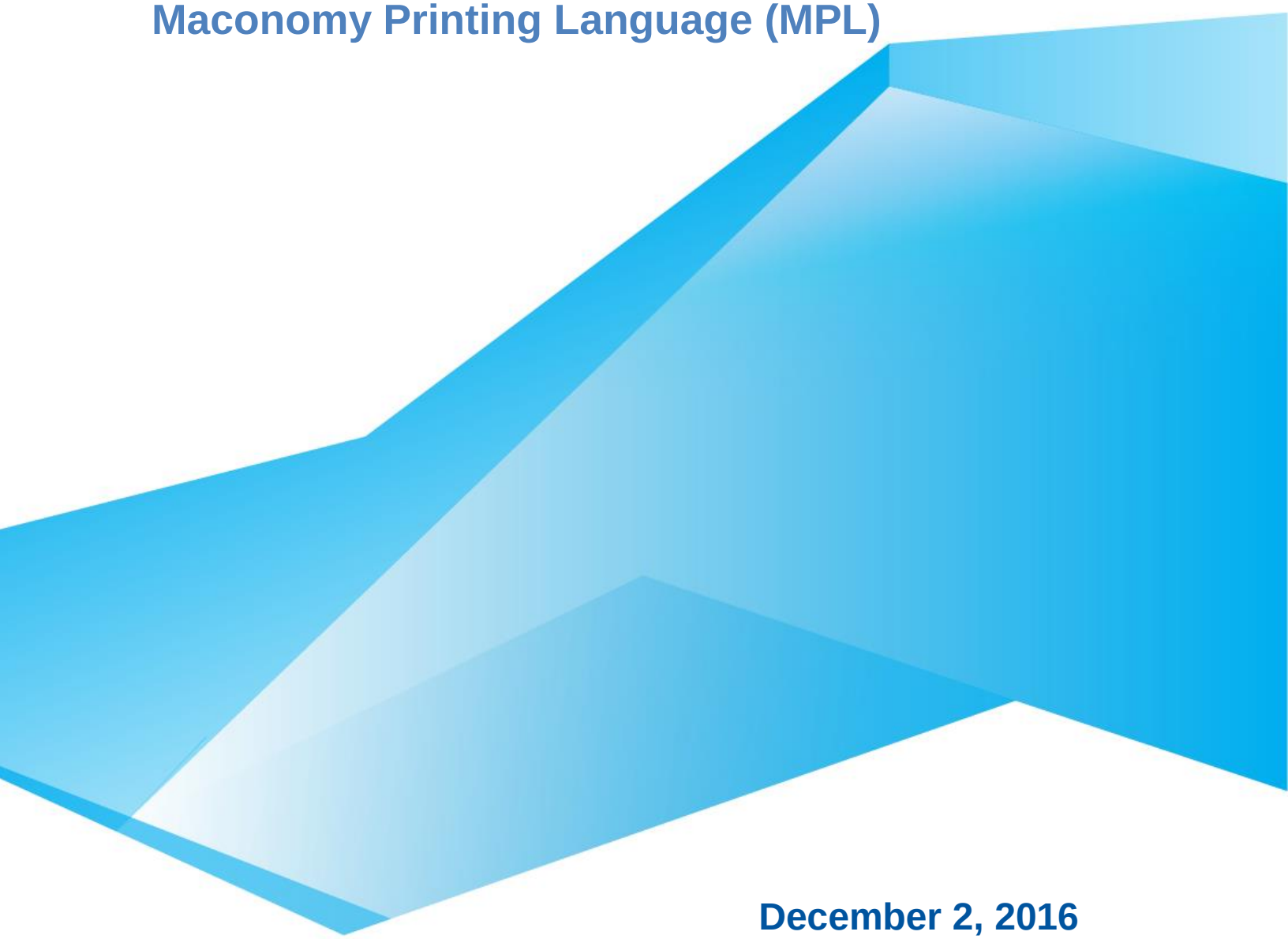
Deltek Know more. Do more.™

Deltek Maconomy 2.3 GA

Maconomy Printing Language (MPL)



December 2, 2016

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 edition published December 2016.

Contents

Introduction

This manual describes how printouts and Universe Reports are designed using the Maconomy Print Language (MPL). It covers all versions of MPL up to version 4.

MPL is a language used for defining both the contents (that it, data to be displayed) and the layout of printouts in Maconomy. It allows a developer to focus on specifying the logical structure of a printout, that is to say, how the elements on the printout should relate to each other, for example, that this column should stretch and those two elements should be aligned. This high-level description in MPL, called a layout, is then executed by the MPL engine, rendering an actual printout. The MPL engine makes sure that the physical layout of the rendered printout is best fitted to the content of the print.

This manual focuses on the language MPL. See the Set-Up section of the Maconomy reference manual for a description of the administration of MPL layouts after they have been created.

Prerequisites

This manual is both an introduction and a reference manual. For this reason, it sometimes seems very technical. The “Central Concepts” chapters give a good overview of the language. When reading the manual for the first time, it is recommended to browse through the technical sections quickly, as the most important concepts are later demonstrated through examples.

To be able to use this manual, you need basic knowledge of Maconomy. Furthermore, you need to be able to use an editor (such as TextPad, EditPad, or Notepad) and use the “Print Layout” windows in Maconomy as described in the Maconomy Reference Manual.

Version History

This section documents the history of changes to MPL. Because customers can use different versions of Maconomy, and hence, different MPL engine versions, this section gives a good overview of in which MPL version a particular feature has been introduced, as well as answer most backward-compatibility questions.

Changes in MPL Version 2

New Functionality

MPL 2 came out with the following new features:

- Print color (color attribute)
- Graphics/images in printouts (image tag)
- Move cursor to specific vertical position (goto tag)
- Style inheritance
- New attributes for blocks
- Extensions for Universe Reports

Changes in MPL Version 3

MPL version 3 is a major new version of MPL that breaks backward compatibility in some ways, but also provides a number of new features. The changes are described in details in “MPL Version 3,” while an overview is provided here.

MPL 2 and MPL 3 are currently both supported by the current Maconomy server, and it is up to the MPL layout writer to decide which MPL version to use. The Maconomy server, when compiling or executing an MPL layout, invokes either the old compiler and print engine (if the version tag states `<mpl 1>` or `<mpl 2>`), or the new compiler and print engine (if the version tag states `<mpl 3>`).

While MPL 3 is compatible with the great majority of existing MPL layouts, there are some things related to integration with existing client and server technologies to consider:

MPL 3 is only supported when using the Java client or the Portal. The MS Windows client does not support printing MPL 3 layouts (although it can be used to import them).

The MPL 3 print engine does not support adding RGL to the print through Active Scripting.

MPL 3 and MPL 2 cannot be mixed in situations where a number of layouts are used to generate one large print.

Also, you cannot mix MPL2 and MPL3 if there is a print layout selection rule that chooses between them. In practice that means that unless you always manually pick layouts when you print, either all layouts for a window must be MPL3, or none.

Deltek recommends having all print layouts for a dialog in a single format.

MPL 3 is intended to replace MPL 2 in the future, but until a new Maconomy server platform has been released, the two versions will coexist.

New Functionality

- These are the most import additions in MPL 3:
- Multiline text (`wrap` attribute on `<text>`, `<var>`, and `<data>` tags and the new `<concat>` tag)
- Ability to switch page orientation with `<newpage>`
- Conditionals can be negated and used with strings and database fields
- Supports PostScript (pfm, afm), OpenType (otf) and TrueType (ttf) fonts

Changed Functionality

The following list of changes in functionality is not a complete list, but it contains the most important ones:

- `<goto>` is no longer supported in headers or footers
- Conditionals no longer leave blank space when skipping content
- `<newpage>` in a row is no longer supported
- The scope of `<define>`, `<redefine>`, `<ruler>`, and `<subruler>` declarations and `<default>` has changed. They may now be specified anywhere in a parenthetical tag, not just at the beginning.

Changes in MPL Version 4 (as of TPU 16 SP0)

MPL 4 is a direct successor to and replacement of MPL 3 as of TPU 16 SP 0. Therefore, everything that applies to MPL 3 applies to MPL 4 as well, unless otherwise stated in this section.

This section provides an overview of the new features in MPL 4, as well as the changes that were introduced in this new version. For a detailed description of these matters, see “MPL Version 4.”

New Functionality

Version 4 allows for fetching custom data and performing custom calculations directly in an MPL layout. This new functionality is enabled through embedding in MPL 4 the *Expression Language* and *MQL*—two technologies that might already be familiar to a Maconomy consultant. The *Expression Language* is also used in other Maconomy layout languages like *MDML* and *MWSL*. *MQL*, on the other hand, is a statically typed database query language that is used for *Universe Reporting* and in *MScript* as well.

With these two new powerful tools at your disposal, you can now customize an MPL layout with any additional information that you might wish to include and that has not been included in the predefined print environment.

In particular, using the *MQL* support in MPL 4 you can now:

- Define reusable, parameterized database queries with the `<query>` tag. The queries are executed against the Maconomy universes, which feature database joins.
- Supply the query with the actual values of parameters it declares and instantiate it to a cursor by means of the `<cursor>` tag.
- Use the newly defined cursor as any other predefined cursor in a `<repeating>` tag.

Moreover, by using the Expression Language support in MPL 4, you can now:

- Perform arbitrarily complex calculations using expressions and standard functions.

- Bind the calculated values to mutable variables (**<var>**) and immutable constants (**<val>**).
- Assign new values to variables using the **<assign>** tag.
- Use expressions as conditions in the conditional tag as well as a path to the image in the **<image>** tag.

Changed Functionality

The following changes were introduced with respect to MPL 3:

- Field reference tag **<field>** has been desupported.
- Variable reference tag **<var>** has been desupported, and instead.
- The **<var>** tag means variable definition in MPL 4.
- When defining a tag, it is now disallowed to have white spaces in between the opening angle bracket "<" and the following tag name.
- Print structure check has been loosened up. For more information, see "Repeating Structure (MPL 4)."

Changes in MPL Version 4 (as of TPU 16 SP2)

The 2.1.1 release of Maconomy, which was delivered as TPU 16 SP2, came out with a couple of new features. Since this version of MPL is entirely backward-compatible with the previous version 4, the version number has not been changed.

New Functionality

As of this release, you can:

- Keep executing a block of MPL code while a certain condition is true by using the new `<while>` tag.
- Include static PDF documents in your MPL layouts by means of the new `<includepdf>` tag.
- Generate barcodes (14 different types) and QR codes using the new `<barcode>` tag.
- Manually control page numbering using the new `<nextpagenumber>` tag.
- Control whether headers and footers should be skipped when the enclosing `<while>`/`<repeating>` tag is empty (that is, no iteration took place) by using the new `skipHeaderFooterIfEmpty` attribute on the `<while>` and `<repeating>` tags.

Central Concepts

This section introduces the terminology and concepts necessary for understanding this manual and for using MPL. It starts off with an example of a simple layout and based on that, it explains the most fundamental concepts in MPL.

Example

MPL is a language for defining print layouts in Maconomy, usually for reporting purposes. When designing a print layout, you must address at least two kinds of concerns:

1. The *contents* of the print — Which data is this layout supposed to present?
2. The *layout* of the print (the presentation layer) — How to best present the data on this print so that it looks visually compelling?

This section examines a very simple layout that lists all of the employees in the Maconomy database, specifying their names, employee numbers, and the location of companies they work for. In other words, based on the data in the Maconomy system, we want to generate a simple table like the following.

NAME	EMPLOYEE N0	COMPANY BASED IN
Charlie Kaufman	01	New York
William Shakespeare	02	United Kingdom
Tigran Hamasyan	03	France
Roman Polanski	04	France
Pedro Almodóvar	05	Spain
Witold Gombrowicz	06	France
David Lynch	07	New York
Spike Jonze	08	New York

To generate such a printout using MPL, you define the following MPL layout:

```
1   <mpl 4>
2   <layout title="All Employees"
3           print="P Employee"
4           originallayout="Standard">
5   <page "A4">
6   <paper>
7     <ruler name=employeesRuler [[85pt]|[65pt]|[80pt]]>
8
9     <repeating cursor=Employee script="L_Employee">
10      <header atStart=true>
11        <array ruler=employeesRuler justification=center fontsize=10>
12          "NAME" "EMPLOYEE N0" "COMPANY BASED IN";
13          <hline>;
14        <end array>
15      <end header>
16
17      <array ruler=employeesRuler justification=right>
18        .Name1:justification=left Employee.EmployeeNumber CompanyNameVar;
19      <end array>
20    <end repeating>
22  <end paper>
```

We will now go through this layout step by step and explain the central concepts in it.

It is hard not to notice that MPL is a markup language—it consists of *tags* that can have *attributes*, very much like XML. Some tags like `paper` or `repeating` are *parenthetical tags*; they have an opening and closing tag, in between which reside their children tags. Other tags, like `ruler` or `page` are *simple tags*—they do not have any children, and therefore do not have a closing tag.

MPL Header

Lines 1–5 specify what is known as the *MPL header*. Apart from the MPL version number (line 1) and the paper format (line 5), we specify what the title of this layout is (line 2), which standard Maconomy layout this layout is based on (line 4), and what the *print environment* of this layout (line 3) is.

Print Environment

A *print environment* gathers all of the data that is available to be used in the layouts based on this environment. This data can be either in the form of database *cursors*, which are collections of database records, or single variables that store the results of calculations that are performed on the data stored in the Maconomy database. Every MPL layout must be based on a predefined print environment in Maconomy.

Accessing Data from the Print Environment

The data in the print environment comes in very handy—cursors and variables are initiated with the right values and are ready to be used in your layout.

Because a cursor is a collection of database records, the best way to access the data that a cursor holds is to iterate over these records one by one. In MPL this is achieved by using the `repeating` tag, like in line 9 of our example, where we iterate over all of the records in the `Employee` cursor. For each iteration of the cursor, we can reference fields in the current record by name prepended with a dot. For example, to reference the field `EmployeeNumber` in the cursor `Employee` (line 18), we can just say `Employee.EmployeeNumber`. We are not required to mention the cursor name, though, because it can be implicitly resolved by the MPL engine, so we can just as well say `.EmployeeNumber` or, as shown in the example, `.Name1`, to reference the field `Name1`.

To print out the value of a variable, similarly, you only have to reference its name (this time without a preceding dot); for example in line 18 we print out the value of the variable `CompanyNameVar`, which holds the first part of the company name that the employee works for—in this case it is the location in which the company is based.

You might be wondering why the variable `CompanyNameVar` updates its value for every iteration of the `Employee` cursor. It must be recalculated for each `Employee`. This is exactly what happens; to be more precise, all of these calculations and updates take place in *scripts* that are executed for every cursor iteration, before any of the `repeating`'s children is executed. In our example this task is carried out by the script `L_Employee` referenced in line 9.

Scripts

Scripts are typically used to update the values of variables that are available in the print environment. However, they can also have database side effects, for example when reprinting invoices the script `P_JobInvoice` is called, which results in incrementing the `VersionNumber` field on the `Invoice` relation in question.

Defining a Print Layout

When we know how to print out the data that we want to present in the layout, it is time to think of how this data should be presented. In other words, we want to define the layout of our print.

In our simple example, we want to display the employee name, number, and company in a table. The table should have a header, so that when a page break occurs, the header is also printed on the next page, followed by the continuation of the table content.

To define how many columns the table should have, as well as what shape these columns should be (for example, width, justification, stretching, and so on), we can use the `ruler` tag. In line 7 we define a ruler that is a column specification for *arrays* that are used in MPL to represent tables. Our ruler definition consists of three columns having the width of 85, 65, and 80 points respectively (1pt = 1 point = 1/72 inch). In between the first and the second column, as well as the second and the third, there is a vertical line, specified in the ruler as the "|" character.

Having defined the ruler, we can now use it in the `array` definition. Lines 11-14 define an array inside a header, which will be the header of our table. The `ruler` attribute of the array is set to the `employeesRuler` ruler that we have just defined. Similarly, the main array printing the employee data (lines 17 - 19) points to `employeesRuler` as well.

Inside an array we specify semicolon-separated rows, each of them complying with the ruler definition. Since our `employeesRuler` specifies three columns, the rows inside the array should have three elements in each row.

Short Versions for Tags and Attributes

Some tags and attributes are used quite often. To avoid unnecessary typing, these tags and attributes can have short forms. For example, instead of enclosing an array definition in between the opening `<array ruler=employeesRuler ..>` and closing `<end array>` tags, we can also use the short form of the array tag and just type `{:employeesRuler..}`. Similarly, sometimes we can skip an attribute name and just specify its value, for example, instead of writing `<repeating cursor=Employee>`, we can just write `<repeating Employee>`. The "MPL Language Basics" section explains the short versions of attributes and tags in more detail. Our example would look somewhat like this when written using short forms for tags and attributes:

```
1   <mpl 4>
2   <layout title="All Employees"
3           print="P Employee"
4           originallayout="Standard">
5   <page "A4">
6   <paper>
7     <ruler employeesRuler [[85pt]|[65pt]|[80pt]]>
8
9     <repeating Employee script="L_Employee">
10      <header atstart+>
11        {:employeesRuler:justification=center:fontsize=10
12          "NAME" "EMPLOYEE N0" "COMPANY BASED IN";
13          <hline>;
14        }
15      <end header>
16
17      {:employeesRuler:justification=right
18        .Name1:left Employee.EmployeeNumber CompanyNameVar;
19      }
20    <end repeating>
22  <end paper>
```

Importing and Executing a Layout in Maconomy

After we have defined an MPL layout, we want to import into Maconomy. To this end, we can use any Maconomy client that supports the “Print Layout” window—we just execute the “Import Layout” action in there and select our layout.

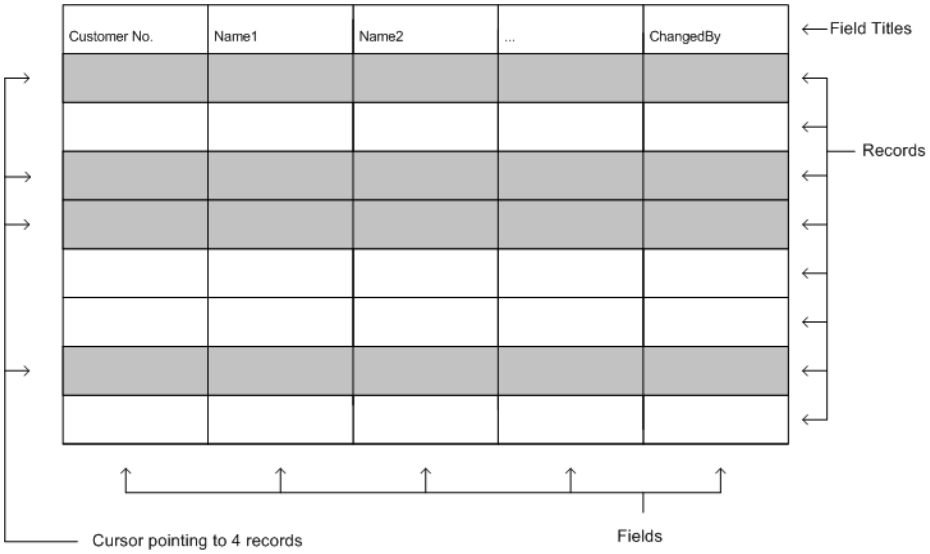
Upon importing, the layout is *compiled* by the MPL compiler. This means that the compiler tries to turn a given MPL text file into an internal MPL representation of this MPL layout. If the given text file represents a valid MPL layout, it gets imported and can be used when printing. Otherwise, error messages that point out which parts of the layout contain errors are reported.

Print Content In More Detail

In this section we describe more formally and in more detail the different kinds of data that are available in a print environment, that is, database cursors and variables. Moreover, as of version 4 of MPL you can fetch custom data from the Maconomy database (the data that has not been included in the print environment) using MQL queries and carry out custom calculations using the Expression Language.

Database Fields and Variables

- **Relations** — Relations are collections of data in the Maconomy database. In this context, a relation can be perceived as a *table of data*. Such a table consists of a fixed number of *columns* and a varying number of *rows*. Each row’s cells contain data, and each column has a unique name, for example, “Customer Number” or “Name1,” signifying the contents of each column.
- **Fields and Records** — In SQL terminology columns are called *fields* and rows are called *records*. Thus a relation is a collection of records, each of which has a number of named fields that contain data as shown in the following figure.



Maconomy is built on a number of different relations. For instance, one relation contains customers, and another relation contains vendors.

A *cursor* points to a number of the records that are contained in a relation. Sometimes a cursor points to all records, but often a cursor is subject to certain assigned *conditions* that ensure that only specific records are included.

When you use Maconomy, the system often displays values that are not contained in a relation. These values are calculated from data in the relations. For example, the discount amount on an item line is calculated from the discount percentage and the extended price.

- **Variables** — The result of these calculations is contained in a *variable* that can be included in printouts exactly like database fields.

Scripts

A script is a program fragment that is used to carry out calculations. Typically, a script ensures that a variable is calculated correctly, but it can also be used to make complicated calculations or updates in the database. For example, the script `S_Page` in the printout "Print Posting Journal" carries out updates in the database—the printout is actually responsible for the entire bookkeeping process.

Scripts cannot be changed using MPL. You must specify when predefined scripts are called, and the MPL compiler ensures that the calls are executed in the same way as in the original printouts.

To ensure that variables, which can be updated by scripts, hold valid values, as well as that the database side effects that scripts might have are executed in the right order, custom layouts must conform to the script structure of their respective original layouts. In other words, the use of scripts in a custom layout must be the same in its original layout—the scripts must be executed the same number of times and in the same order. To ensure this, we must compare the tags that have scripts attached, as well as all repeating and conditional blocks that the tags with scripts are embedded in, because they may change the number of times that the scripts are executed. For more details on script structure, see "Print Structure."

Custom Data and Custom Calculations

As of version 4 of MPL you can fetch custom data into an MPL layout using MQL queries that are run against Maconomy universes. MQL is a rich query language that is very similar to SQL, but more powerful in some respects. It is a *statically typed language*, which means that queries can be validated when you compile layouts. This validation step can capture a wide variety of errors, and, on the other hand, prove the absence of a certain class of errors (for example, the number of parameters that are passed to the query, their types, as well as incompatible field types of returned records, and so on). MQL queries are *reusable*, which means that you can declare a query once and instantiate its different incarnations with different values of parameters. Moreover, they run against Maconomy Universes, which predefine a lot of very useful joins that are quite hard to get right without knowing the Maconomy database schema intimately.

In addition to these custom data sources, MPL 4 also introduces a means to perform custom calculations by embedding the Expression Language[1] into MPL. You can define new values and variables and populate them with the results of arbitrarily complicated calculations.

MPL Language Basics

This section describes the basic elements of an MPL definition. It describes how the logical structure of a printout is defined using tags, and how the formatting of the individual tags can be modified using attributes.

Simple Tags

Simple tags describe a print element. For instance, `<text ...>` is a simple tag that specifies a text string.

[1] The Expression Language is used in other Maconomy layout languages like MDML as well.

Parenthetical Tags

Parenthetical tags occur in pairs that consist of a start tag and an end tag (corresponds to a start parenthesis and an end parenthesis). The contents are enclosed in the two parenthetical tags, for example, the description of an island is enclosed in the tags `<island>`.

```
... <end island>.
```

Attributes

Attributes can be assigned to all tags. Attributes contain additional information about the layout element that describes the tag. Attributes are specified between the tag name and “>” and are written as *`attribute name=attribute value`*.

Example

If you want the printout to contain the text string “Hello!,” you use the tag `text`. Furthermore, you should specify that the text (the title) should be “Hello!” You do this by setting the attribute to “Hello!” The complete code is:

```
<text title="Hello!">
```

Certain attributes are mandatory, whereas you can leave others out. An example of a mandatory attribute is the fact that `text` tags should be assigned a `title` attribute, which specifies the text that should be printed. The attribute that specifies the font of the text, however, is not mandatory, but has a default value that is used if the attribute is not specified.

Attribute Values

All attributes have a *type*. When specifying an attribute, you can only specify attribute values of the permitted type. Furthermore, the type is used for deriving the attribute used when a nameless attribute is used (see “Nameless attributes”).

MPL recognizes the following attribute types:

- *STRING* — Strings are specified in quotes and are typically used in connection with text to be shown on the printout, for example, “Hello!”
- *ID* — Identifiers are specified as is, that is, without quotes, for example, DateVar.
- *INTEGER* — Integers, for example, 3 or 19245. An INTEGER cannot be negative.
- *LENGTH* — Lengths are specified as an integer or decimal value followed by a unit of length. The basic length units are `pt` (points, 1/72 inch), `mm` (millimeter), `cm` (centimeter), and `in` (inch). Examples: `3pt`, `0.14cm`, and `12mm`. You cannot specify a negative length.

  You can also use `pagewidth` and `pageheight`, which are the width and height of the printable part of the paper for which the printout is formatted. The length specification `pageheight` can only be used for vertical height, and `pagewidth` can only be used for horizontal width. Example: `0.5pagewidth` specifies half the width of the paper less the left and right margins.

  You can define a length unit yourself, called grid (see “Grid” in “Advanced MPL”). If a grid is defined, you can, for example, specify:

```
1grid.
```

  Finally, you can specify lengths using constants. A number of constants are predefined by the system; you can find these in “Predefined Lengths” in “Advanced MPL.” You can also define your own constants; see “Length Constants” in “Advanced MPL.”
- *BOOLEAN* — Truth values. The only permitted values are true and false.

- *POS* — Positions. These are specified as a pair of lengths, corresponding to horizontal and vertical position, respectively. Example: `(2cm,4pt)`.
- *RULER* — Specification of column layout; see “Rulers—Column Definitions” in “Arrays.”
- *PARAMLIST* — Specification of a list of parameters used for links in MPL for Universe Reports. The parameter list contains a number of parameter/value pairs, separated by a comma. The parameter is a simple text value. The value can be a field name, a variable, or a text. Example with all three types of parameters:

```
[param1=.field1, param2=var1, param3="Text"]
```

  For more information, see “MPL for Universe Reports.”

- EXPRESSION — Represents the type of compound calculations that yield a value, either to be printed out, bound to a mutable variable (*var)* or an immutable value (*val)*, or assigned to a tag attribute of type *EXPRESSION*.

  Expressions are always delimited by curly braces, that is, { and }. Valid values of expressions are, for example, {123.4} and { x > 7}.

  For more information, see “Expressions.”

Short Forms

Certain tags are used so often that it is convenient to write them using a short form. For instance, you can abbreviate the tag `<text title="Hello!">` to `"Hello!"` Short forms are different from tag to tag. The short form of a given tag is therefore mentioned when the tag is described.

Short Attribute Forms

If a tag is written using a short form, you cannot specify attributes between the tag name and “>”. Instead, you specify attributes immediately after the short form, separated by colons. Example: the code

```
<text title="Hello!" fontname="Times" fontsize=12>
```

can be abbreviated to

```
"Hello!":fontname="Times":fontsize=12
```

You can also abbreviate the attribute value in logical attributes, that is, attributes that can be assigned the values true and false. Instead of

```
"Hello!":italic=true
```

you can write

```
"Hello!":italic+
```

Also, instead of writing

```
"Hello!":bold=false
```

you can write

```
"Hello!":bold-
```

Nameless Attributes

Some attributes are used very often. Therefore sometimes you can exclude their names to limit the amount of code to be written. Based on the type of attribute value, the MPL compiler automatically identifies the attribute. Such attributes are called *nameless attributes*.

Example

Instead of writing

```
<text title="Hello!">
```

you can write

```
<text "Hello!">
```

because the attribute `title` is nameless for the tag `text`. In its short form, the attribute `justification` for the tag `text` is nameless. Thus, instead of writing

```
"Hello!":justification=center
```

you can write

```
"Hello!":center
```

The description of each tag lists the attributes that can be nameless for the tag in question.

Tags and Attributes

The following sections describe all of the tags that are used in MPL. The description of each tag also lists the tag's attributes. You can find a complete list of all tags and their attributes in "Attribute List" in "Grammar."

Comments

You specify comments by entering two hyphens. The MPL compiler ignores rest of the line after the hyphens. Here is an example:

```
-- This is a comment
-- to a small layout with one text string
<mpl 2>
-- A layout called "Example"
<layout "Example" print="P Invoice"
   originallayout="Standard">
<page "A4">
-- Top of page
<paper>
   -- The printout contains the text: "Hello World":center
-- Bottom of page / end of layout
<end paper>
```

Original Structure Layout

If you are basically satisfied with a printout, but you want to change a few text strings and remove a couple of elements, you should export the original layout so that you get an MPL layout that corresponds to the chosen printout. This makes it easy to implement minor changes.

If you want make extensive changes to the layout, it is often helpful to export the structure layout. A structure layout is a viable template for a printout: The structure of an MPL layout. However, printouts with this layout are empty. After you have exported the structure layout you can fill in the contents and format the layout.

A structure layout can differ for various original layouts that have been assigned to the same printout. The concept of structure layouts does not apply to MPL for Universe Reporting.

Structure of an MPL Layout

This section describes the elements that are necessary in every MPL layout. Specifically, there must always be a heading that specifies the general properties of the printout, followed by a specification of the contents.

mpl

A layout description must always begin with

```
<mpl 2>
```

This tag specifies the version of the MPL language in which the layout is written (in this case, version 2). The MPL compiler supports layouts that were written in previous versions of the language. See “Enhancements in MPL Version 2” for a list of the new features in this version of MPL.

layout

The tag `layout` names the layout and specifies its association with original layouts. The following attributes are mandatory in standard MPL, but ignored in MPL for Universe Reports:

- `title` specifies the name of the layout. This name is used as the name of the layout in the table part of the window Print Layout and in layout selection pop-up fields in print windows. `title` has the type *STRING* and is both nameless and mandatory.
- `print` specifies the name of the printout. This is an existing name that can be found in the window Print Layout. `print` has the type *STRING* and is mandatory.
- `originallayout` specifies the name of the layout that was used as a template for the layout. For more information, see “Print Structure.” `originallayout` has the type *STRING* and is mandatory.

Example

```
<layout "My first layout"
   print="Print_Invoice"
originallayout="Standard" >
```

page

The tag `page` indicates which paper format should be used for the printout. The MPL layout heading must contain a page declaration. Paper formats are specified in the Paper Formats window in Maconomy.

If you, for example, specify an A4 paper format, no elements will be wider than the A4 paper (minus margins), and page breaks will take place when the A4 paper has been filled up vertically. When the MPL definition is compiled, no check is made as to whether the printout is actually printed on the specified type of paper.

The following attributes exist for `page`:

- `name` specifies the paper format for which the layout should be formatted. You can specify paper format names that are defined in the window Paper Formats. The attribute has the type *STRING* and is both nameless and mandatory.
- `orientation` takes the values `portrait` (the paper is standing on the short edge) or `landscape` (the paper is on the long edge). The attribute has the type *ID* and is nameless. The default value (that is, if the attribute is not specified) is `portrait`.

Example

If you want to format the printout for A4, type:

```
<page "A4">
```

If you want to format the printout for US Letter landscape, type:

```
<page "US Letter" landscape>
```

paper

The tag `paper` defines the printout as such. It is a parenthetical tag that surrounds all of the elements that make up the printout (excluding the front page; see below). The `paper` tag takes two attributes:

- `cursor` specifies a cursor name — One page is printed per record within the cursor.

  The template layout determines whether a cursor should be assigned, and, if so, which cursor. For more information, see “Print Structure.” The attribute has the type *ID* and is nameless.

- `script` is used to specify the name of a script to be run for each item in the specified cursor. As is the case with `cursor`, the template layout determines `script`. The attribute has the type *STRING* and is nameless.

Example

```
<paper cursor=Journal script="S_Page"
    ...
<end paper>
```

frontpage

Printouts can have a front page. This page is printed before the rest of the printout, if the printout is executed from a print window (a window with a “Print” button). These dialogs are usually displayed when you select options in the submenu Reporting or use the function “Print...” in the File menu. When you print using the function “Print This” in the File menu, the front page is not printed.

```
<frontpage>
    ...
<end frontpage>
```

The `frontpage` tag takes no attributes. As with other *stacking* tags (see “Stacking Tags”), front pages consist of definitions followed by elements. You cannot use field tags, headers, footers, repetitions, conditions, `newpage`, or `border` on front pages, and the content of the front page is limited to one page. If the information cannot fit on one page, an error message is displayed. Furthermore, you cannot use the `stacking` tag with scripts on a front page. The use of front pages is illustrated in “A Printout Example: Time Sheets.”

Visible Elements

A printout is constructed from the basic elements text, field, variable, island (frame), and line. The remainder of an MPL layout is all about formatting: The placement of the basic elements, whether part of the printout should be repeated for every item in a cursor, and whether parts of the printout should only be printed under certain circumstances.

This section describes the use of the basic elements, and will enable you to create your first simple layout, which we look at in the next section. In this section, we do not look at islands and lines; these are described later.

Predefined Data From the Print Environment

One kind of data that we can print out in an MPL layout is the data from the predefined print environment of our layout, that is, cursor fields and variables.

Fields

The `field` tag

```
<field attributes>
```

specifies that the contents of a database field are to be printed. This `field` tag takes one mandatory attribute:

- `data` specifies the name of the database field to be printed. The attribute is both nameless and mandatory.

Often you want to use the short form of `field` tags:

```
.field
```

Note the dot. This is the short form of

```
<field data=field>
```

If you want to specify from which cursor the field should be taken, you can specify the following attribute:

- `cursor` specifies the name of the cursor from which the field is to be taken. The attribute has the type *ID*.

Often you want to use the short form of `field` tags:

```
cursorname.field
```

Note the dot. This is the short form of

```
<field data=field cursor=cursorname>
```

If you do not specify a cursor name, the value is taken from the nearest cursor with a field of the specified name. However, it is good design practice to specify a cursor name if several cursors exist that contain the same field name.

The attributes for specifying the typography of the field (for example, `fontname`, `fontsize`, `bold`, `italic`, and `underline`) work in exactly the same way as for texts, and are thus not explained here. If you do not enter a font name and a size, the field has the font Helvetica 9 pt. Apart from this, you can specify the following attributes:

- `justification` specifies the justification of the field. `justification` is nameless in the short form and has the type *ID*. It can take the values `left`, `center`, and `right`. If you have

not specified `justification`, a field is justified according to its type: Fields of the types *INTEGER*, *REAL*, and *AMOUNT* are right-justified, whereas all other fields are left-justified.

- `width` functions just like the `width` attribute for text. However, the width of fields is set to the default value if `width` is not specified. This is due to the fact that the value of the field is not available at the time when the layout is being compiled.
- `indent` specifies an extra indentation of a field. The box will also be wider in accordance with the specified length. You cannot use the `indent` attribute in combination with right justification and centering. `indent` has the type *LENGTH*. You cannot use this attribute when the field appears in a canvas.
- `pos` specifies the positioning of a field element within canvas. The attribute is nameless and has the type *POS*. You can only use it when the field appears on a canvas, and is mandatory, if so (see “Canvas” for further information about the canvas).
- `zerosuppression` specifies whether the value 0 is to be printed out, if the value is zero for numeric fields, that is, fields of the type *AMOUNT*, *INTEGER*, or *REAL*. You cannot specify this attribute for fields of other types.
- `link` specifies that the text is to function as a link. This applies to MPL for Universe Reporting only. For more information, see “Links.”

In MPL 3 the attributes `wrap` (BOOLEAN), `lines` (INTEGER), `height` (LENGTH) are also supported and are used to control text wrapping around multiple lines, as well as adding explicit line breaks. See “wrap Attribute for <text>, <field>, and <var> Tags.”

In MPL 4, the <field> tag has been desupported. Its short form, however, is still valid as one of the short forms of the <eval> tag. For more detail, see “Field and Variable Reference Tags Desupported in MPL4.”

Example

```
Journal.Journalnumber:bold+:fontname="Courier"
```

Variables

The `var` tag

```
<var attributes>
```

specifies that the contents of a variable are to be printed. The variable tag takes one mandatory attribute:

- `data` specifies the name of the variable to be printed. The attribute has the type *ID*.

Often you want to use the short form of `var` tags:

```
var
```

which is the short form of

```
<var data=var>
```

The attributes `justification`, `fontname`, `fontsize`, `bold`, `italic`, `underline`, `width`, `indent` (replaced by `pos` in canvases), `zerosuppression`, and `link` work in the same way for variables as for fields (see the previous section).

In MPL 3 the attributes `wrap` (BOOLEAN), `lines` (INTEGER), and `height` (LENGTH) are also supported and are used to control text wrapping around multiple lines, as well as adding explicit line breaks. See “wrap Attribute for <text>, <field> , and <var> Tags.”

In MPL 4, the <field> tag has been desupported. Its short form, however, is still valid as one of the short forms of the <eval> tag. For more detail, see “Field and Variable Reference Tags Desupported in MPL4.”

Example

```
Companyname:indent=2cm:bold+
```

User defined data

In addition to printing out the predefined data, we can define the data—whether it is simple static text or complex arbitrarily expressions.

Texts

The `text` tag

```
<text attributes>
```

specifies that a text is to be printed, for example, a column heading or a label. The `text` tag takes one mandatory attribute:

- `title` specifies the text that is to be printed. The attribute has the type *STRING* and is both nameless and mandatory.

Often you want to use the short form of text tags:

```
"text"
```

which is the short form of

```
<text title="text">
```

Apart from this, you can specify the following attributes:

- `justification` specifies whether the text should be left-, center-, or right-justified.

  If the `width` attribute is also used, the justification takes place within the specified width. Otherwise, the justification occurs within the space that is made available by the surrounding elements. For instance, a right-justified text on the page is placed to the extreme right on the page, and a centered text in an island is placed in the middle of the island.

  If the attribute stretch on the island is false, the island will only be as wide as its contents, and if the text is the only content, you cannot tell whether justification is left, center, or right.

  `justification` has the type *ID* and is nameless. It can take the values left, center, and right. If justification is not specified, texts are left-justified.

- `fontname` specifies the font that is to be used for printing the text. All fonts that are available on the computer on which the layout is compiled can be used—remember that they must also be available on other computers and on the server, if printouts are executed on the server. No error message is displayed if an unknown font is specified. The attribute has the type *STRING*. The default font is Helvetica. For more information, see “Font Administration in Maconomy” in the Maconomy Administrator’s Guide.

- `Fontsize` specifies the size of the selected font. The attribute has the type *INTEGER*. The default font size is 7pt.

- `bold` specifies that the text should be printed in **boldface**. The attribute has the type *BOOLEAN*.
- `italic` specifies that the text should be printed in *italics*. The attribute has the type *BOOLEAN*.
- `underline` specifies that the text should be printed underlined. The attribute has the type *BOOLEAN*.
- `width` specifies the width of the box that surrounds the text. The attribute has the type *LENGTH*.

  If this attribute is not specified, the remaining part of the layout is formatted according to the actual width of the text. The text box is hence adjusted to fill in the available width space.

  If `width` is specified, the remaining part of the layout is formatted as if the text had the specified width, and the text box is not adjusted. This has the following consequences:

  - If the length is specified to be shorter than the text's own width, the text is cut off.
  - Justification is relative to the specified width.
  - If the text appears in a column, this column is given a width that ensures space for the width specified by the attribute width.
- `indent` specifies an extra indentation of the text. The text box will also be wider in accordance with the specified length. You cannot use the `indent` attribute in combination with right justification and centering. `indent` has the type *LENGTH*. You cannot use this attribute when the text appears in a canvas (see "Canvas" for further information about the canvas).
- `pos` specifies the positioning of a text element of a canvas. The attribute is nameless and has the type *POS*. You can only use it when the text appears on a canvas, and is mandatory, if so (see "Canvas" for further information about the canvas).
- `link` specifies that the text is to function as a link. This applies to MPL for Universe Reporting only. For more information, see "Links."

  In MPL 3 the attributes `wrap` (BOOLEAN), `lines` (INTEGER), and `height` (LENGTH) are also supported and are used to control text wrapping around multiple lines, as well as adding explicit line breaks. See "wrap Attribute for <text>, <field>, and <var> Tags."

Example

```
"Hello world!":center:width=4cm:fontsize=18
```

See also "Alternative Text Tag" in "Advanced MPL."

Arbitrary Expressions

The `eval` tag, introduced in MPL 4,

```
<eval {expression} other_attributes>
```

specifies that the given expression should be evaluated and the result value printed. The `eval` tag takes one mandatory attribute:

- `expression` of type *EXPRESSION*, which denotes an arbitrary *Expression Language* construct that is evaluated and the result of that evaluation is printed. For more information on expressions, see “Expressions” and “Literal Values for Different Types
- When declaring a `var` or a `val` or just using literals as values in expressions, it is useful to know how the different literals look for values of different types:

Type	Comma separated example values
INTEGER	`457, 77, -123`
REAL	`41.789, 99.4`
AMOUNT	`AMOUNT(99.74), AMOUNT(6.45)`
BOOLEAN	`true, false`
DATE	`DATE(2013, 12, 25), DATE(1987, 2, 15)`
TIME	`TIME(12, 23, 58), TIME(23, 15, 33)`
STRING	`"Text”, "example string”`
POPUP	`GenderType'Male, CountryType’France`

Standard FunctionsWhen declaring a `var`, `val` or just using literals as values in expressions, it is useful to know how the different literals look like for values of different types:

Type	Comma separated example values
INTEGER	`457, 77, -123`
REAL	`41.789, 99.4`
AMOUNT	`AMOUNT(99.74), AMOUNT(6.45)`
BOOLEAN	`true, false`
DATE	`DATE(2013, 12, 25), DATE(1987, 2, 15)`
TIME	`TIME(12, 23, 58), TIME(23, 15, 33)`
STRING	`"Text”, "example string”`
POPUP	`GenderType'Male, CountryType’France`

The preferred form of evaluating and printing out expressions is the short form of the `eval` tag, for example,

```
^{addMonths(currentDate, 20)}
```

The full form of the above `eval` tag is:

```
<eval expression={addMonths(currentDate, 20)}>
```

There is no compelling reason, though, to use the full form, and it is in the language mostly for completeness reasons.

The attributes `justification`, `fontname`, `fontsize`, `bold`, `italic`, `underline`, `width`, `indent` (replaced by `pos` in canvases), `zerosuppression`, `link`, `wrap`, and `lines` work in the same way for `eval` in MPL 4 as for `field` and `variable` tag for MPL 3 (see the previous sections).

Example MPL Layout

This section contains an example of the creation of an MPL layout. The layout developed in the example is a new layout for the printout “Print Warehouse Information Card” and will be based on the layout “Standard.”

Getting Started

When you want to define a new layout, it is common practice to export either the structure layout or the original layout for a layout with the desired functionality. Thus to define a new layout for the layout “Print Warehouse Information Cards” we export the structure layout for the layout “Standard” from the window Print Layout in the Maconomy client.

The result of the export is a file in which the printout name and original layout name have been correctly set, and cursor and script names have been attached to blocks as required. A list of all available variables and fields is provided in the document “List of variables in MPL.” This document is updated for each version of Maconomy and can be found on the Maconomy Partner web site.

Header

The printout should be formatted for A4 paper and named “Simple.” Thus you should insert the following layout heading:

```
<mpl 2>
<layout "Simple"
   print="Print_Inventory_Info_Card"
   originallayout="Standard">
<page "A4">
```

Page

The layout “Standard” for the printout “Print Warehouse Information Cards” prints one warehouse information card per page. This is due to the specification of a cursor in the `paper` tag. The cursor is “`Inventory`.” A script is also specified, namely “`Inventory_PageScript1`.” The resulting code is

```
<paper cursor=Inventory script="Inventory_PageScript1">
   ...
<end paper>
```

Note that the heading specified in the last section, combined with using the `paper` tag mentioned before, is very similar to the resulting layout of an export of the structure layout.

Contents

The procedure for placing elements side by side in the structure is described later in this document, so we will therefore use the following structural form for this exercise:

```
descriptive text
field/variable
field/variable
```

where the descriptive text is formatted as bold text. Enter the following code to print out name and address:

```
"Name":bold+
```

```
    .InventoryName:indent=1cm
"Address":bold+
    .Address1:indent=1cm
    .Address2:indent=1cm
    .Address3:indent=1cm
    .Address4:indent=1cm
```

Complete Layout

Now we have the complete layout:

```
<mpl 2>
<layout "Simple"
          print="Print_Inventory_Info_Card"
          originallayout="Standard">
<page "A4">
<paper cursor=Inventory script="Inventory_PageScript1">
    "Name":bold+
    .InventoryName:indent=1cm
    "Address":bold+
    .Address1:indent=1cm
    .Address2:indent=1cm
    .Address3:indent=1cm
    .Address4:indent=1cm
    "Attention":bold+
    .AttPerson:indent=1cm
    "Company No.":bold+
    .CompanyNumber:indent=1cm
    "Company Name":bold+
    CompanyName1Var:indent=1cm
    "Phone":bold+
    .Telephone:indent=1cm
    "Fax":bold+
    .Telefax:indent=1cm
    "Telex":bold+
    .Telex:indent=1cm
<end paper>
```

The result of importing this definition and printing with the new layout is a number of pages, each of which looks like this:

Name
Standard
Address
The Docks
Harbourtown, Berks

Attention

Company No.
20
Company Name
Advertising Agency 1
Phone
234 521555
Fax
234 521556
Telex
warehouse@20.com

Basic Tags

This section contains a detailed description of a number of most commonly used MPL tags.

Stacking Tags

A *stacking* tag is a parenthetical tag that is used for placing elements on top of each other. Most parenthetical tags in MPL are stacking tags. This means that all elements in such a tag are placed on top of each other—they are stacked. We have already seen an example of a stacking tag: In the `paper` tag, the things written between `<paper>` and `<end paper>` are all placed below each other. The only parenthetical tags in MPL that are not stacking are `row` and `canvas`.

The contents of stacking tags are a number of elements that are to be printed. Before these elements, you can specify various definitions (see “Rulers—Column Definitions” in “Arrays” and “Length Constants” in “Advanced MPL” for further information about these definitions). These definitions apply within the tag in which the definitions are made. The stacking tag in which the definition appears is called the definition’s *scope*.

Stacks

The parenthetical tag

```
<stack attributes>
    ...
<end stack>
```

is the simplest example of a stacking tag. Without attributes it is used to ensure that elements are stacked on top of each other. This can be useful in connection with rows that make it possible to place two stacks next to each other.

`stack` cannot appear in a canvas or in rows if the `script` attribute has been specified, or if the stack has a header or footer. For more information, see “Print Structure.”

`stack` has the following attributes:

- `script` specifies the name of a script (program) that is to be run before the contents are printed. Typically, the script takes care of initializing variables that are to be used later in the printout. Scripts can only be used in the same way as in the original layout. For more information, see “Print Structure.” `script` is nameless and has the type *STRING*.
- `baseline` is used to specify the baseline of the stack. If two elements are placed next to each other, they can be placed in such a way that their baselines are aligned. If `baseline` is set to `top`, the stack inherits the baseline of the upper element—that is, if a stack is placed next to a text, the text appears at the same baseline as the upper element of the stack. If `baseline` is set to `bottom`, the stack inherits the baseline of the lower element. The default value for `baseline` is `bottom`, and `baseline` has the type *ID*.

  The baseline for texts, variables, and fields is determined by the font used. Here the baseline is the line on which the text is printed. The baseline is placed immediately below the bottoms of letters such as “a” and “b,” while letters such as “q” and “g” extend below the baseline.

- `height` specifies the height of the stack. This only influences the placing of subsequent elements. If the contents of the stack cannot be placed within the specified margin, or if the height of the contents cannot be determined (due to conditions or repetitions), an error message

is displayed. Furthermore, you cannot use the `height` attribute if the stack contains other stacks with a header or a footer, or if the `script` attribute has been set—the system displays an error message. If `height` is not specified, the height of the stack is dictated by its content. The attribute has the type *LENGTH*.

- `width` specifies the stack's width. This influences the positioning of elements to the right of the stack as well as the justification of elements in the stack. If the contents of the stack cannot be placed within the specified margins, the system displays an error message. If `width` is not specified, the stack's width is stretched to fit the space available. The attribute has the type *LENGTH*.
- `indent` specifies extra indentation of the stack (all elements contained in the block are indented). The attribute has the type *LENGTH*. This attribute cannot be used when the stack appears in a canvas.
- `pos` specifies the positioning of the stack in a canvas. The attribute is nameless and has the type *POS*. You can only use it when the stack appears in a canvas, and is then nameless and mandatory. See also "Exact Positioning of Elements" in "Canvas."

Stacks cannot appear in a canvas or in rows if the `script` attribute has been set, or if the stack is provided with a header or a footer. For more information, see "Print Structure."

Example

```
<stack>
    "Upper"
    "Lower"
<end stack>
```

Islands

The parenthetical tag

```
<island attributes>
    ...
<end island>
```

defines a stacking tag. The tag results in the drawing of a frame around the contents and is used for highlighting or grouping information on printouts.

The behavior of an island can be modified using the following attributes:

- `stretch`. If this attribute is set to `false`, the island's width is determined by its contents. If the attribute is not set or is set to `true`, the island's width is determined by its surrounding elements. If the island is defined on the page, it is stretched across the entire width of the page. If the island is positioned in a column, it is stretched to the width of the column. The attribute has the type *BOOLEAN*.
- `baseline` is used to specify the baseline of the island. As for the `stack` tag, you can set `baseline` to either `top` or `bottom` to specify whether the island should inherit the baseline of the upper or lower element. In addition, you can set `baseline` to `title`, meaning that the island inherits the baseline of the island title. The default setting is `bottom`, and the attribute has the type *ID*.
- `justification` only makes sense if the `width` attribute has been set or if the `stretch` attribute has been set to `false`. In these cases, the island may be smaller than the space allotted to it by the surrounding elements. It can therefore be placed to the left, in the middle

(default), or to the right. `justification` has the type *ID* and can take the values `left`, `center`, and `right`.

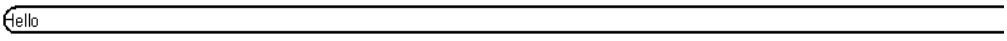
- `rounded` is used to specify whether the corners of the island should be rounded (default) or square. The attribute has the type *BOOLEAN*.
- `topmargin` is used to specify the upper inner margin of the island, that is, the distance between the upper line of the island and the top of the upper element. The attribute has the type *LENGTH*.
- `bottommargin` is used to specify the lower inner margin of the island, that is, the distance between the lower line of the island and the bottom of the lower element. The attribute has the type *LENGTH*.
- `leftmargin` is used to specify the left inner margin of the island, that is, the distance between the left line of the island and left side of the contents. The attribute has the type *LENGTH*.
- `rightmargin` is used to specify the right inner margin of the island, that is, the distance between the right line of the island and right side of the contents. The attribute has the type *LENGTH*.
- `indent` is used to indent the island and can therefore you can only use it if the `justification` attribute has been set to `left`. The attribute has the type *LENGTH*.
- `pos` is used to specify the position of an island in a canvas. The attribute is nameless and has the type *POS*. You can only use it when the island is a part of a canvas, and in those cases it is nameless and mandatory. See also “Exact Positioning of Elements” in “Canvas.”
- `height` is used to specify the height of the island’s contents. If the island’s contents are higher than the value specified, the compiler displays an error message. The attribute has the type *LENGTH*.
- `width` is used to specify the width of the island’s contents. If the island’s contents are wider than the value specified, the compiler displays an error message. If `width` has been set, the `stretch` attribute value is ignored. The attribute has the type *LENGTH*.

You can also use island to frame rows (see “Rows in Stacking Tags” in “Arrays”), but only if the attributes `leftmargin`, `rightmargin`, and `indent` have been set to zero (or not set, because zero is the default value). If these attributes have not been set correctly, you cannot refer to rulers that have been defined outside the island.

Example

```
<island>
   "Hello"
<end island>
```

will result in the following printout



The MPL fragment

```
<island leftmargin=1cm
   rightmargin=1cm
   topmargin=5mm
   bottommargin=5mm
   stretch->
   "Hello"
<end island>
```

will result in the following printout

For further information and an illustration of the lengths that can make up the formatting of an island, see “Island Lengths” in “Tips and Tricks.”

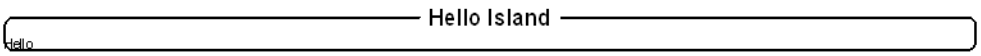
Island Titles

You can assign a title to an island. Such a title will be printed as part of the top line of the island. The `island` tag has a number of attributes that determine the behavior of the title:

- `title` is used to specify the island title text. The attribute has the type *STRING* and is nameless.
- `titlejustification` is used to specify whether the title should be placed on the left, in the middle, or to the right on the top line of the island. The possible values are thus `left`, `center` (default), and `right`.
- `fontname`, `fontsize`, `bold`, `italic`, and `underline` are used to specify the text format. See the description of the `text` tag for further information on the use of these attributes. The default font is 16pt, bold, Helvetica.

Example

```
<island title="Hello Island" bold+ fontsize=12>
   "Hello"
<end island>
```



Repetitions, Conditions, and While

The result of executing an MPL print definition of course depends on the data that is stored in the Maconomy system. In this manual, we started out by setting up a print definition that would print selected setup information about warehouses. When the print definition is executed, that is, not when it is compiled, the data on the resulting printout is an exact representation of data in the Maconomy system.

An MPL print definition also depends on other forms of data. When you print an invoice, naturally you want to print all invoice lines, and the number of invoice lines depends on data that is in the Maconomy system at the time of printing. To do this, you can use the `repeating` tag that prints out its contents for each record in a specified cursor. Similarly, you may want only to print lines with cash discount if the customer actually receives a discount. To do this, you can use the `conditional` tag that prints out its contents subject to a condition specified in the MPL definition.

You cannot define your own repetitions and conditions in MPL, because the MPL compiler demands that you follow the structure of the original layout. For more information, see “Print Structure.” To ensure this, it is recommended that you base your layout on a structure layout that contains the necessary conditions and repetitions.

Repeating Blocks

The parenthetical tag

```
<repeating attributes>
```

```
    ...
<end repeating>
```

is used for repeating the contents for each record in a given cursor. The use of `repeating` iterating over the predefined standard cursors must follow the way in which it is used in the original layout. For more information, see “Print Structure.” The following describes how the block is repeated is specified using the `cursor` attribute:

- `cursor` specifies the name of the cursor, which defines the repetition. It can be either a standard predefined cursor or a custom MQL cursor defined in an MPL 4 layout. The contents of the repetition (between `<repeating ...>` and `<end repeating>`) will be printed for each record in the cursor, and fields from the cursor will be available within the tag. `cursor` has the type *ID* and is mandatory and nameless.

`repeating` also has the following attributes:

- `groupby` is used to define the `cursor` attribute. It enables grouping of the records in the cursor. Records where the fields that are marked by the `groupby` attribute are alike will become grouped, and each group can be assigned a common heading.

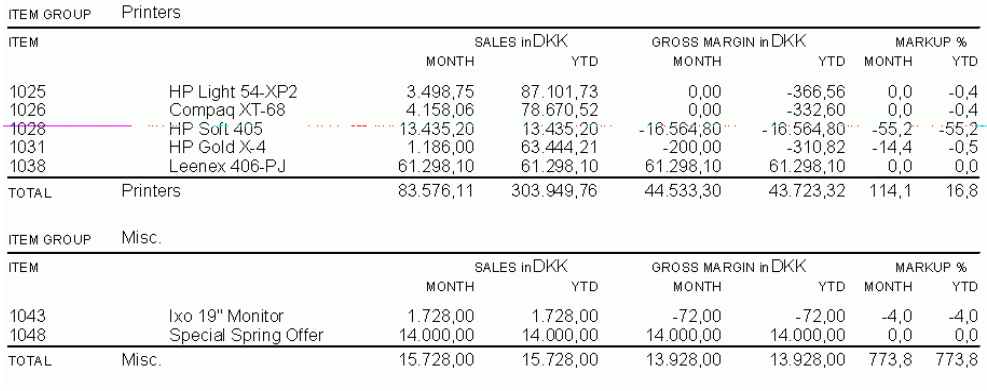
  An inner repeating block for the same cursor but without the `groupby` attribute must always be placed inside a repeating block with the `groupby` attribute—meaning that with several nested repeating blocks with the same cursor, the innermost block should have no `groupby` attribute. A repetition within a repetition block will terminate when the value of the fields specified by the `groupby` attribute of the surrounding `groupby` change. Note that the `groupby` attribute must be used in exactly the same way as in the original layout.

  The following example is from the printout “Print Item Group Statistics,” layout “Standard”:

```
<repeating cursor=ITEM
       groupby=[ItemGroup]
           script="ITEMGROUPBLOCK">
               .ItemGroup:bold+
                   ...
                   <repeating cursor=ITEM script="ITEMBLOCK">
                       .ItemText1
                   <end repeating>
<end repeating
```

  The outer repetition block will begin with an item record in some item group (and print it). As long as the item group does not change, the inner repetition block will print out the item text, but as soon as the inner repetition block reaches an item record in a new item group, the repetition will terminate, and the outer repetition block will take over, so that a new item group will be printed.

  The printout “Print Item Group Statistics” will therefore look like this:



ITEM GROUP	Printers						
ITEM		SALES in DKK		GROSS MARGIN in DKK		MARKUP %	
		MONTH	YTD	MONTH	YTD	MONTH	YTD
1025	HP Light 54-XP2	3.498,75	87.101,73	0,00	-366,56	0,0	-0,4
1026	Compaq XT-68	4.158,06	78.670,52	0,00	-332,60	0,0	-0,4
1028	HP Soft 405	13.435,20	13.435,20	-16.564,80	-16.564,80	-55,2	-55,2
1031	HP Gold X-4	1.186,00	63.444,21	-200,00	-310,82	-14,4	-0,5
1038	Leenex 406-PJ	61.298,10	61.298,10	61.298,10	61.298,10	0,0	0,0
TOTAL	Printers	83.576,11	303.949,76	44.533,30	43.723,32	114,1	16,8

ITEM GROUP	Misc.						
ITEM		SALES in DKK		GROSS MARGIN in DKK		MARKUP %	
		MONTH	YTD	MONTH	YTD	MONTH	YTD
1043	Ixo 19" Monitor	1.728,00	1.728,00	-72,00	-72,00	-4,0	-4,0
1048	Special Spring Offer	14.000,00	14.000,00	14.000,00	14.000,00	0,0	0,0
TOTAL	Misc.	15.728,00	15.728,00	13.928,00	13.928,00	773,8	773,8

Be aware that incorrect use of repeating blocks might crash the Maconomy server. Consider the following MPL-fragment:

```
<repeating cursor=myCursor groupby=[myField]>
       -- (1) Fields of myCursor will assume the "first" record values
       -- for each group of records (with identical myField)
       <repeating cursor=myCursor>
          -- (2) Fields of myCursor will assume all record values
       <end repeating>
          -- (3) Fields of myCursor cannot be accessed. This should result
          -- in an MPL compilation error, but unfortunately
          -- crashes the server
<end repeating>
```

Fields from the cursor named `myCursor` can only be used in subfragments (1) and (2) and cannot be used in subfragment (3). Note that only some fields make sense in (1), notably `myField` and derived fields.

The values for the `groupby` attribute are specified as [ID_1,...,ID_2] where each ID specifies a field name in the cursor. The attribute is nameless.

When repeating over an MQL cursor in MPL 4, the use of `groupby` is strongly discouraged in favor of defining a multilevel MQL query and using nested repeating to iterate over groups. For more detail, see “Multilevel Queries.”

- `script` specifies the name of a script (program) that is to be run before the contents are printed. Typically, the script takes care of initializing variables that are to be used later in the printout. Scripts can only be used in the same way as in the original layout. `script` is nameless and has the type *STRING*.
- `skipHeaderFooterIfEmpty` specifies whether the header and footer should be skipped if no iterations took place, that is, when the cursor had no records. It is of type *BOOLEAN* and defaults to `false`. If you want to reference cursor fields of this repeating in either the header or the footer, you must set this attribute to `true`.
- `height` specifies the height of each iteration of the repetition. If the contents cannot be placed within the specified height, or if the height of the contents cannot be determined (due to conditions or repetitions), an error message is displayed. If `height` is not specified, the height of the stack will be dictated by its content. The attribute has the type *LENGTH*.
- `width` specifies the repetition’s width. This influences the positioning of elements to the right of the stack as well as the justification of elements in the stack. If the contents of the stack cannot be placed within the specified width, the system will issue an error. If `width` is not

specified, the stack's width will be stretched to fit the space available. The attribute has the type *LENGTH*.

- `indent` specifies extra indentation of the repetition (all elements contained in the block are indented). The attribute has the type *LENGTH*.

The use of the attributes `cursor`, `groupby`, and `script` must be exactly as in the original layout, when iterating over standard, predefined cursors. For more information, see "Print Structure."

Repetitions cannot appear in a canvas (see "Canvas") or in rows (see "Arrays").

Conditionals

The parenthetical tag

```
<conditional attributes>
    ...
<end conditional>
```

is used to specify elements which should only be printed under some circumstances. Whether the contents should be printed is determined by the value of exactly one of the following attributes:

- `variable` specifies the name of the variable that must be `true` for the contents to be printed. In the print definition for "Print Posting Journal,",for instance, a conditional with the variable "IncludeTotals" is used to determine whether totals should be printed. This variable is set to `true` when you select the field "Include Totals" in the client Print window. The `variable` attribute is nameless and has the type *ID.*
- `field` (*supported in MPL 3 and 4 only*) specifies the name of the field that must be `true` for the contents to be printed. This attribute goes hand in hand with the optional `cursor` attribute that the specified field must belong to. If not specified, `cursor` is resolved automatically to the first cursor in the scope chain that contains the `field` in question. Both `field` and `cursor` attributes are of type *STRING.*
- `expression` (*supported by MPL 4 only*) specifies a Boolean expression that must evaluate to `true` for the contents to be printed. `expression` is nameless and has the type *EXPRESSION.*

The remaining attributes of the tag `<conditional>` include:

- `script` specifies the name of a script (program) that is to be run before the contents are printed (that is, if the condition is true). `script` is nameless and has the type *STRING*.
- `negate` is of the type *BOOLEAN*. If it is set to `true`, the value of the conditional will be negated—that is, the content of the conditional will be printed if its value is `false`. This applies to MPL 3 and MPL for Universe Reports only.
- `baseline` is used to specify the baseline of the conditional block. As for stacks, `baseline` can be set to either `top` or `bottom`, specifying whether `baseline` should be inherited from the upper or lower element.
- `height` specifies the height of the conditioned block. If the condition is `true` (and the contents are printed), the following elements will treat the conditioned block as if it had the given height. If the contents do not fit in the allotted space, or the height of the contents cannot be determined (see the `height` attribute for `stack`), an error message is displayed. If the attribute is not specified, the height is dictated by the contents. The attribute has the type *LENGTH*

- `width` specifies the width of the conditioned block. This influences the positioning of elements to the right of the conditional as well as the justification of elements in the conditional. If the contents of the conditional block cannot be placed within the specified space, the system will display an error message. If `width` is not specified, the conditional's width will be stretched to fit the space available. The attribute has the type *LENGTH*.
- `indent` specifies extra indentation of the conditional block (all elements contained in the block are indented). The attribute has the type *LENGTH*.

The use of conditions is subject to certain restrictions that are dictated by the original layout. See "Print Structure."

A conditional cannot appear in a canvas. At most, one conditional can appear in any given row (this restriction does not apply to MPL 3).

In MPL 3 the attributes `field` (STRING) and `cursor` (STRING) are also supported and are used to make a conditional depend on values of a database field. For more information on conditional changes in MPL 3, see "Conditionals" and "Skipping False Conditionals." MPL 4 adds expressions as a possible means of specifying the guarding condition for conditionals. For more detailed description of expression, see "Expressions."

Example 1

In this example we use a nameless attribute `variable` to specify the condition guarding this `conditional`:

```
<conditional EUSalesVar>
   "VAT NO."
   PayerVATNumber:indent=1cm
<end conditional>
```

Example 2

In this example we use a nameless attribute `expression` to specify the condition guarding this `conditional`:

```
<conditional {AgeVar > 75 && congratulateSeniorsVar} >
   "Congratulations to " .EmployeeName
   "on having lived for"  AgeVar "years!"
<end conditional>
```

While Loops

The parenthetical tag

```
<while attributes>
   ...
<end while>
```

is a looping construct in MPL—that is, it keeps executing its children as long as its guarding `condition` attribute evaluates to `true`. `while` loops can be used together with the Expression Language to perform arbitrary calculations. On top of that, they can be also employed to repeat the execution of a chosen part of an MPL layout while a certain condition holds. In the latter scenario, though, you should remember that `while` loops are part of the layout script structure in the same ways as repeating blocks and conditionals are, which basically means that when you customize an existing layout you cannot add a `while` loop around a block of MPL code from the

original layout that might potentially call scripts (at least one of the tags has a script attached). For more details on this restriction, see “Print Structure.”

The `<while>` tag has the following attributes:

- `condition` specifies the condition that must be `true` for the next iteration of the loop to execute. The attribute is mandatory, nameless, and of type *EXPRESSION.*
- `skipHeaderFooterIfEmpty` specifies whether the header and footer of this `while` loop should be skipped if no iterations took place, that is, when the guarding condition evaluated to false the first time the loop was executed. The attribute is of type *BOOLEAN* and defaults to `false`. When set to `true`, both the header and the footer are not printed when no iterations took place.

`while` loops support `indent`, `height`, and `width` attributes in the exact same way as repeating blocks.

The <while> tag has been introduced in MPL 4 as of TPU 16 SP2.

Example 1

The following code implements the insertion sort algorithm, sorting characters in the input `numbers` string in ascending order. It goes through the characters in the `numbers` string one by one, and inserts them in ascending order into the resulting `sortedNumbers` string. Because `sortedNumbers` string is at all times sorted, the inner `while` loop stops as soon as it finds an element bigger than or equal to the current one—this is exactly the right spot to insert the current element.

```
<var numbers {"918273645"}>
<var sortedNumbers {""}>

<var currentIndex {0}>
<val numbersLength {length(numbers)}>

-- Iterate over all the numbers
<while {currentIndex < numbersLength}>
  <val currentElem {charAt(numbers, currentIndex)}>
  <var insertionIndex {0}>

  -- Find insertionIndex to insert currentElem into the resulting sortedNumbers
  <while {    insertionIndex < currentIndex
          and charAt(sortedNumbers, insertionIndex) < currentElem} >
    -- Iterate as long as currentElem is bigger than the elements in
    -- sortedNumbers
    <assign insertionIndex {insertionIndex +1}>
  <end while>

  -- Insert currentElem into sortedNumbers at the found insertionIndex
  <val prefix {substring(sortedNumbers, 0, insertionIndex)} >
  <val postfix {substring(sortedNumbers, insertionIndex)}>
  <assign sortedNumbers {prefix + currentElem + postfix}>

  <assign currentIndex {currentIndex +1}>
<end while>

^{"Input numbers: " + numbers} -- prints 918273645
^{"Sorted numbers: " + sortedNumbers} -- prints 123456789
```

Example 2

In addition to performing calculations, you can also use `while` loops to repeat executing an MPL code snippet as long as a certain condition is satisfied.

For example, suppose we have a layout that lists all of the employees in a repeating over the `Employee` cursor. A skeleton of it could look somewhat like this:

```
<repeating Employee>
…
<end repeating>
```

We want to print the contents for each employee twice, adding the text “Copy 1 of 2” the first time and “Copy 2 of 2” the second time. To this end, we can modify the above snippet in the following way:

```
<var copyNumber {1}>
<while {copyNumber <= 2}>
  <repeating Employee>
   …
  <end repeating>


  ^{"Copy " + copyNumber + " of 2”}
  <assign copyNumber {copyNumber +1}>
<end while>
```

Horizontal Lines and Spaces

Inside a stacking block, it is sometimes convenient to control horizontal spacing between the stacked elements as well as to print horizontal lines in between them. This can be achieved be means of the `<skip>` and `<hline>` tags in MPL.

skip

The simple tag

```
<skip attribute>
```

is used to insert extra vertical space between elements and has the following attribute:

`height` is used to specify the height of the extra space that is to be inserted. The attribute is mandatory and nameless and has the type *LENGTH*.

Example

```
<island>
   "Hello!"
<end island>
<skip 5mm>
<island>
   "Bye bye!"
<end island>
```

Here the `skip` attribute is included in the block to ensure space between the two islands.

hline

The simple tag

```
<hline attribute>
```

is used for inserting a horizontal line in a stacking tag. The line will be stretched across the width of the stacking tag. The tag has one attribute:

`multi` is used to specify the number of lines that are to be inserted. For example, set the `multi` attribute to `2` to insert a double line underneath a total. The attribute has the type *INTEGER*. If not specified, one line is inserted.

Example

```
<stack width=1cm>
   "123":right
   "+234":right
   <hline>
   "357":right
   <hline multi=2>
<end stack>
```

This results in the following printout:

Arrays

Arrays are used for arranging elements into rows and columns. Because arrays can be used within arrays, they offer almost unlimited possibilities for formatting data.

Columns and Rows

array

The parenthetical tag

```
<array attributes>
   ...
<end array>
```

specifies an array. The contents of an `array` tag must be rows. The rows are placed above each other, and the elements in the rows are arranged in columns. You can use the following short form:

```
{ shortattributes
   ...
}
```

`array` has the following attributes:

- `ruler` is used to specify the format of the columns of the array. Rulers are described in "Rulers—Column Definitions." You can specify a ruler value or an ID that refers to a ruler with a specified name. The attribute is nameless.
- `baseline` is used to specify the baseline of the array. As for the tag `stack`, you can set `baseline` to `top` or `bottom`,specifying whether the baseline is inherited from the upper or lower row. The default value is `bottom`. The attribute has the type *ID*.

- `height` specifies the height of the array. This only influences the placing of the elements that follow the array. If the contents of the array cannot be placed within the specified margin, the system will display an error message. If `height` is not specified, the height of the array is dictated by the contents of the array. The attribute has the type *LENGTH*.
- `width` specifies the array's width. If the contents cannot be placed within the specified width, the system will display an error message. If none of the columns is stretchable, the width of the array will be the sum of the columns' widths, whether width is specified or not. If one or more columns are stretchable, the array is given the specified width. The attribute has the type *LENGTH*.
- `indent` specifies extra indentation of the array. The attribute has the type *LENGTH*. This attribute cannot be used when the array appears in a canvas or if the array uses a named ruler.
- `pos` specifies the positioning of the array in a canvas. The attribute is nameless and has the type *POS*. You can only use it when the array appears in a canvas, and in that case it is nameless and mandatory. See also "Exact Positioning of Elements" in "Canvas."

Following the description of the `row` tag in the next section, you will find a number of examples of the use of arrays.

row

The parenthetical tag:

```
<row attributes>
    ...
<end row>
```

is used to specify rows. Often you will want to use the short form:

```
...;shortattributes
```

A row consists of elements placed next to each other. A row cannot contain repetitions and can contain only one condition.

- `align` is used to specify how the elements in the row are placed next to each other. The attribute can be given the following values:
  - `baseline` is used to specify that the elements in the row are placed in accordance with the baseline. A simple element's baseline—that is, texts, fields, and variables—is selected on the basis of the font (characters such as "w" and "t" are placed on the baseline, while "g," "j," and "q" extend a little below the baseline). Baseline for a stacking tag is decided by this tag's baseline attribute. If no value is specified for align, baseline is standard.
  - `top` is used to specify that the elements should be placed in such a way that the upper edge of the elements are aligned.
  - `center` is used to specify that the elements should be placed in such a way that the middles of the elements are aligned.
  - `bottom` is used to specify that the elements should be placed in such a way that the lower edges of the elements are aligned.

  The `align` attribute has the type *ID* and is nameless in short as well as in long form.
- `height` specifies the height of the row. This only influences the placing of subsequent rows. If the contents of the row cannot be placed within the specified margin, the system will display an error message. If `height` is not specified, the height of the row is dictated by the contents of the row. The attribute has the type *LENGTH*.

skip

A previous section described how you can use the `skip` tag in stacking tags. You can also use the `skip` tag to define the distance between rows. To do this, the tag must be the only element in a row.

A row that only contains a `skip` tag is not included when checking the number of columns in an array.

Examples

The following array:

```
{
    "REFERENCE"     .Reference:width=5cm;
    "RECEIVER"      .Receiver:width=5cm;
}
```

results in the following printout with that contains two rows and two columns:

```
REFERENCE  Peter Smith
RECEIVER   Allan Thompson
```

It is superfluous to set the width of both fields because the width of the column is derived from the widest element. The example, therefore, shows how to set the width of the column, instead.

The elements in rows can also be stacking tags. For example:

```
{
    <stack baseline=bottom>
    "CUSTOMER"
    "REFERENCE"
    <end stack>
    .Reference:width=5cm;
    <skip 2mm>;
    "RECEIVER"      .Receiver:width=5cm;
}
```

which results in the following printout:

```
CUSTOMER
REFERENCE  Peter Smith

RECEIVER   Allan Thompson
```

As you can see, the printout contains two rows: the first element in the first row is a stack that has two elements, which therefore span across two lines.

Finally, arrays can occur in rows:

```
{
    {
    "CUSTOMER NO."    .ThePaymentCustomer:width=5cm;
    "REFERENCE"       .Reference:width=5cm;
    "RECEIVER".Receiver:width=5cm;
```

```
    }
    {
    "ORDER NO."      .OrderNumber:width=5cm:left;
    <skip 5pt>;
    "INVOICE DATE"   .InvoiceDate:width=5cm;
    }
    ;:top     -- end row with two arrays
}
```

which results in the following printout

CUSTOMER NO	203108	ORDER NO.	2060001
REFERENCE	Peter Smith	INVOICE DATE	06-04-04
RECEIVER	Allan Thompson		

This representation is often used in printout headers (for example, at the top of an invoice) where the information is represented in two columns, and information is grouped by inserting `<skip 5pt>` between the elements. You achieve two columns with information by inserting an array with one row (stretching from line 2 to 12). This row contains two array elements, each of which represents a column. Thus the first column is defined between lines 2 and 6, and the other column is defined between lines 7 and 11.

Rows in Stacking Tags

```
{
    <conditional EUSalesVar>
    "TAX NO." PayerTaxNumber;
    <end conditional>
    "CUSTOMER NO." .PaymentCustomer;
}
```

Horizontal Lines

Just as in stacking tags, you can specify horizontal lines in arrays. You can use these as sum or header lines, or in combination with vertical lines to make tables. The syntax for horizontal lines (`hline`) is as follows:

```
<hline attributes>
```

In arrays the tags have the following attributes:

- `columns` specifies how many columns the line is to stretch across. `columns` has the type *INTEGER* and is nameless. If you do not specify a `span` attribute, one of the following things happens:
  - If the `hline` tag appears as the only element in the row, the line will stretch across all columns.
  - If the `hline` tag appears with other elements in the row, the default value for `columns` is `1`.
- `multi` specifies how many lines you want. You might, for example, set `multi` to `2` if you want to make a double line under a sum. The attribute has the type INTEGER. If it is not specified, a single line is drawn.
- `left` and `right`. Horizontal space is placed between the columns to ensure a nice array layout. By using these attributes, you can specify whether the line is to underline this extra

space to the left or right of the columns across which the line is stretched. The attribute has the type *BOOLEAN*. The default value is `true`—that is, if they are not specified, the line will stretch across the space between the columns to the left as well as to the right.

Because all columns have extra space on the left and on the right, the specification of left+ or right+ means that half of the distance between the columns is underlined. The first column only has extra space on the left if a column separator has been specified before the first column (see “Column Separators”). Similarly, the last column only has extra space on the right if a column separator has been specified after the last column.

Example

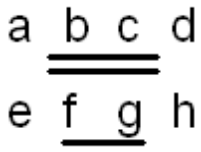
```
{
   <hline columns=3>;
   <conditional ItemSalesOut script="Item Sales">
   "TOTAL ITEM SALES"   .Currency .ItemSumCurrency;
   <end conditional>
}
```

Note that both rows contain three elements, the first due to the `span` attribute. You could also leave out the `column` attribute.

The following example illustrates the use of the other attributes:

```
{
   "a" "b" "c" "d";
   "" <hline columns=2 multi=2>     "" ;
   "e" "f" "g" "h";
   "" <hline columns=2 left- right-> "" ;
}
```

which results in the following printout (enlarged to illustrate the effect of `left-` and `right-`):



Note that the bottom line is shorter than the double line because of the use of `left-` and `right-`.

Rulers—Column Definitions

A *ruler* specifies how the columns of an array should behave—how wide they should be and how they should be separated. To begin, we will explain how a ruler is used to specify column widths.

Rulers correspond to tab stops in word processors. However, rulers have more options.

A column is viewed as a tag that only has a short form.

A ruler value is given as:

```
[columndefinition]
```

where `columndefinition` is a number of columns separated by `columnseparators` (defined in “Column Separators”). Furthermore, a `columndefinition` can start and end with a column separator. A *column* is written as:

```
[columnattributes]
```

A ruler value should contain as many columns as the array that it describes.

This means that a ruler value without column separators has the following format:

```
[[columnattributes][columnattributes] ... [columnattributes]]
```

Every occurrence of `[columnattributes]` defines the behavior of one column. The following column attributes can be provided:

- `stretch` indicates whether the column should stretch. If the attribute is not provided, the column will not stretch. If at least one column in an array is allowed to stretch, the whole array will stretch to fill the surrounding space. The extra space will be distributed evenly among the stretchable columns. The attribute has the type *BOOLEAN*.
- `width` specifies the minimum width of the column. The attribute has the type *LENGTH* and is nameless.

Example

Consider:

```
<island width=10cm>
    {:ruler=[[][stretch+]]
    "ORDER NO."      .OrderNumber;
    "INVOICE DATE"   .InvoiceDate;
    "VERSION NO."    .VersionNumber;
    }
<end island>
```

The ruler has two columns, just as the three rows in the array. The first column has no specifications for the column containing the fixed text. Thus the width of the column will only be determined by its contents, and it will therefore be made wide enough to contain the three fixed texts. The `stretch` attribute is specified for the other column. This means that the array will stretch across the space available in the island: 10 cm. Therefore, the second column will be 10 cm wide minus the width of the widest text line (probably “INVOICE DATE”) and minus any column spacing.

When you prepare tables like this you should consider carefully which fields should be assigned any additional space. Usually, fields and variables that contain text must stretch, because you cannot predict their width.

Naming Rulers

You often want the columns of two arrays to have the same width. This is done in MPL by letting the two arrays share the same ruler—the ruler is given a name, and the two arrays refer to the name of the ruler definition. Ruler definitions can occur in the beginning of the following stacking tags: `frontpage`, `paper`, `header`, `footer`, `conditional`, `repeating`, `island`, `stack`, and `span`. The scope of the ruler is explained in “Ruler Scope.”

To name a ruler, the following tag is used:

```
<ruler attributes>
```

The tag has two attributes:

- `name` specifies the name by which the ruler should be known. The attribute has the type *ID* and is mandatory.

- `value` specifies the ruler that is to be named. The attribute value must be a ruler. The attribute is nameless and mandatory.

The attribute can also have the type *ID*, in which case it must be the name of a ruler. This attribute type can be used to assign a new name to a ruler, but it is not particularly useful.

Example

Consider

```
<ruler common [[][]]>
{:common
   "ORDER NO.":fontsize=12:bold+
   .OrderNumber:width=5cm:left:fontsize=12:bold+;
}
PaymentAddress1
PaymentAddress2
PaymentAddress3
{:common
   "CUSTOMER NO." .PaymentCustomer:width=5cm;
   "INVOICE DATE" .InvoiceDate:width=5cm;
}
```

If you only look at the first five lines of this fragment, the ruler will seem superfluous (because no column attributes have been specified, this corresponds to not specifying a ruler). The ruler's purpose is exclusively to ensure that the two arrays are formatted with the same column definition. This ensures that `.PaymentCustomer` and `.InvoiceDate` will appear in the same column as the order number:

ORDER NO. 2342944
ARMaSoft Limited
The eSpace Centre
CB2 4DU Cambridge
CUSTOMER NO. 10024
INVOICE DATE 06-04-04

Subrulers

You might want two arrays to share certain (but not all) columns. For example, the sum amount on an invoice is to be placed in the same column as the amounts on the invoice lines, but the information on the sum lines does not have to be placed in the same column as the information on the invoice lines. For this purpose a subruler is used. With a subruler you can take a selection of columns from a ruler and group other columns into one column.

This might sound complicated but, for example, the definition:

```
<subruler new [[2][3:4]] parent=old>
```

will create a ruler with the name "new" consisting of two columns. The first column is the second column from a ruler named "old" and the second column in "new" consists of a grouping of the third and fourth column in the ruler "old."

Subruler definitions can appear in the beginning of the following stacked tags: `frontpage`, `paper`, `header`, `footer`, `conditional`, `repeating`, `island`, `stack`,and `span`.

The syntax for subruler is:

```
<subruler attributes>
```

The tag has three attributes:

- `name` specifies the name that you want to give to the subruler. The attribute has the type *ID* and is nameless and mandatory.
- `value` is the subruler itself. The attribute value must be a subruler. The attribute has the type *ID* and is nameless and mandatory.
- `parent` specifies the name of the ruler from which you want to define a subruler.

  The ruler must be recognized and can also be the name of a subruler. The attribute has the type *ID* and is nameless and mandatory.

A subruler value is very similar to a ruler value and is specified as:

```
[subcolumndefinition]
```

where the `subcolumndefinition` is a row of subcolumns divided by column separators. Furthermore a `subcolumndefinition` can be started and ended by a column separator. A subcolumn is written as:

```
[Range]
```

or

```
[INTEGER]
```

A range is specified as *INTEGER*:*INTEGER*. A range n_1:n_2 specifies that the column must correspond to the columns n_1 to n_2 (both included). If only *INTEGER n* is specified, this is actually an abbreviation of *n*:*n*.

As for columns, subcolumns can be viewed as tags that only have a short form: [shortattributes]. The only attribute is a range that has the type *RANGE*. This type is either *INTEGER* or *INTEGER*:*INTEGER*. The attribute interval is nameless and, therefore, you will usually only write a subcolumn as [*INTEGER*] or [*INTEGER*:*INTEGER*].

Subcolumns must refer to the parent ruler's columns from left to right, and they cannot overlap.

Iteration: If [n1:n2] [m1:m2] is a part of the subruler, n1≥n2 and m1≥m2.

Examples of legal subrulers:

```
[[1][3][5]]
[[1:2][4:5]]
[[1:3][4][5]]
```

Examples of illegal subrulers:

```
[[5][3][1]]  -- Columns not ordered
[[2:1][4:5]] -- Columns not ordered [[1:3][3:4][5]]  -- Two columns are
overlapping
```

Example

The following example is taken from a simplified layout to the layout "Print Invoice," but many details have been left out. We define the following rulers:

```
<ruler lines [[1cm][stretch+][15mm]]>
<subruler primarylines[[1:2][3]] parent=lines>
```

```
<subruler secondarylines [[2][3]] parent=lines>
```

The first line defines all of the columns on an invoice line. Consider the ruler “lines” as defining three tab stops. The first column is 1 cm wide and is used as the indent marker that is to be inserted before secondary lines (“Discounts” and “Extra Text” in this example). The second column is the text column. This should be as wide as possible and has therefore been made stretchable. The last column contains amounts and is set to 15 mm.

The second line defines primary lines. Here the text must start from the very left and the first column therefore consists of the first and second column from the ruler “lines.” The second column is the amount column, which should contain the price.

The third line defines secondary lines that are related to primary lines. This ruler is similar to the ruler “primarylines,” but in this case the text column is indented as it only consists of column 2 from the ruler “lines.”

The rulers can now be used as follows:

```
<repeating InvoiceLine script="InvoiceLineBlock">
    <conditional PricesOut>
    {:primarylines
           .ExternalItemText CorrectedPriceWithoutDiscount;
    }
    <end conditional>
    <conditional DiscountOut>
    {:secondarylines
           .DiscountText CorrectedDiscountCurrency;
    }
    <end conditional>
    <repeating InvoiceBOMLine script="BOMLineBlock">
    <conditional PricesOut>
    {:secondarylines
           InvoiceBOMLine.ExternalItemText
           CorrectedPriceWithoutDiscount:zerosuppression+;
    }
    <end conditional>
    <end repeating>
<end repeating>
```

which results in the following printout

Special Spring Offer	0,00
KP Keyboard	1.200,00
Mouse, Standard X-4	200,00
Ixo 19" Monitor	1.800,00
Zitech Pentium 166 MHz	3.800,00
Ixo 19" Monitor	1.800,00
May discount	-72,00

“Spanning Columns” contains a description of an alternative way of achieving this formatting. You can use whichever of the two methods you prefer the most.

Ruler Scope

The name of the ruler is recognized in all of the stacking tags in which it is defined. However, it will not be recognized in a tag where the left or right edge has been indented in relation to the stacking

tag in which the ruler is defined. Such indentation is inserted due to the use of the `indent` attribute in a parenthetical tag or the use of `leftmargin` or `rightmargin` in islands.

The following example illustrates the ruler scope:

```
-- my_ruler is not recognized here
<stack>
   <ruler my_ruler [[][stretch+]]>
   -- my_ruler is recognized here
   <island>
   -- my_ruler is recognized here
   <end island>
   <island leftmargin=1cm>
   -- my_ruler is not recognized here as the island's
   -- contents are indented
   <end island>
   <stack>
   -- my_ruler is recognized here
   <end stack>
   <stack indent=1cm>
   -- my_ruler is not recognized here as the stack
   -- is indented
   <end stack>
<end stack>
-- my_ruler is not recognized here
```

Column Separators

You can insert column separators between columns in rulers or subrulers. A column separator specifies what should be inserted between the columns.

A column separator is a tag; you can only use the `vline`, `text`, and `text2` tags as column separators. The `vline` tag is described later in this manual; it is used more often as a column separator than as an independent tag. The `text2` tag is described in "Alternative Text Tag."

The following fragment constitutes a legal ruler:

```
[[]<text "Hello">[]<vline>[]]
```

This ruler will write the text "Hello" in each row between the first and the second column and insert a vertical line between the second and the third column. You will often use the short form of these tags, and the ruler will look as follows:

```
[[]"Hello"[]|[]]
```

Note that the short form of `vline` is just a vertical line: `|`.

The most common characters in column separators are dividing characters ("-," "/," and so on). Furthermore, you can also insert multiple spaces using this tag to obtain extra space between columns. To make this easier, you can simply specify a width:

```
[[]2cm[]]
```

This ensures a column separation of 2 cm.

Thus you can specify the following column separators:

- Vertical lines are used to create tables. A vertical line is written as `<vline` *attributes*`>` or with the short form "|*shortattributes*."

  The only attribute to the `vline` tag is `justification` (see also "vline"). The attribute is rarely used when `vline` is used as the column separator.
- Strings that should be repeated in all columns are specified using the `text` or `text2` tags.
- The distance between columns. If you want to change the distance between all columns, you should change the length constant `InterColumnSpacing` instead (see "Length Constants" in "Advanced MPL").

Subrulers do not inherit column separators from the parent ruler. If, for example, you want to repeat vertical lines, you must specify these again in the subruler.

Example

If you want to create a simple table, it can be done as follows:

```
<ruler tableline[|[]|[]||]>
{:tableline
   <hline>;
   "Field":bold+  "Selection Criteria":bold+;
   <hline>;
   "Delivery Mode"DeliveryModeVar; "Delivery Terms"   DeliveryTermsVar;
   "Carrier"TransporterVar;
   "Consignment Type"   ConsignmentTypeVar;
   <hline>;
}
```

which results in the following printout:

Field	Selection Criteria
Delivery Mode	Mail
Delivery Terms	FOB
Carrier	
Consignment Type	

The following is an example of the use of `subruler`:

```
<ruler tableline[|[]|[2cm]"-"[2cm]|]>
<subruler headingline[|[1]|[2:3]|] parent=tableline>
{:headingline
   <hline>;
   "Field":bold+ "Selection Criteria":bold+:center;
}
{:tableline
   <hline>;
   "Invoice No."  FromInvoiceNumber:right ToInvoiceNumber:left;
   "Ordrenr."     FromOrderNumber:right ToOrderNumber:left;
   "Company No."  FromCompanyNumberVar:right ToCompanyNumberVar:left;
   "Bill to Customer No."     FromPaymentCustomer:right
   ToPaymentCustomer:left;
```

```
    <hline>;
}
```

This fragment also shows how you can use a text as a column separator. The result is the following printout:

Field	Selection Criteria
Invoice No.	200001 - 200007
Ordrenr.	0 - 0
Company No.	-
Bill to Customer No.	-

Spanning Columns

It is often useful to make an element in an array stretch across several columns, for example, if a heading should span across two columns. The `span` tag is a parenthetical tag that can contain only one element. This element can be a parenthetical tag, although not a row. You use this tag to specify the number of columns across which the contents should span. The syntax is as follows:

```
<span attributes>
    ...
<end span>
```

The contents of a `span` tag is one element (possibly preceded by definitions of rulers, defaults, and length constants). Often you will use the short form, that is, surrounding parentheses:

```
(
    ...
)shortattributes
```

The tag has the following attribute:

- `columns` is used to specify the number of columns across which the contents should span. The tag has the type *INTEGER* and is mandatory and nameless both in the short and the long forms.

Example

First we will consider an example based on the print layout for “Print Balance”:

```
{:[[1cm][stretch+][][15mm][15mm][15mm][15mm]]
    "" "" "" ("Period":center):2 ("Year to Date":center):2; ("Account"):2
    "Tax Code" "Debit":center "Credit":center "Debit":center
    "Credit":center;
    <repeating cursor=ACCOUNTSTRUCTURELINE script="B_BLOCK">
    <conditional variable=ShowAccount>
          AccountNoField .Description VATCodeField
          MovementDebitField:right
          MovementCreditField:right
          BalanceDebitField:right
          BalanceCreditField:right;
    <end conditional>
    <end repeating>
}
```

Here you should pay attention to the fact that the `span` tag is used twice on line 2. The heading “Period” spans across two columns: The debit and credit columns for ”Period.” Similarly, the heading “Year to Date” spans across a debit column and a credit column. On line 3, the heading “Account” spans across two columns (account number and account description). The result is as follows:

			Period		Year to Date	
Account		Tax Code	Debit	Credit	Debit	Credit
	REVENUE :					
	External Revenues :					
1010	Licenses		0,00		0,00	
1020	Update Contracts		0,00		0,00	
1030	Services		0,00			189.740,31
1040	3rd. Party Products & Services		0,00		0,00	
1510	License fee Parent Company		0,00			7.000,00
1520	CSS fee Parent Company		0,00		0,00	
2010	Licenses & Update Contr.		0,00		0,00	
2020	Oracle Licenses		0,00		0,00	
2030	Services		0,00		0,00	
2040	3rd. Party Products & Services		0,00		16.495,38	

We will now return to the example from “Subrulers.” Here we define the ruler:

```
<ruler invoicelines [[1cm][stretch+][15mm]]>
```

The rulers can now be used as follows:

```
{:invoicelines
   <repeating Invoiceline script="Invoicelineblock">
   <conditional PricesOut script="PricesOut">
          (.ExternalItemText):2
          CorrectedPriceWithoutDiscount;
   <end conditional>
   <conditional DiscountOut script="DiscountOut">
          "" .Discounttext CorrectedDiscountCurrency;
   <end conditional>
   <repeating InvoiceBOMLine script="BOMLineBlock">
          <conditional PricesOut>
             {:secondarylines
                InvoiceBOMLine.ExternalItemText
                CorrectedPriceWithoutDiscount:zerosuppression+;
          <end conditional>
   <end repeating>
   <end repeating>
}
```

The resulting printout matches the printout on the figure above, but without the use of subrulers.

vline

We are already familiar with the `vline` tag and how it is used as a column separator. Vertical lines can also be used as an element in rows:

```
<vline attributes>
```

The short form of the `vline` tag is simply:

```
| shortattributes
```

The tag has a single attribute:

- `justification` is nameless in both the long and the short forms and has the type *ID*. It can take the values `left`, `center`, and `right`. If justification is not specified, the default value is `left`.

It is not an easy task to use the `vline` tag to create tables; it is recommended to use the tag as column separator instead.

Example

The program fragment:

```
{
   <hline 5 left- right->;
   |:left "1" |:center "2" |:right;
   <hline 5 left- right->;
   "" |:left "3" |:right "";
   "" <hline 3 left- right-> "";
}
```

results in the following printout:

Notice how `left-` and `right-` have been used to prevent the horizontal lines from stretching below the column separators that are inserted before and after the vertical lines.

Printout Example: Time Sheets

In this section we will create a layout for the printout “Print Time Sheet.” The section describes the practical procedures involved in the creation of a new layout.

Getting Started

Export the structure layout for the printout “Print Time Sheet” from the window Print Layout in the Maconomy client. The result is a valid MPL definition. The structure is shown below:

```
<mpl 1>
<layout title="Standard"
          print="Print_TimeSheets"
          originallayout="Standard">
<page "A4">
<frontpage>
<end frontpage>
<paper>
   <header onfirstpage+ height=61pt>
   <end header>
   <stack>
          <repeating TimeSheetHeader
                 script="S_TimeSheetHeader">
          <repeating TimeSheetLine
                 script="S_TimeSheetLine">
             <header>
             <end header>
          <end repeating>
          <stack>
          <end stack>
   <end repeating>
   <end stack>
<end paper>
```

The structure layout defines the structure of the original layout, that is, the allowed structure of new layouts. For more information, see “Print Structure.” The structure layout is therefore a good starting point for the design of your own layouts.

To see which fields and variables you are able to access for each printout in Maconomy, use the manual “Variables and Cursors in Printouts,” which is updated for each application version of Maconomy. If you want to see *where* and *how* the fields and variables are used, it is a good idea to export the original layout.

Create the New Layout

The Header

We want to create a layout that can be printed on A4 paper in landscape orientation. We shall name the layout “Landscape, standard.” The header of the MPL definition will look as follows:

```
-- Layout:
```

```
--     "Landscape, standard" to be used by consultants
--
-- Design:
-- Peter Smith (May 5 2004)
--
-- Based on structure layout version:
-- Application version: Maconomy W 8.0
-- Application date (yyyy/mm/dd): 2004/02/24
<mpl 2>
<layout title="Landscape, standard"
   print="Print_TimeSheets"
   originallayout="Standard">
<page "A4" landscape>
```

As you can see, a comment has been included in the MPL header, making it easy to pinpoint:

- The layout designer
- The purpose of the layout
- The Maconomy version for which the layout has been designed

The information that you want to include in the comments is, of course, up to you.

The Front Page

The front page of the print layout “Print Time Sheet” has two purposes:

1. Inform about who is printing the time sheet
2. Show the selection criteria specified for the printout

The information about the printout is bundled in an island that we have named “Time Sheets.” This island contains an array with two columns where the first column contains the user name and company name, and the other contains the time and date. Both columns have been stretched so that they take up the entire space and become equally sized (ensuring that we have room for the contents of the fields). Furthermore, the time and date have been right-adjusted.

The selection criteria information has also been placed in an island that has been made smaller than the paper. The array in this island has three columns: A Text column, a From column, and a To column. The last two columns are separated by hyphens (specified as ‘-’ instead of “-” in the ruler; the difference is explained in “Alternative Text Tag” in “Advanced MPL”). The text column cannot stretch, and the two other columns share the extra space.

The first row is aligned to this ruler, but the next row is not, because only one superior is specified and not a range. This is done by letting `SeniorEmployeeVar` stretch across two columns.

The fifth row is different from the rest, as the “From” and “To” information is a date consisting of week number, slash (“/”), and year. For each of these pieces of information we specify an embedded array consisting of one row with the date.

```
<frontpage>
   -- Header
   <island "Time Sheets" leftmargin=5mm rightmargin=5mm
            bottommargin=2mm>
   {:[[stretch+][stretch+]]
         CompanyName Time:right;
         UserName  TodaysDate:right;
```

```
    }
    <end island>
    <skip 1cm>
    -- Selection Criteria
    <island width=14cm
    leftmargin=2cm rightmargin=2cm
    topmargin=2mm bottommargin=2mm>
    {:[[][stretch+]'-'[stretch+]]
           "Employee No." FirstEmployeeNumber
           LastEmployeeNumber;
           "Superior"(SeniorEmployeeVar):2;
           "Secretary"      (SecretaryEmployeeVar):2;
           "Submitted"      (SubmittedVar):2;
           "Week No."{FirstWeek '/' FirstYear;}
                  {LastWeek '/' LastYear;};
           "Layout Name" (LayoutName):2;
    }
    <end island>
<end frontpage>
```

In the preceding print definition we have used span and embedded arrays in the rows that did not fit into the overall ruler. This is easier to understand if there are only a few rows—if there had been more rows, the easiest and most elegant solution would be to use named rulers and subrulers:

```
<frontpage>
    -- Header
    ...
    -- Selection Criteria
    <island width=14cm
    leftmargin=2cm rightmargin=2cm
    topmargin=2mm bottommargin=2mm>
    <ruler DateRangeLine
           [[][stretch+]'/'[stretch+]'-'[stretch+]'/'[stretch+]]>
    <subruler RangeLine [[1][2:3]'-'[4:5]] parent=DateRangeLine>
    <subruler SimpleLine [[1][2:3]] parent=RangeLine>
    {:RangeLine
           "Employee No."   FirstEmployeeNumber LastEmployeeNumber;
    }
    {:SimpleLine
           "Superior"SeniorEmployeeVar;
           "Secretary" SecretaryEmployeeVar;
           "Submitted" SubmittedVar;
    }
    {:DateRangeLine
           "Week No."FirstWeek FirstYear LastWeek LastYear;
    }
    {:SimpleLine
           "Layout Name"    LayoutName;
    }
```

```
   <end island>
<end frontpage>
```

If you want to include more date range lines, this definition also ensures the alignment of the week and year fields.

The Paper Content

The Layout for Each Time Sheet Line

The inner repeating block from the structure layout looks as follows:

```
<repeating cursor=TIMESHEETLINE
   script="S_TIMESHEETLINE">
<end repeating>
```

The contents of this block will be printed for each line on a time sheet. The desired information here is job number, job name, activity number, entry description, number of hours for each of the 7 days of the week, and an hours total. This information should be presented in one row so that any extra space is distributed to the two text fields (job name and entry description). As job numbers can be quite long, 1.5 cm has been set aside for this column.

```
<repeating cursor=TIMESHEETLINE script="S_TIMESHEETLINE">
   {:[[1.5cm][stretch+][][stretch+][][][][][][][][]]
   .JobNumber JobName .ActivityNumber .TextofEntry
   .NumberOfDay1    .NumberOfDay2 .NumberOfDay3
   .NumberOfDay4    .NumberOfDay5 .NumberOfDay6
   .NumberOfDay7    TotalLine;
   }
<end repeating>
```

The Layout for Each Time Sheet

Information about the time sheet, the column headings, and the totals should be printed on each time sheet.

The headings are placed in the same columns as the fields. The headings and the fields should therefore share a ruler. This is achieved by letting the array “flow” out of the inner repeating tag. By inserting a line after the headings, we get the following representation:

```
<repeating cursor=TIMESHEETHEADER script="S_TIMESHEETHEADER">
   {:[[1.5cm][stretch+][][stretch+][][][][][][][][]]
   "Job No." "Job Name"    "Activity No."      "Description"
   "Mon"     "Tue"  "Wed"  "Thu"  "Fri"  "Sat"  "Sun"  "Total";
   <hline 12>;
   <repeating cursor=TIMESHEETLINE script="S_TIMESHEETLINE">
          ...
   <end repeating>
   }
<end repeating>
```

The information about the time sheet should include employee number and name, period, whether the time sheet has been submitted, and if so, when:

```
{:[[][][stretch+]]
```

```
    "Employee"      .EmployeeNumber .EmployeeName;
    "Period" .PeriodStart:width=1.5cm.PeriodEnd:width=1.5cm;
    "Submitted"     .Submitted .DateSubmitted;
}
```

Finally, the totals are made part of the array that surrounds the time sheet lines. Before the totals we insert a line, and after the totals we insert a little extra space to separate the information from the information for the next time sheet, if any:

```
<hline 12>; "" "" "" ""
SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7 GrandTotal;
<skip 5mm>;
```

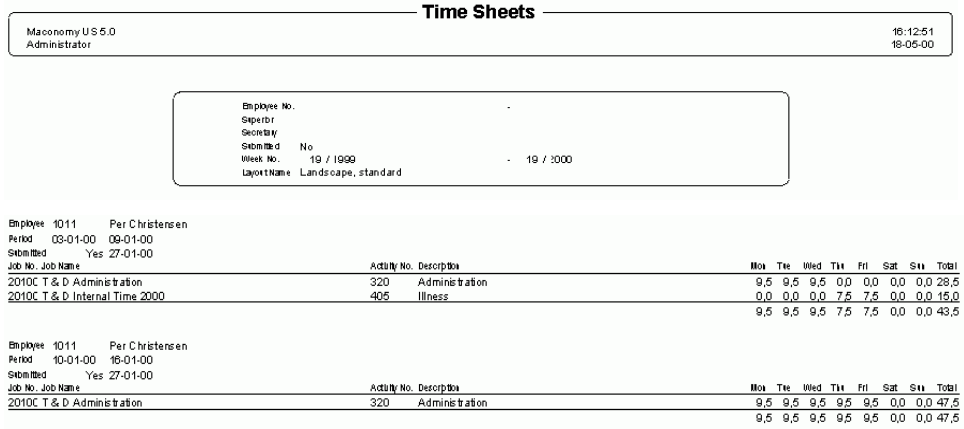
The Entire Layout

If you put the fragments described above together, you get the following layout:

```
-- Layout:
--     "Landscape, standard" to be used by consultants
--
-- Design:
-- Peter Smith (May 5 2004)
--
-- Based on structure layout version:
-- Application version: Maconomy W 8.0
-- Application date (yyyy/mm/dd): 2004/02/24
<mpl 2>
<layout title="Landscape, standard"
    print="Print_TimeSheets"
    originallayout="Standard">
<page "A4" landscape>
<frontpage>
    -- Header
    <island "Time Sheets" leftmargin=5mm rightmargin=5mm
              bottommargin=2mm>
    {:[[stretch+][stretch+]]
           CompanyName Time:right;
    }
    leftmargin=2cm rightmargin=2cm
    topmargin=2mm bottommargin=2mm>
    {:[[][stretch+]'-'[stretch+]]
           UserName  TodaysDate:right;
    <end island>
    <skip 1cm>
    -- Selection Criteria
    <island width=14cm
    "Employee No." FirstEmployeeNumber
    LastEmployeeNumber;
    "Superior"(SeniorEmployeeVar):2;
    "Secretary"      (SecretaryEmployeeVar):2;
```

```
    "Submitted"      (SubmittedVar):2;
    "Week No."{FirstWeek '/' FirstYear;}
    {LastWeek '/' LastYear;};
    "Layout Name" (LayoutName):2;
    }
    <end island>
<end frontpage>
<paper>
    <repeating cursor=TIMESHEETHEADER
              script="S_TIMESHEETHEADER">
    {:[][][][stretch+]]
           "Employee".EmployeeNumber     .EmployeeName;
           "Period"  .PeriodStart:width=1.5cm   .PeriodEnd:width=1.5cm;
           "Submitted"      .Submitted    .DateSubmitted;
    }
    {:[[][stretch+][][stretch+][][][][][][][][]]
           "Job No." "Job Name"    "Activity No."      "Description"     "Mon"
              "Tue"  "Wed"  "Thu"  "Fri"  "Sat"  "Sun"  "Total";
           <hline 12>;
           <repeating cursor=TIMESHEETLINE
                 script="S_TIMESHEETLINE">
              .JobNumber JobName .ActivityNumber .TextOfEntry
              .NumberOfDay1 .NumberOfDay2 .NumberOfDay3 .NumberOfDay4
              .NumberOfDay5 .NumberOfDay6 .NumberOfDay7 TotalLine;
           <end repeating>
           <hline 12>;
           "" "" "" ""
           SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7
           GrandTotal;
           <skip 5mm>;
    }
    <end repeating>
<end paper>
```

Below both the front page for the printout and the following page are shown.



Time Sheets

Maconomy US 5.0 — 16:12:51
Administrator — 18-05-00

Employee No. -
Superbr
Secretary
Submitted No
Week No. 19 / 1999 - 19 / 2000
Layout Name Landscape, standard

Employee 1011 Per Christensen
Period 03-01-00 09-01-00
Submitted Yes 27-01-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
2010C	T & D Administration	320	Administration	9,5	9,5	9,5	0,0	0,0	0,0	0,0	28,5
2010C	T & D Internal Time 2000	405	Illness	0,0	0,0	0,0	7,5	7,5	0,0	0,0	15,0
				9,5	9,5	9,5	7,5	7,5	0,0	0,0	43,5

Employee 1011 Per Christensen
Period 10-01-00 16-01-00
Submitted Yes 27-01-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
2010C	T & D Administration	320	Administration	9,5	9,5	9,5	9,5	9,5	0,0	0,0	47,5
				9,5	9,5	9,5	9,5	9,5	0,0	0,0	47,5

Basic Tags Continued

Having analyzed the Time Sheets example, we can continue our tour through the basic MPL tags.

Headers and Footers

Headers and footers are used to specify print elements that must be printed at the top or bottom part of each page. They are often used to specify print information (page number, date, the name of the printout), information about the contents (for example, which employee's time sheets are being printed on a given page), and column headers for repetitions.

In MPL, you can specify several types of headers and footers:

- With page headers, you can specify text elements that you want printed in the top part of each page.
- As with page headers, you can use page footers to specify text elements that you want printed in the bottom part of each page.
- You can specify block headers in repetitions, conditions, and stacking tags. These headers contain text elements that you want printed in the top part of each page as long as the block is printed.
- As with block headers, you can use block footers to specify text elements that you want printed in the bottom part of each page as long as the block is printed.

If you specify both a page and a block header/footer, both headers/footers will be printed. The usual rules of scope apply, meaning that if a given print element results in a page break, the block header/footer that belongs to the closest surrounding block (and only that header/ footer) will be printed.

Page Headers and Footers

You can specify page headers and footers anywhere in the page definition, that is, within the parenthetical tag `<paper> ... <end paper>` and not within any other parenthetical tags. You should always specify these headers and footers after any possible definitions (ruler statements and so on) on the page; they can share rulers with the contents of the page.

Headers and footers function as stacking tags; however, they cannot contain other headers or footers, `newpage` tags, repetitions, conditions, or stacks with scripts.

The contents of a page header will be printed in the top part of each page. Using an attribute, you can control whether a header should be printed on the first page. Similarly, the contents of a page footer will be printed in the bottom part of each page, and you can use an attribute to control whether a footer should be printed on the last page.

Headers and footers are often used to print out information about the contents of the printout, the date, and possibly page numbers (available as a variable in all prints).

Block Headers and Footers

You can specify block headers and footers anywhere in repetitions, conditions, and stacks. Headers/footers will be printed in the top/bottom part of the page if a page break is triggered while printing the contents of the repeating/conditional tag or stack.

If you have specified a header both on the page and in a block, both headers will be printed. The page header will be printed first, followed by the block header. Similarly, a block footer will be printed over the page footer.

You can specify headers/footers in several nested blocks. If an element specified inside such nested blocks triggers a page break, the header/footer in the closest surrounding block with a header/footer is printed.

header

Headers are specified using the parenthetical tag:

```
<header attributes>
    ...
<end header>
```

The `header` tag has the following attributes:

- `onfirstpage` is used only in page headers and specifies whether the header should be printed on the first page. The attribute has the type *BOOLEAN* and the default value is `false`.
- `atstart` is used in repetitions/conditions and stacks and specifies whether the header should be printed immediately before the printing of the actual block is initiated or only at page breaks. The attribute has the type *BOOLEAN* and the default value is `false`.
- `script` is used to specify the name of a script (program) that is executed each time that the header is printed. The attribute has the type *STRING*. The use of scripts must correspond to their use in the original layout. For more information, see “Print Structure.”
- `height` specifies the height of the header and is therefore used to determine where the text after the header should begin. If the contents are higher than the specified height, an error message is displayed. The attribute has the type *LENGTH*.

In addition to the height of the header, the length constant `HeaderSkip` also determines the distance between the header and the header text.

footer

Footers are specified using the parenthetical tag:

```
<footer attributes>
    ...
<end footer>
```

The footer tag has the following attributes:

- `onlastpage` is used only in page footers and specifies whether the footer should be printed on the last page. The attribute has the type *BOOLEAN*.
- `atend` is used in repetitions/conditions and stacks, and specifies whether the footer should be printed immediately after the block has been printed or only at page breaks. The attribute has the type *BOOLEAN*.
- `script` specifies the name of a script (program) that is executed each time the footer is printed. The attribute has the type *STRING*. The use of scripts must correspond to their use in the original layout. For more information, see “Print Structure.”
- `height` specifies the height of the footer and is therefore used to determine how far down the page the main text extends to. If the contents are higher than the specified height, an error message is displayed. The attribute has the type *LENGTH*.

This means that text is only printed until the bottom margin of the page plus the height of the footers plus the length constant `FooterSkip`.

- `pagebottom` is used in repetitions/conditions and stacks, and specifies whether the footer should be printed at the bottom of the page or follow the block. The attribute has type *BOOLEAN* and its default value is `true`.

Example

We will now attempt to use headers and footers in the time sheet layout described in the previous section.

Key information about the printout should be included in the page header:

```
<header onfirstpage+>
   <island "Time Sheets"
            leftmargin=5mm rightmargin=5mm bottommargin=2mm>
   {:[[stretch+][stretch+]] CompanyName TodaysDate:right;
   }
   <end island>
<end header>
```

You can then, for example, specify the page number in the page footer:

```
<footer onlastpage+>
   {:[[stretch+][][][][stretch+]]
   "" "-" PageNumber:center "-" "";
   }
<end footer>
```

The first and last columns are empty but will be stretched. This ensures that the three middle columns are centered.

As specified in the print definition in the last section, a line with column headings was printed before each list of time sheet lines. If such a list triggers a page break, it would be convenient if the line were repeated on each new page. This can be achieved by replacing the heading line with a header in the inner repeating block. By setting the attribute `atstart` to `true`, the heading is also printed before the first time sheet line:

```
<header atstart+>
   "Job No. "     "Job Name"  "Activity No. "   "Description"
   "Mon"    "Tue" "Wed" "Thu" "Fri" "Sat" "Sun" "Total";
   <hline 12>;
<end header>
```

If a page break is triggered in the middle of a time sheet, we want the printout to show that the time sheet is continued on the next page.

In this printout, it is not possible to display subtotals. The variables SumDay1, SumDay2, and so on, are updated by the script S_TIMESHEETLINE. This script is executed for each record before the contents of the repeating block are printed. This means that .NumberOfDay1 is added to SumDay1 on each line before the contents are printed, and therefore, before the printout breaks to a new page. A footer based on SumDay1 will, therefore, display subtotals that include the first line from the following page.

This is achieved by inserting a footer in the inner repeating block:

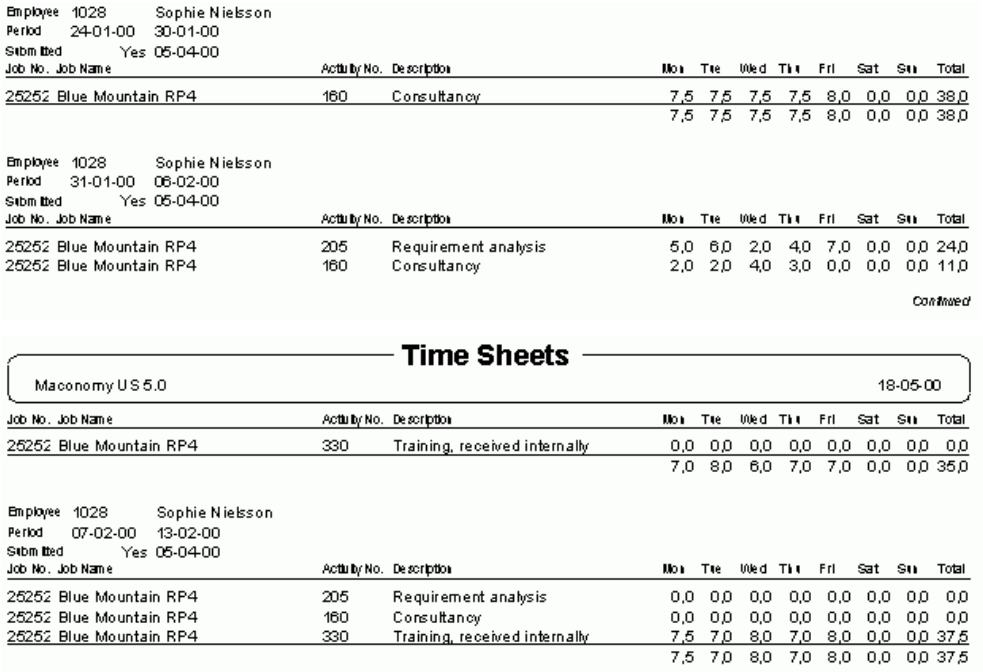
```
<footer>
   ("Continued":right:italic+):12;
<end footer>
```

We do not use the `atend` attribute here because the footer should not be printed when the repeating block has terminated. With these changes, the `paper` part of the time sheet layout will look as follows:

```
<paper>
   <header onfirstpage+>
   <island "Time Sheets"
            leftmargin=5mm
            rightmargin=5mm
            bottommargin=2mm>
          {:[[stretch+][stretch+]]
            CompanyName TodaysDate:right;
          }
   <end island>
   <end header>
   <footer onlastpage+>
   {:[[stretch+][][][][stretch+]]
          "" "-" PageNumber "-" "";
   }
   <end footer>
   <repeating cursor=TIMESHEETHEADER
            script="S_TIMESHEETHEADER">
   {:[[][][stretch+]]
          "Employee".EmployeeNumber .EmployeeName;
          "Period"  .PeriodStart:width=1.5cm   .PeriodEnd:width=1.5cm;
          "Submitted"      .Submitted .DateSubmitted;
   }
   {:[[][stretch+][stretch+][][][][][][][][]]
          <repeating cursor=TIMESHEETLINE
                script="S_TIMESHEETLINE">
             <header atstart+>
                "Job No." "Job Name"   "Activity No."      "Description" "Mon"
                   "Tue"  "Wed"  "Thu"  "Fri"  "Sat"  "Sun"  "Total";
                <hline 12>;
             <end header>
             .JobNumber JobName.ActivityNumber .TextOfEntry
             .NumberOfDay1 .NumberOfDay2 .NumberOfDay3 .NumberOfDay4
             .NumberOfDay5 .NumberOfDay6 .NumberOfDay7 TotalLine;
             <footer>
                ("Continued":right:italic+):12;
             <end footer>
          <end repeating>
          <hline 12>; "" "" "" ""
          SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7
```

```
            Grandtotal;
            <skip 5mm>;
    }
    <end repeating>
<end paper>
```

The next page displays the result of including headers and footers in the definition. The printout displays the bottom of a page and the top of the following page.



Employee 1028 Sophie Nielsson
Period 24-01-00 30-01-00
Submitted Yes 05-04-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
25252	Blue Mountain RP4	160	Consultancy	7,5	7,5	7,5	7,5	8,0	0,0	0,0	38,0
				7,5	7,5	7,5	7,5	8,0	0,0	0,0	38,0

Employee 1028 Sophie Nielsson
Period 31-01-00 06-02-00
Submitted Yes 05-04-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
25252	Blue Mountain RP4	205	Requirement analysis	5,0	6,0	2,0	4,0	7,0	0,0	0,0	24,0
25252	Blue Mountain RP4	160	Consultancy	2,0	2,0	4,0	3,0	0,0	0,0	0,0	11,0

Continued

Time Sheets

Maconomy US 5.0 18-05-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
25252	Blue Mountain RP4	330	Training, received internally	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
				7,0	8,0	6,0	7,0	7,0	0,0	0,0	35,0

Employee 1028 Sophie Nielsson
Period 07-02-00 13-02-00
Submitted Yes 05-04-00

Job No.	Job Name	Activity No.	Description	Mon	Tue	Wed	Thu	Fri	Sat	Sun	Total
25252	Blue Mountain RP4	205	Requirement analysis	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
25252	Blue Mountain RP4	160	Consultancy	0,0	0,0	0,0	0,0	0,0	0,0	0,0	0,0
25252	Blue Mountain RP4	330	Training, received internally	7,5	7,0	8,0	7,0	8,0	0,0	0,0	37,5
				7,5	7,0	8,0	7,0	8,0	0,0	0,0	37,5

Canvas

This section describes the use of a canvas in MPL. A canvas is used for placing elements at exact positions on the paper.

Exact Positioning of Elements

One of the main advantages of using MPL is that the you do not need to worry about the exact position of each element, but can concentrate on the logic of the print definition. In some circumstances it is, however, necessary to position print elements at exact places on the paper. This can be useful if you want a printout to match a preprinted form, and even necessary if you want texts or fields to overlap. You use the parenthetical tag `canvas` to achieve this. You can also use this tag can if you want to draw lines that are not horizontal or vertical.

canvas

While stacking tags can be illustrated as a sheet of ruled paper where the text is positioned horizontally underneath each other, a canvas corresponds to the canvas of a painter, where the element can be positioned freely. The syntax for canvases is:

```
<canvas attributes>
    ...
<end canvas>
```

Every element in a canvas must be assigned a `pos` attribute that specifies where the element is to be positioned. You specify the position of the element's top-right corner in relation to the top-left corner of the canvas. The canvas tag has the following attributes:

- `height` specifies the height of the canvas. The attribute has the type *LENGTH*. If the tag is specified, all elements in the canvas must observe the specified height. If not specified, the height of the canvas is calculated based on the contents.
- `width` specifies the width of the canvas. The attribute has the type *LENGTH*. If the tag is specified, all elements in the canvas must observe the specified width. If not specified, the width of the canvas is calculated based on the contents.
- `indent` specifies the indentation of the canvas. The attribute has the type *LENGTH*. The attribute cannot be used if the canvas is part of another canvas.
- `pos` specifies the position of a canvas in another canvas. The attribute has the type *POS*. When the attribute is used (that is, when the canvas is part of another canvas) it is nameless and mandatory.

If you place a canvas in a row, the baseline for the last element in the canvas will be the baseline for the canvas itself. Because the order of elements is not subject to any rules, you can specify the elements from which the baseline should be inherited.

Elements in a canvas can overlap each other.

Example

The following example illustrates the use of canvas:

```
{
    <canvas>
    "hello":(0mm,0mm)
    "world":(8mm,8mm)
    <end canvas>
    "!!";
}
```

which results in the following printout:

```
hell
      world !!
```

Note that baseline for the canvas is inherited from the last element (that is, “world”). “!!” is therefore positioned on the same line as “world.”

Lines

You can use a canvas to draw lines:

```
<line attributes>
```

The tag has the following attributes, both of which are mandatory. Both attributes have the type *POS*:

- `start` specifies the starting point of the line.
- `end` specifies the ending point of the line.

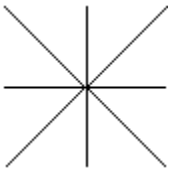
Lines can go in any direction and have any length (as long as they are within the canvas), and lines are allowed to cross each other or other elements.

Example

The following example illustrates the use of `line`:

```
<canvas width=2cm height=2cm>
   <line start=(0cm,0cm) end=(2cm,2cm)>
   <line start=(1cm,0cm) end=(1cm,2cm)>
   <line start=(2cm,0cm) end=(0cm,2cm)>
   <line start=(2cm,1cm) end=(0cm,1cm)>
<end canvas>
```

which results in the following printout



Custom Calculations

MPL 4 introduces expressions as a unified means of referring to the predefined environment values (that is, fields and variables) as well as calculating new ones. The newly calculated values can then be bound to named symbols—*vals* (that is, constant values) or *vars* (that is, mutable variables). *Vals* represent constants, and their values cannot be changed throughout the execution of the print. On the other hand, *vars* represent mutable variables and can be assigned new values using the `<assign>` tag.

To define a new *val*, you can use the `<val>` tag, for example:

```
<val currencySymbol {if currencyName = "USD" then "$"
                     else if currencyName = "EUR" then "€"
                     else if currencyName = "GBP" then "£"
                     else currencyName }>
```

the `currencySymbol` val represents a new value that can be used as any other predefined environment variable in the scope in which it is defined. So, for instance, you could use it to print out an amount like this:

```
 "Billing price" billingPriceVar currencySymbol;
```

Vars are defined in very similar ways to vals, but in contrast to them, vars can be assigned new values by means of the `<assign>` tag. Let us consider an example where we want to calculate an average of hours per time sheet line. In MPL 4, you can code it in the following way (provided that the TimeSheetLine cursor contains a field called `.HoursRegistered`):

```
<var HoursSum {0}>
<var LineCount {0}>
<repeating TimeSheetLine>
   <assign HoursSum {HoursSum + .HoursRegistered}>
   <assign LineCount {LineCount + 1}>
   ...
<end repeating>
<val AverageHoursPerLine {HourSum / LineCount}>
```

At the end, we create a *val* that holds the resulting average of hours per line.

Both *vals* and *vars* can be referenced only in the scope in which they are defined or in any of its nested scopes.

Value Definitions

The `val` tag:

```
<val attributes>
```

is used to define a new constant value that can be used in expressions. Once this is defined, the value cannot be changed—it is immutable.

The `val` tag takes three attributes:

- `name` specifies the name of the new value. The attribute has the type *ID* and is mandatory and nameless.
- `value` is an expression that specifyies the value that the new `val` should have. The attribute has the type *EXPRESSION* and is mandatory and nameless.
- `type` optionally specifies the type of the new value. The MPL compiler will check that the type of the expression supplied by the `value` attribute matches the specified type. If this attribute is not specified, the value's type is inferred by the MPL compiler. The attribute is optional and has the type ID, but the passed ID must match a valid type name.

Variable Definitions

The `var` tag:

```
<var attributes>
```

is used to define a new mutable variable that can later be modified using the `assign` tag. A variable that is defined using this tag can be used in expressions in the same way as existing variables.

The `var` tag takes three attributes:

- `name` specifies the name of the new variable. The attribute has the type *ID* and is mandatory and nameless.
- `value` is an expression that specifyies the initial value of the new variable. The attribute has the type *EXPRESSION* and is mandatory and nameless.
- `type` optionally specifies the type of the new variable. The MPL compiler will check that the type of the expression supplied by the `value` attribute matches the specified type. If this attribute is not specified the variable's type is inferred by the MPL compiler. The attribute is optional and has the type ID, but the passed ID must match a valid type name.

Variable Assignments

The `assign` tag:

```
<assign attributes>
```

is used change the value of a variable defined using the `var` tag.

The `assign` tag takes two attributes:

- `var` specifies the name of the variable to modify. The attribute has the type *ID* and is mandatory and nameless.
- `value` is an expression that specifies the new value of the variable. The attribute has the type *EXPRESSION* and is mandatory and nameless.

Database Queries

Starting with MPL version 4, you can declare your own database queries.

Up until MPL version 3 you were limited to using only the predefined cursors found in the standard Maconomy prints. However, this is not always sufficient for the task at hand. Sometimes the predefined cursors do not provide the information that you need, or they cannot be used freely due to the structure requirements on customized layouts (see “Print Structure”). In the “Print Employee” standard layout, for example, you will find this repeating block.

```
<repeating Employee script="L_Employee">
    {:ruler12
    .EmployeeNumber   .Name1   .Telephone;
    }
<end repeating>
```

This repeating block is iterating over a predefined cursor named “Employee,” but in this instance that cursor is used with a script and due to the structure requirements the cursor cannot be reused in another location in the customized layout.

To enable you to get to the information that you need in your prints, MPL version 4 introduces custom database queries. These are based on the Maconomy query language (MQL) that queries the Maconomy universes.

The description of MQL database queries and their use in MPL is described below using examples. The examples contain very little formatting, if none at all. This means that if you try to run the examples, the output will not be very beautiful. However, the examples are easier to read because we focus only on the data part in this section.

After describing queries by example we will give a more formal definition of the new tags: `<query>`, `<cursor>`, and `<parameter>`.

Queries, Cursors, and Repeating Blocks

Before looking into defining queries and cursors and using them in an MPL layout, let us first get an overview of the concepts involved.

You use the `<query>` tag to define a new database query. Custom queries in MPL take parameters, which, for example, can be used in the where clause. The query itself is written using MQL. Defining a query does not execute anything against the Maconomy database. A simple query could look like this:

```
<query EmployeeQuery>
   mselect EmployeeNumber, Name1, CostPrice
   from Employee
<end query>
```

A query must be instantiated, yielding a cursor, which can then be used in a repeating block. If a query is defined to take parameters, actual values for the parameters must be supplied when the query is instantiated. Because a query can be instantiated multiple times with different parameter values, queries in MPL are reusable. In our simple example, however, there are no parameters. To instantiate the query and get a cursor we would write:

```
<cursor name=EmployeeCursor query=EmployeeQuery>
<end cursor>
```

We now have a cursor. Still, nothing is executed against the database. To do that we need to use a `<repeating>` block:

```
<repeating EmployeeCursor>
   <header atstart+>
   {:ExampleRuler
         "EMPLOYEE NUMBER."  "EMPLOYEE NAME"  "COST PRICE";
   }
   <end header>
   {:ExampleRuler
   .EmployeeNumber  .Name1  .CostPrice;
   }
<end repeating>
```

Now, our query is executed and the three fields *EmployeeNumber*, *Name1*, and *Costprice* are printed.

You can use a cursor in repeating blocks multiple times. It will, however, return the same data every time (provided that the database has not changed). More interesting is to create another cursor that binds the query parameters to different values.

Queries with Parameters

Let us have a closer look at defining queries. The query we looked at before was without parameters, so let us define a query that takes parameters. If we, for example, want to print out all employees who have a particular superior employee, we could define this query:

```
<query EmployeeWithSuperior>
   mselect name1, costprice from employee
   where SuperiorEmployee = superiorParam
   using parameters superiorParam type string
<end query>
```

This query takes one parameter, the employee number of the superior whom we want to use for restricting the query.

To supply the actual values for the parameters of our query, we use the `<parameter>` tag inside our cursor definition:

```
<cursor name=EmployeeCursor query=EmployeeWithSuperior>
   <parameter superiorParam {"1010"}>
<end cursor>
```

When using the cursor *EmployeeCursor* in a repeating block, we will list all subordinate employees of the employee with employee number *1010*.

Instantiating a Query inside a Repeating Block

Also, as previously mentioned, you can instantiate a query with parameters multiple times. This is, for example, useful when binding a query inside another repeating block.

Consider the two query definitions from before:

```
<query EmployeeQuery>
   mselect EmployeeNumber, Name1, CostPrice
   from Employee
<end query>
```

```
<query EmployeeWithSuperior>
   mselect Name1, Costprice
   from Employee
   where SuperiorEmployee = SuperiorParam
   using parameters SuperiorParam type string
<end query>
```

We want to use these two queries to list all employees, and for each employee we want to list the subordinates of that employee. To do that we use two embedded repeating blocks and bind one of our cursors inside the outer repeating block like this:

```
<cursor name=EmployeeCursor query=EmployeeQuery>
<end cursor>
<repeating EmployeeCursor>
   -- Instantiate the current employee
   -- as the superior employee.
   <cursor name=SubordinateCursor
            query=EmployeeWithSuperior>
         <parameter parameter SuperiorParam
               {EmployeeCursor.EmployeeNumber}>
   <end cursor>
   -- List current employee
   {:Ruler3Col
         .EmployeeNumber .Name1 .CostPrice;
   }
   -- List all subordinates
   <repeating SubordinateCursor>
         {:Ruler2Col
            .Name1 .CostPrice;
         }
   <end repeating>
<end repeating>
```

The cursor *SubordinateCursor* is redefined for each iteration of the outer repeating block. Each time the cursor is defined, the *superiorParam* parameter is bound to the current employee. After defining the *SubordinateCursor*, we print out information about the current employee and then we iterate over the *SubordinateCursor*.

Scoping of Queries, Cursors, and Fields

Queries and cursors are scoped, which means that they cannot be used outside the block where they are defined. Likewise, the fields of a cursor cannot be accessed outside the scope where the cursor is executed—that is, a field on a particular cursor cannot be accessed outside a repeating block iterating over that cursor. When looking up fields we can either just name the field using the field syntax that is prefixing the field with a “.” or we can explicitly name the cursor in which the field is going to be looked up. When MPL determines which field to print it always tries to find the field in the nearest repeating block.

If we revisit the example above, the scoping rules mean that the fields *Name1* and *CostPrice* are looked up on the nearest cursor—that is, a field on a particular cursor cannot be accessed outside a repeating block iterating over that cursor. When looking up fields we can either name the field using the field syntax that is prefixing the field with a “.” or we can explicitly name the

cursor in which the field is going to be looked up. When MPL is the *SubordinateCursor*, a field on a particular cursor cannot be accessed outside a repeating block iterating over that cursor. When looking up fields we can either just name the field using the field syntax that is prefixing the field with a "." or we can explicitly name the cursor in which the field is going to be looked up. Since the field names exist on that cursor, the information is taken from the cursor. However, both cursors in the example define fields with these particular names. To access the fields with the same names on the outer cursor, *EmployeeCursor*, we must prefix the names with the cursor name.

Take a look at the inner repeating, where we access fields on both the inner and the outer cursor by prefixing with the cursor name where necessary. Note that the field *EmployeeNumber* does not exist on the inner cursor, so MPL finds the field on the outer cursor, even though the field is not prefixed with the cursor name:

```
<repeating EmployeeCursor>
   -- Bind the current employee as the superior employee.
   -- (cursor definition as before)


   -- List current employee
   -- (left out for brevity)


   -- List all subordinates
   <repeating SubordinateCursor>
   {:Ruler2Col
          .Name1          -- SubordinateCursor.Name1
          .CostPrice      -- SubordinateCursor.CostPrice
          .EmployeeNumber -- EmployeeCursor.EmployeeNumber
          EmployeeCursor.Name1
          EmployeeCursor.CostPrice
   }
   <end repeating>
<end repeating>
```

Multi-Level Queries

MQL supports multi-level queries, and that can be used in MPL prints. Let us consider an example of a multi-level query:

```
<query RegTimeQuery>
   mselect
   -- Outer query
    [
          Employee.EmployeeNumber,
          Employee.EmployeeName,


          -- Inner query
          [
             Job.JobNumber,
             -- alias for field
             Registered.SumNumberRegistered as Hours
```

```
        ] as cursor Job    -- name of inner cursor
    ] as cursor Employee -- name of outer cursor


    -- selecting from pre-defined Maconomy Jobs universe
    -- featuring joins over employees, timesheets,
    -- jobs, etc.
    from JobCost::Universes::Jobs
<end query>
```

The preceding query selects its data from the predefined Maconomy *Jobs* universe, which can be seen from the line:

```
from JobCost::Universes::Jobs
```

Two embedded queries are defined in this query definition. This corresponds to a “groupby” definition in plain SQL, but it enables us to do embedded iteration over the result of the query—that is, a field on a particular cursor cannot be accessed outside a repeating block that iterates over that cursor. When looking up fields we can either name the field using the field syntax that is prefixing the field with a “.” or we can explicitly name the cursor in which the field is going to be looked up. With MPL, we can use two embedded repeating blocks to iterate over the result.

In this example, the outer cursor iterates over all distinct values of the fields *Employee.EmployeeNumber* and *Employee.EmployeeName* in the *Jobs* universe. Note that fields in an MQL universe can contain “.” in their names. For each distinct value pairs of these two fields, the inner cursor will iterate over *Job.JobNumber* and *Hours* (which is an alias for the field *Registered.SumNumberRegistered*).

The outer and inner cursors have been assigned the names “Employee” and “Job,” and these names are used when iterating over a cursor that is defined using the query:

```
<cursor name=RegTimeCursor query=RegTimeQuery>
<end cursor>

-- Iterating the outer "Employee" cursor
<repeating RegTimeCursor.Employee>
    .Employee.EmployeeNumber .Employee.EmployeeName

    -- Iterating the inner "Job" cursor
    <repeating RegTimeCursor.Employee.Job>
    .Job.JobNumber .Hours
    <end repeating>
<end repeating>
```

Note how the cursors are accessed. The defined `<cursor>` name, *RegTimeCursor*, is used to prefix the cursor names that are defined in the query, and the full path to a cursor inside the query is used. To access the outermost query, *Employees*, you must write:

```
<repeating RegTimeCursor.Employee>
```

To access the innermost cursor, *Jobs*, which is embedded in the outermost cursor, *Employees*, you must type the full path:

```
<repeating RegTimeCursor.Employee.Job>
```

Aggregates

MQL also supports aggregates on values—that is, a field on a particular cursor cannot be accessed outside a repeating block iterating over that cursor. When looking up fields we can either name the field using the field syntax that is prefixing the field with a "." or we can explicitly name the cursor in which the field is going to be looked up. With MPL that is count, sum, and maximum, minimum, and average values. Revisiting and extending the example from the above we get:

```
<query RegTimeQuery>
   mselect
   -- Outer query
   [
          Employee.EmployeeNumber,
          Employee.EmployeeName,

          -- Inner query
          [
             Job.JobNumber,
             -- alias for field
             Registered.SumNumberRegistered as Hours
          ] as cursor Job    -- name of inner cursor
   ] as cursor Employee -- name of outer cursor

   -- make sum of hours available on outer cursor
   aggregate sum() on Hours

   -- selecting from pre-defined Maconomy Jobs universe
   -- featuring joins over employees, timesheets,
   -- jobs, etc.
   from JobCost::Universes::Jobs
<end query>
```

The only addition is the line marked in boldface. This line tells MQL to calculate the sum of the field *Hours* and make it available on the cursor one level out from where this field is defined. In our case the sum will be available on the *Employees* cursor as the field *Hours$SUM* and what we get is the sum of registered hours per employee.

We must update our example from before to include the new field:

```
<repeating RegTimeCursor.Employee>
   .Employee.EmployeeNumber .Employee.EmployeeName
   .Hours$SUM
   <repeating RegTimeCursor.Employee.Job>
   .Job.JobNumber .Hours
   <end repeating>
<end repeating>
```

The new field is marked in boldface. Adding a bit more formatting yields a print like the following, where we see the time registrations per job per employee and also see the sum of hours registered per employee.

Employee No.	Name	Hours Total
E018	Employee18	85.0
Job.		Hours
10250181		35.8
10250185		49.2
E019	Employee19	44.9
Job.		Hours
10250195		44.9
E021	Employee21	42.7
Job.		Hours
10250211		42.7

Metadata Cursor

When executing an MQL query a special top-level metadata cursor name *main* exists. However, by default this top-level cursor is hidden.

Let us have a look at a simple query definition:

```
<query EmployeeQuery>
   mselect
   EmployeeNumber,
   Name1
   from Employee
<end query>
```

As usual, we bind the query to a cursor:

```
<cursor name=EmployeeCursor
   query=EmployeeQuery>
<end cursor>
```

And then we execute the query:

```
<repeating EmployeeCursor>
   .EmployeeNumber .Name1
<end repeating>
```

To get access to the metadata top-level cursor, we change the cursor definition slightly:

```
<cursor name=EmployeeCursor
   query=EmployeeQuery
   showmaincursor+>
<end cursor>
```

We now have a multi-level cursor where the top-level cursor is called *main*, and the next level cursor is called *query*. The name of the inner cursor, *query*, is automatically chosen by MQL because we did not name the cursor in the MPL query definition. We can now change the repeating block like this:

```
<repeating EmployeeCursor.main>
   .query$RowCount -- Total row count of lowest sublevel
   .query$Date     -- Date of execution
   .query$Time     -- Time of execution
   .query$Count    -- Row count of query on next level
```

```
   <repeating EmployeeCursor.main.query>
   .EmployeeNumber .Name1
   <end repeating>
<end repeating>
```

The three first of the new fields are only present at the top level and show us the total row count of the query (which is the same as the sum of rows fetched by each of the iterations of the innermost cursor of the query). The last field, *query$Count*, is present on any level except the innermost and tells of the count of rows on the cursor immediately below. The first part of the field's name is named after the cursor that it is counting—in this case the *query* cursor.

If we revisit the first multi-level query example from before, we get a better feel of how automatic *count* fields work:

```
<query RegTimeQuery>
   mselect
   -- Outer query
   [
          Employee.EmployeeNumber,
          Employee.EmployeeName,

   -- Inner query
   [
          Job.JobNumber,
          -- alias for field
          Registered.SumNumberRegistered as Hours
   ] as cursor Job    -- name of inner cursor
   ] as cursor Employee -- name of outer cursor

   -- selecting from pre-defined Maconomy Jobs universe
   -- featuring joins over employees, timesheets,
   -- jobs, etc.
   from JobCost::Universes::Jobs
<end query>
```

We now bind the query using the `showmaincursor` attribute:

```
<cursor name=RegTimeCursor
   query=RegTimeQuery
   showmaincursor=true>
<end cursor>
```

We can now iterate over the cursor like this:

```
<repeating RegTimeCursor.main>
   .query$RowCount  -- Total row count of Jobs listed
   .query$Date      -- Date of execution
   .query$Time      -- Time of execution
   .Employee$Count  -- Number of Employees listed
   <repeating RegTimeCursor.main.Employee>
   .Job$Count     -- Number of Jobs for this Employee
   .Employee.EmployeeNumber
```

```
    .Employee.EmployeeName
    <repeating RegTimeCursor.main.Employee.Job>
           .Job.JobNumber
           .Hours
    <end repeating>
    <end repeating>
<end repeating>
```

We now see that we have two count fields: *Employee$Count* and *Job$Count*. The prefixes before the "$" sign are now the names of the cursors that we named in the query definition. Also note that the name of the *main* cursor now must be part of the prefix when accessing the cursors in the `<repeating>` tags.

Aggregates on the Top Level

If you use aggregates in MQL on fields that belong to the outermost cursor of your query, you must access the *main* cursor to access the aggregate values. These always exist on the cursor that is immediately above the cursor whose fields are aggregated. The cursor that is immediately above the top-level cursor that is defined in an MPL query definition is the *main* cursor.

Consider this query definition:

```
<query EmployeeQuery>
    mselect EmployeeNumber,
    Name1,
    CostPrice
    aggregate min() on CostPrice,
    max() on CostPrice
    from Employee
<end query>
```

Here, we are asking for the minimum and maximum cost price. To access the aggregate values we must access the main cursor:

```
<cursor name=EmployeeCursor
    query=EmployeeQuery
    showmaincursor+>
<end cursor>
```

Now, we can print the aggregate values (again with no formatting to simplify the example):

```
<repeating MyCursor.main>
    -- Print the aggregate values
    .CostPrice$Min .CostPrice$Max

    -- Print the employee information
    <repeating MyCursor.main.query>
    .EmployeeNumber .Name1 .CostPrice;
    <end repeating>
<end repeating>
```

Tags for Database Queries

In the previous section we saw a number of examples that illustrate the use of database queries in MPL. In this section we will give a more formal definition of the tags that are involved and their attributes.

Queries

The `query` tag:

```
<query attributes>
   MQL query
<end query attributes>
```

is used to define a new database query. You can place the query tag anywhere in the layout where a repeating can also be placed. A query definition is scoped and cannot be accessed outside the scope where it is defined. The `query` tag has no short form.

The tag is a parenthetical tag that encloses the MQL query that should be executed against the database. Note that defining a `query` does not execute the MQL query. The `query` must be instantiated to a `cursor`, and the `cursor` is iterated over in a repeating tag. The MQL query is executed just before the iteration starts.

The `query` tag takes one attributes:

- `name` specifies the name of the query. The name is used when a `cursor` tag binds the query. The attribute has the type *ID* and is nameless and mandatory.

For a thorough description of the `query` tag including examples of use, see “Database Queries.”

Example

```
<query EmployeeQuery>
   mselect EmployeeNumber, Name1, CostPrice
   from Employee
<end query>
```

Cursors

The `cursor` tag:

```
<cursor attributes>
   parameter bindings...
<end cursor>
```

is used to instantiate a previously defined `query` that yields a `cursor` that you can use in a `repeating` block. You can place a `cursor` tag anywhere in the layout where a `repeating` can be placed, but it can only refer to already defined queries that are in scope, which means that the referenced `query` must be defined in the same block or in an enclosing block in the layout. The query is *not* executed by being instantiated by a `cursor` tag. The query is executed only when the cursor is iterated over in a `repeating` block.

The tag is a parenthetical tag that encloses a number of `parameter` tags that must match the `query` definition’s *parameter* section.

The `cursor` tag takes three attributes:

- `name` specifies the name of the cursor. The name is used when a `repeating` tag refers the cursor for execution. The attribute has the type *ID* and is mandatory.

- `query` specifies the name of the query that is bound by this cursor definition. The referenced query must be defined in the same or an enclosing scope as the cursor definition. The attribute has the type *ID* and is nameless and mandatory.
- `showmaincursor` designates whether the *main* metadata cursor is accessible when iterating over the cursor in a `repeating` block. The attribute has the type *BOOLEAN* and is optional. Its default value is `false`.

For a thorough description of the `cursor` tag including examples of use, see “Database Queries.”

Example

```
<query EmployeeQuery>
   mselect EmployeeNumber, Name1, CostPrice
   from Employee
<end query>

<cursor name=EmployeeCursor query=EmployeeQuery>
<end query>
```

Cursor Parameters

The `parameter` tag:

```
<parameter attributes>
```

is used to bind formal parameters declared in `query` definition to actual values inside a `cursor` tag. The parameters given inside a `cursor` tag must match exactly the parameters that are declared in the `query` that is referenced by the enclosing `cursor` tag.

The `parameter` tag takes two attributes:

- `name` specifies the name of the formal parameter to bind. The name must match the name of one of the parameters that is declared by the referenced query. The attribute has the type *ID* and is mandatory and nameless.
- `value` specifies the value to bind to the formal parameter and can be any expression. The attribute has the type *EXPRESSION* and is mandatory and nameless.

For a thorough description of the `parameter` tag including examples of use, see “Database Queries.”

Example

```
<query EmployeeWithSuperior>
   mselect name1, costprice from employee
   where .SuperiorEmployee = superiorParam
   using parameters superiorParam type string
<end query>

<cursor name=EmployeeCursor query=EmployeeWithSuperior>
   <parameter superiorParam {"1010"}>
<end cursor>
```

Print Structure

All MPL layouts must be based on an existing layout. This layout—the *original layout*—specifies which cursors, fields, variables, and scripts can be used in the layout. To ensure that the definition of cursors makes sense and that the scripts are executed in the predefined order, all MPL layouts must have the same structure as the original layout.

Usually, you will not run into any problems in the practical development of layouts because you often base your new layouts on an existing original or structure layout. This section is therefore only relevant to you if you receive error messages about the structure of your layout or if you simply want to better understand the concept of print structure.

Structure

The MPL compiler ensures that the structure of an MPL layout sufficiently matches the structure of the original layout. The structure consists of all the printout's stacks, repetitions, conditions, headers, footers, and the paper itself. The structure describes how these elements are contained in one another.

If you export the structure layout, stacks, headers, and footers that do not have scripts assigned will be excluded.

Whether the MPL layout matches the structure of an original layout is defined using the concepts "script structure" and "stackless structure," which only contain a part of the structure described above.

Trees

When talking about the structure of MPL layouts it is often easier to perceive the structure as a tree. The root of the tree is the `paper` tag, and each block (repetition, condition, or stack) is a branch of the tree. You often refer to such a branch as a *node*. The analogy between MPL layouts and trees is used in the rest of this section to explain and illustrate the presented concepts.

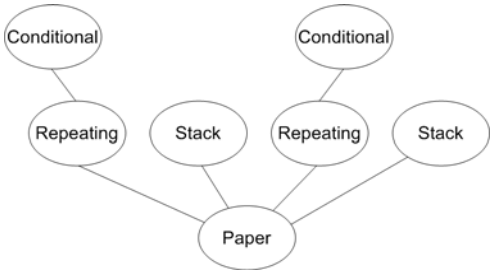
Example

See the structure layout for the printout "Print Employee Report."

```
<paper cursor=EMPLOYEE
   script="STANDARD_PAGESCRIPT1">
   <repeating cursor=CUSTOMERTIMEACTIVITY
         script="CUSTOMERTIMEACTIVITY">
   <conditional script="CUSTOMERTIMEACTIVITYTOTAL"
            variable=ShowTotalVar>
   <end conditional>
   <end repeating>
   <stack script="CUSTOMERTIMETOTAL">
   <end stack>
   <repeating cursor=INTERNALTIMEACTIVITY
            script="INTERNALTIMEACTIVITY">
   <conditional variable=ShowTotalVar>
   <end conditional>
   <end repeating>
```

```
   <stack script="EMPLOYEETOTAL">
   <end stack>
<end paper>
```

The corresponding tree looks as follows:



Script Structure

The *script structure* is a layout that is made up of all entities that can have a script assigned—that is, paper, stacks, repetitions, conditions, headers, and footers. An entity is a part of script structure if it either:

1. has a script assigned.
2. is a condition, repetition, while loop, or a paper that contains a block observing rule 1 or 2.

If you view the tree, the rule can be defined as follows: If a block observes rule 1, all conditions , repetitions, while loops, and a paper along the path between the block and the root must be included.

The script structure in the user-defined layout must be identical to the script structure in the original layout. The purpose of this condition is to ensure that scripts are executed the same number of times and in the same order for all layouts with the same original layout.

The examples in this section (below) have been constructed for the purpose of illustration. Note that layouts with a similar structure exist.

Example 1

The following structures have the same script structure:

```
<paper script="s1">
   <stack script="s2">
   <repeating cursor=c1>
   <end repeating>
   <end stack>
<end paper>
```

and

```
<paper script="s1">
   <stack script="s2">
   <end stack>
<end paper>
```

The script structures are identical, because the repeating block is not a part of the script structure.

Example 2

The following structures do not have the same script structure:

```
<paper script="s1">
   <repeating cursor=c1>
   <stack script="s2">
   <end stack>
   <end repeating>
<end paper>
```

and

```
<paper script="s1">
   <stack script="s2">
   <end stack>
<end paper>
```

In this example, the repeating block is part of the script structure, because it contains a block with a script.

Example 3

The following structures do not have the same script structure:

```
<paper script="s1">
   <stack script="s2">
   <end stack>
   <stack script="s3">
   <end stack>
<end paper>
```

and

```
<paper script="s1">
   <stack script="s3">
   <end stack>
   <stack script="s2">
   <end stack>
<end paper>
```

The script structures are not identical, because the order of the scripts that are used is different in the two fragments.

Example 4

The following structures have the same script structure:

```
<paper script="s1">
   <stack>
   <stack script="s2">
   <end stack>
   <end stack>
<end paper>
```

and

```
<paper script="s1">
   <stack script="s2">
   <end stack>
<end paper>
```

Example 5

The script structure for the printout “Print Employee Report” looks as follows:

```
<paper cursor=EMPLOYEE
   script="STANDARD_PAGESCRIPT1">
   <repeating cursor=CUSTOMERTIMEACITIVITY
         script="CUSTOMERTIMEACTIVITY">
   <end repeating>
   <stack script="CUSTOMERTIMETOTAL">
   <end stack>
   <repeating cursor=INTERNALTIMEACTIVITY
   <end repeating>
   <stack script="EMPLOYEETOTAL">
   <end stack>
<end paper>
```

Stackless Structures (MPL 1 and 2)

The concept of stackless structure has been simplified and loosened up in MPL 4; see the next section about the repeating structure for more details.

Until MPL 4, a *stackless structure* is defined as the structure that is obtained by removing all stacks and conditionals (that is, `stack` and `conditional` elements).

The stackless structure of the user-defined layout must be a subtree of the stackless structure of the original layout. A subtree is achieved as a result of pruning the tree; that is, nodes or branches must be removed from the top of the tree. In other words, you can leave out part of the stackless structure from the original layout, but you can only leave out an entire block, not part of its contents.

Example 1

The following structure:

```
<paper script="s1">
   <conditional variable=v1>
   <end conditional>
<end paper>
```

is a subtree of:

```
<paper script="s1">
   <conditional variable=v1>
   <repeating cursor=c1>
   <end repeating>
   <end conditional>
<end paper>
```

whereas the following structure:

```
<paper script="s1">
   <repeating cursor=c1>
   <end repeating>
<end paper>
```

is not a subtree, because the elements have not been removed from the inside.

Example 2

The stackless structure of the printout “Print Employee Report” looks as follows:

```
<paper cursor=EMPLOYEE
   script="STANDARD_PAGESCRIPT1">
   <repeating cursor=CUSTOMERTIMEACTIVITY
               script="CUSTOMERTIMEACTIVITY">
   <end repeating>
   <repeating cursor=INTERNALTIMEACTIVITY
               script="INTERNALTIMEACTIVITY">
   <end repeating>
<end paper>
```

Repeating Structure (MPL 4)

Repeating structure is a simplified and loosened up model of what was known as a *stackless structure* until MPL 4.

A *repeating structure* consists of all of the repeatings in an MPL 4 layout. We say that the repeating structure of a custom layout (or just *custom repeating structure*) must comply with the repeating structure of its original layout (or just *original repeating structure*).

Before digging into the details of what it actually means for a custom repeating structure to comply with its original repeating structure, let us first understand what purpose this check serves.

Every print environment contains a set of predefined cursors that can be iterated over by means of the `<repeating>` tag. These cursors can reference other cursors in the *where* clauses of their underlying *SQL* queries. For instance:

```
<repeating TimeSheetHeader >
   <repeating TimeSheetLine>
   <end repeating>
<end repeating>
```

for the `TimeSheetLine` cursor to contain only the lines of the `TimeSheetHeader` to which it belongs, `TimeSheetLine` must reference the `TimeSheetHeader` cursor in the *where* clause of its underlying *SQL* query. For this to work, it is necessary that any repeating over the `TimeSheetLine` cursor is embedded inside of a repeating over the `TimeSheetHeader` cursor.

In general, for all of the cursors in a custom layout we want to guarantee that their underlying references are initialized, which basically means that such a cursor is embedded in the same sequence of parent repeatings as at least one of the sequences of parent repeatings in the original layout.

As an example, let us consider the following original repeating structure:

```
<repeating A>
   <repeating B>
         <repeating D>
         <end repeating>
   <end repeating>
   <repeating D>
   <end repeating>
   <repeating C>
   <end repeating>
<end repeating>
```

The following custom repeating structure complies with the given original repeating structure, because each repeating in the custom structure is embedded in the exact same sequence of parent repeatings as in the original layout:

```
<repeating A>
   <repeating D>
   <repeating C>
   <end repeating>
   <end repeating>
<end repeating>
```

In particular, the repeating over *C* in the original structure expects to be embedded in a repeating over *A*, which is clearly the case in the custom structure as well. Likewise, *D* in the original structure is embedded both just in *A*, and in the sequence *A* followed by *B*. The custom structure complies with the first parent sequence, namely a single parent *A*. *A* has no parents both in the original and custom structure, hence the custom repeating structure complies with the original structure

On the other hand, the custom repeating structure below does not comply with the original repeating structure:

```
<repeating B>
   <repeating D>
   <end repeating>
<end repeating>
```

D in the original layout is embedded in *A* followed by *B*, or in a single parent repeating *A*. In the custom layout, though, *D* is embedded only in *B*; therefore, the repeating over *D* violates the repeating structure. Likewise *B*, because it is expected to be embedded in *A* like in the original structure.

In the given examples, we were using name aliases to represent repeating blocks. As far as the repeating structure is concerned, for the two repeatings to be considered the same, not only do they have to iterate over the same cursor, but also have the same values of attributes: `script` and `groupBy` when these attributes are set.

Advanced MPL

This section describes the remaining tags in MPL. Among other things, we will cover two tags that make it possible to control page breaks and to redefine embedded formatting constants. Furthermore, these sections will describe a number of tags that are not necessary in the structural definition of printouts, but make it easier to build elegant MPL definitions that are easily edited and maintained.

Additional Tags

border

A `border` tag specifies a vertical limit above which all preceding elements must be printed. Text elements that are specified before the `border` statement in the MPL definition will only be printed above the position of the border and continue on the following page, while elements that are specified after the border will be printed further down on the page. The elements after the border start at the height specified in the border.

```
<border attributes>
```

Note, however, that footers will be placed below the border.

The tag has the following attributes. You must specify exactly one of the following attributes:

- `top` specifies the position on the page where the border should be inserted. The attribute has the type *LENGTH* and defines the distance from the top of the page (that is, including page margins).
- `bottom` is similar to top, but specifies the distance to the bottom of the page. The border tag can only be specified in the `paper` tag; that is, it cannot be specified within other parenthetical tags.

If you specify more than one border tag, each tag only affects the positioning of the elements after the previous border tag:

```
elem1  -- placed in the top 10 cm of the page
...
elemm  -- placed in the top 10 cm of the page
<border top=10cm>
elemm+1 -- placed in the top 5 cm of the page
...
elemn  -- placed in the top 5 cm of the page
<border top=5cm>
elemn+1 -- placed 5 cm from the top of the page
```

Note that the element $elem_{m+1}$ following the first border is placed 10 cm from the top edge of the page, but because it is followed by a border, which is placed 5 cm from the top edge of the page, a page break is triggered before this element, and $elem_{m+1}$ is printed in the top part of a new page.

Example

Let us assume that you want to print out an invoice on stationery with a preprinted giro payment slip in the lower part of the page. Let us also assume that the height of the giro payment slip is 101.66 mm. You should therefore add a border of 101.66 mm from the bottom of the page.

The following is an extract from the print definition for “Print Invoice”:

```
<paper cursor=INVOICE script="SIDE">
   -- Invoice lines
   <repeating cursor=INVOICELINE
            script="INVOICELINEBLOCK">
   <repeating cursor=INVOICEBOMLINE
               script="BOMLINEBLOCK">
         ...
   <end repeating>
   <end repeating>
   -- Invoice totals
   "TOTAL DUE" .Currency .TotalCurrency;
   <conditional script="VATBLOCK"
            variable=VATBlockIncludedVar>
   <repeating cursor=CODE script="VATLOOPBLOCK">
         ...
   <end repeating>
   <end conditional>
   -- Giro payment slip
   <border bottom=101.6mm>
   <canvas>
   ...
   <end canvas>
<end paper>
```

Because the border has been specified after the invoice lines and totals, none of these will be printed in the bottom 101.6 mm of the page.

If we assume that the printout contains many invoice lines, these lines will be printed on the first page until reaching the 101.6 mm bottom border mark, after which a page break is triggered. Nor will the totals be printed in the bottom 101.6 mm of the page, meaning that if the printing of the invoice lines ends 12 cm from the bottom of the page, the first 18.4 mm of the totals will be printed, followed by a page break and the remaining totals. When all invoice lines and totals have been printed, the next print start position will be moved to 101.6 mm from the bottom of the page, where printing is resumed. Thereafter, the border has no effect.

newpage

You can force a page break anywhere in your definition using the `newpage` tag:

```
<newpage>
```

The `newpage` tag has no attributes in MPL 2. In MPL 3, the attribute `orientation` (of type ID) is supported in some cases. See “Paper Orientation Change” and “<newpage> in row no longer allowed” for more on `newpage` changes in MPL 3.

You can include the tag in stacking tags (however not in headers and footers). In MPL 2, it can also be used in arrays where it should be inserted as the only element in a row.

Example

If you want a page break after each time sheet in the printout from "A Printout Example: Time Sheets," you can force a page break for each iteration of the TIMESHEETHEADER repetition.

```
<repeating cursor=TIMESHEETHEADER
             script="S_TIMESHEETHEADER">
    ...
    {:[[][stretch+][][stretch+][][][][][][][][]]
           ...
           "" "" "" ""
           SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7 Grandtotal;
           <newpage>;
    }
<end repeating>
```

Here we have replaced `<skip 5mm>` with `<newpage>`. You could also choose to insert the page break outside the array:

```
    ...
    "" "" "" ""
    SumDay1 SumDay2 SumDay3 SumDay4 SumDay5 SumDay6 SumDay7
    Grandtotal;
    }
    <newpage>
<end repeating>
```

The result will be exactly the same. Note that a forced page break will be inserted after each time sheet, that is, there will also be a page break after the last time sheet (the last page of the printout is an empty page).

Alternative Text Tag

In the above we saw how you could set default values for attributes. This functionality is often used to set defaults for text formatting. However, you often need two sets of defaults; Maconomy's default fonts for fixed texts are formatted to 7 pt, whereas texts that are parts of fields (for example, hyphens in ranges or slashes in dates) are formatted to 9 pt.

To support two different default values for texts, MPL has an alternative text tag called `text2` that has the same characteristics as `text` but can be assigned separate default attribute values:

```
<text2 attributes>
```

You will often use the short form:

```
'text'
```

which is the short form for

```
<text2 title="text">
```

The short form of `text2` is different from the short form of `text` as the text is in single quotes instead of double quotes. `text2` has exactly the same attributes as text. The default value for `fontsize` is 9; the other default values are the same.

In "The front page" in "A Printout Example: Time Sheets" we became familiar with the use of the `text2` tag. Here we used the short form "/" in dates and "-" in ranges to ensure that the separators were assigned the same font sizes as the fields.

Grid

The `grid` tag enables you to define your own length units. The length unit is called grid, but it can vary according to orientation (horizontal or vertical). A grid definition (only one) is provided immediately after the margin definition (if any) in the print header.

```
<grid attributes>
```

The tag has the following attributes, which are both mandatory and have the type *LENGTH*.

- `hor` specifies the length of a horizontal grid unit.
- `ver` specifies the length of the vertical grid unit.

When you have defined a grid, you can use the tag as a unit alongside `cm`, `in`, and so on.

Example

To use Danish inches to specify the horizontal grid unit and Danish feet to specify the vertical length unit, you can define the following grid:

```
<mpl 2>
<layout "Simple"
   ancestor="Print_Inventory_Info_Card"
   ancestorlayout="Standard">
<page "A4">
<grid hor=2.62cm    -- A Danish inch
   ver=31.40cm> -- A Danish foot
<paper>
   <island width=5grid height=0.5grid>
   <end island>
<end paper>
```

Here the island is assigned a width of 5 Danish inches and a height of 1/2 Danish foot.

Colors

MPL enables you to change the print color. You can specify the printing color by means of two attributes:

1. The `color` attribute has type *ID* and the following legal values: black, red, blue, green, yellow, cyan, white, magenta.
2. The `rgb` attribute has type *RGB*, which is a triple that contains percentages of Red, Green, and Blue respectively: (*INTEGER*, *INTEGER*, *INTEGER*).

The `color` attribute is really a short form, because it represents all combinations of `rgb` where values are either 0 or 100, for example, (100,0,0) = red, (100,100,100) = white.

The `color` and `rgb` attributes are both style attributes, which can be specified for almost all tags. Furthermore, the `rgb` attribute is nameless. The attributes can be used in the following tags and tag types:

- All parenthetical tags
- Ruler columns
- vline
- text

- text2
- field
- var
- line
- hline
- title

When the three percentages in an `rgb` are the same (for example, (30, 30, 30)), the result is a shade of gray.

Example

```
"Text":color=blue
.FieldName:(50,50,50)
<stack rgb=(100,50,50)>
    ...
<end stack>
```

Images

The `image` tag:

```
<image attributes>
```

specifies that an image is to be printed.

MPL supports images of the following types:

- JPEG (Joint Photographic Experts Group). The usual file extensions are .jpg and .jpeg.
- PNG (Portable Network Graphics). The usual file extension is .png.

For an image to be printed, it is necessary that it must be imported into the database using the Maconomy client window Document Archives. In this window, documents are organized in user-defined document archives, so that all documents are identified by their names and document archives. A reference to an image document in MPL takes the following form:

DocumentGroup\DocumentName

It is mandatory that an image has exactly one of four attributes:

- `title` specifies a direct reference to the image document. The `title` attribute is nameless.
- `fieldname` specifies the name of a field that contains a reference to the image document.
- `varname` specifies the name of a variable that contains a reference to the image document.
- `expression` specifies an expression that yields a reference to the image document (that is, a value of type STRING). The `expression` attribute is nameless. For more information on expressions, see “Expressions.”

Example

```
<image title="MyImages\MyLogo.jpg">
```

If you want to specify the cursor from which the fieldname should be taken, you can use the following attribute:

- `cursor` specifies the name of the cursor from which the field is to be taken. The attribute has the type *ID*.

If you do not specify a cursor name, the value will be taken from the nearest cursor with a field of the specified name.

In addition to this, you can specify attributes concerning the image size and justification:

- `height` has the type *LENGTH* and specifies the height of the image.
- `width` has the type *LENGTH* and specifies the width of the image.
- `justification` has the type *ID*. It can take the values `left`, `center`, and `right`.

  If justification is specified, `width` must be specified as well. This applies to MPL for Universe Reports only.
- `link` specifies that the image is to function as a link. This applies to MPL for Universe Reporting only. For more information, see “Links.”

By default, the image will be scaled to the height and width that are indicated by these attributes. If only the `height` attribute is provided, the width will be scaled to maintain the default correlation between height and width for the image. The same goes when only the `width` attribute is provided. If neither a `height` nor a `width` attribute is provided, the default size of the image is used.

Note, however, that when an MPL layout is compiled, the default size of the image is not known. The height of an image is therefore assumed to be 0 if the `height` attribute is not provided, regardless of whether the `width` attribute is provided or not. The same goes for the width of the image. This means that a construction like:

```
<image title="MyImages\MyLogo.jpg"> "Text"
```

will result in “Text” being written on top of the image. To avoid this, while still keeping the default size of the image, you can use the following attributes:

- `scaleheight` has the type *BOOLEAN* and specifies whether the image height should be scaled. The default value is `true`.
- `scalewidth` has the type *BOOLEAN* and specifies whether the image width should be scaled. The default value is `true`.

If we change the example above to:

```
<image title="MyImages\MyLogo.jpg" scaleheight- height=100pt> "Text"
```

the image will keep its default size, but the compiler will assume that the image height is 100pt and consequently print “Text” 100pt below the top of the image.

Barcodes and QR codes

Barcodes and QR codes are ways of encoding information in a visual form that can be easily scanned by electronic devices. Every modern smartphone these days is capable of scanning QR codes and most types of barcodes, which avoids the unnecessary manual process of typing the information into an IT system.

The `<barcode>` tag enables you to place a barcode in an MPL layout in a very easy way. It is enough to specify the barcode type and the data to be encoded, and the barcode image will be generated by the MPL engine. Such a barcode can be treated as any other image in MP; all of the attributes applicable to images, apart from the ones used to specify the path to the image, also apply to barcodes.

The three main barcode attributes are:

- `data` specifies the data that is to be encoded in the barcode. Different barcode types expect different data formats. This attribute is mandatory, nameless, and of type *EXPRESSION*.
- `type` specifies which barcode type should be rendered to encode the given data. It is of type *ID*. Exactly one of the `type` or `typeExpression` attributes must be specified. MPL supports 15 types of barcodes:
  - `EAN13` — International Article Number consisting of 13 digits (EAN-13), equivalent to UCC-13.
  - `UPCA` — Universal Product Code A (UPC-A) consisting of 12 digits, equivalent to EAN-12 or UCC-12.
  - `EAN8` — International Article Number consisting of 8 digits (EAN-8), equivalent to UCC-8.
  - `UPCE` — Universal Product Code E, consisting of 8 digits.
  - `EANSUPP` — EAN-13 with an EAN-5 supplement (specified using the `dataSupplement` attribute).
  - `CODE128` — Plain barcode 128, variable length.
  - `CODE128UCC` — UCC/EAN-128 with a full list of 128 ASCII characters, variable length.
  - `INTER25` — Interleaved 2 of 5 barcode, encoding an even number of digits (variable length). When the attribute `generateCheckSum` is `true`, an extra checksum digit is generated and hence an odd number of digits is required as input data text.
  - `POSTNET` — Postal Numeric Encoding Technique, which can be:
    - ZIP — Consisting of 5 digits.
    - ZIP+4 — Consisting of 9 digits.
    - ZIP+4 + DP (Delivery Point) — Consisting of 11 digits.
  - `PLANET` — Postal Alpha Numeric Encoding Technique, variable length code consisting of digits only.
  - `CODE39` — Code 39, also known as Alpha39, Code 3 of 9, Code 3/9, Type 39, USS Code 39, or USD-3. Variable length, the valid character set includes: uppercase letters (A through Z), numeric digits (0 through 9), and a number of special characters (-, ., $, /, +, %, and space). An additional character (denoted '*') is used for both start and stop delimiters.
  - `CODABAR` — Also known as Ames Code, NW-7, Monarch, Code 2 of 7, Rationalized Codabar, ANSI/AIM BC3-1995, or USD-4. Variable length, allowed symbols: 12 from the group: (digits 0-9, dash, $), 4 symbols from the group: (:/+.), and 4 start/stop symbols: (ABCD, in some specifications EN*T).
  - `PDF417` — PDF417 barcode, variable length.
  - `DATAMATRIX` — DATAMATRIX barcode, variable length.
  - `QRCODE` — QR code, variable length.
- `typeExpression` — the same as `type`, except that this attribute is of type *EXPRESSION*, which allows for choosing a barcode type at run time, depending on the data in Maconomy. Exactly one of the `type` or `typExpression` attributes must be specified.

In addition to the above attributes, the `<barcode>` tag supports a number of attributes that are applicable only to certain barcode types. If not applicable to a certain barcode type, these attributes simply take no effect.

- `dataSupplement` — The `EAN-5` data supplement of type *EXPRESSION,* applicable for `EANSUPP` barcodes only.
- `guardBars` — Some barcodes can render guard bars around them. The attribute is of type *BOOLEAN* and should be set to `true` if the guard bars are to be generated.
- `textJustification` — Some barcodes support justification of the data text rendered along with the barcode. The attribute is of type *ID*, and its valid values are: `left`, `right`, and `center`.
- `generateChecksum` — Some barcodes can generate a checksum character for the data to be encoded. The attribute is of type *BOOLEAN* and should be set to `true` if a checksum is to be generated.
- `checksumText` — If the `generateChecksum` attribute is set to `true`, the generated checksum can be also displayed as part of the data text rendered along with the barcode, if the `checksumText` attribute is set to `true` as well.
- `startStopText` — Of type *BOOLEAN*, `true` if the data text rendered along with the barcode should be surrounded by the start/stop text (if applicable).
- `extended` — Of type *BOOLEAN*, `true` if the barcode should operate on the extended character set (if applicable).

In addition, the attributes `width`, `height`, `scalewidth`, `scaleheight`, `justification`, and `pos` work in the exact same way as for images.

The `<barcode>` tag was introduced in MPL 4 as of TPU 16 SP2.

Example

The following code snippets generate all 15 kinds of barcodes that MPL 4 supports.

```
<default tag=eval attribute=fontsize value=12>
<val ean8Data {"96385074"}>
^{"EAN 8 : " + ean8Data}
<barcode {ean8Data} EAN8 height=50pt>

<skip 10pt>
<val ean13Data {"5901234123457"}>
^{"EAN 13 : " + ean13Data}
<barcode {ean13Data} EAN13 height=50pt>

<skip 10pt>
<val upcaData {"785342304749"}>
^{"UPC-A : " + upcaData}
<barcode {upcaData} UPCA height=50pt>

<skip 10pt>
<val upceData {"03456781"}>
^{"UPC-E : " + upceData}
<barcode {upceData} UPCE height=50pt>

<skip 10pt>
<val eanSupp_ean {"1234567891234"}>
<val eanSupp_sup {"54321"}>
^{"EAN-SUP, EAN: " + eanSupp_ean + ", SUPP: " + eanSupp_sup}
<barcode {eanSupp_ean} datasupplement={eanSupp_sup} EANSUPP height=50pt>
```

```
<skip 10pt>
<val code128Data {"0123456789 hello $%*@"}>
^{"CODE 128 : " + code128Data}
<barcode {code128Data} CODE128 height=50pt>

<skip 10pt>
<val code128UCCData {"01912345678901213101000355l0ABC123"}>
^{"CODE 128 UCC: " + code128UCCData}
<barcode {code128UCCData} CODE128UCC height=50pt>

<skip 10pt>
<val code128UCCData2 {"(01)00000090311314(10)ABC123(15)060916"}>
^{"CODE 128 UCC, example 2: " + code128UCCData2}
<barcode {code128UCCData2} CODE128UCC height=50pt>

<skip 10pt>
<val codeInter25 {"41-1200076041-00"}>
^{"Barcode Interleaved 2 of 5: " + codeInter25}
<barcode {codeInter25} INTER25 height=50pt>

<skip 10pt>
<val codeInter252 {"06110123456783"}>
^{"Barcode Interleaved 2 of 5, 2: " + codeInter252}
<barcode {codeInter252} INTER25 height=50pt>

<skip 10pt>
<val codePostnet {"01234"}>
^{"POSTNET, ZIP: " + codePostnet}
<barcode {codePostnet} POSTNET height=20pt>

<skip 10pt>
<val codePostnet2 {"012345678"}>
^{"POSTNET, ZIP+4: " + codePostnet2}
<barcode {codePostnet2} POSTNET height=20pt>

<skip 10pt>
<val codePostnet3 {"01234567890"}>
^{"POSTNET, ZIP+4 and dp: " + codePostnet3}
<barcode {codePostnet3} POSTNET height=20pt>

<skip 10pt>
<val codePlanet{"01234567890"}>
^{"PLANET: " + codePlanet}
<barcode {codePlanet} typeExpression={"PLANET"} height=20pt>

<skip 10pt>
<val code39 {"MPL 4 IN ACTION"}>
^{"Barcode 3 of 9: " + code39}
<barcode {code39} CODE39 height=50pt >

<skip 10pt>
<val code39_2 {"MPL 4 in action"}>
^{"Barcode 3 of 9 Extended: " + code39_2}
<barcode {code39_2} CODE39 height=50pt  extended+ >

<skip 10pt>
<val codeBar {"A123A"}>
^{"CODABAR: " + codeBar}
<barcode {codeBar} CODABAR height=50pt startstoptext+>

<skip 10pt>
```

```
<val text {"This is some pretty long text with strange characters like
%^&^!~"}>
^{"PDF 417: " + text}:wrap+
<barcode {text} PDF417 height=50pt>

<skip 10pt>
^{"DATAMATRIX: " + text}:wrap+
<barcode {text} DATAMATRIX height=80pt>

<skip 10pt>
<val qrData {"MPL 4 supports barcodes now!"}>
^{"QRCode : " + qrData}
<barcode {"MPL 4 supports barcodes now!"} QRCODE height=150pt>
```

Running this code snippet as a part of an MPL 4 layout results in the printout that is shown on the next two pages.

EAN 8 : 96385074

EAN 13 : 5901234123457

UPC-A : 785342304749

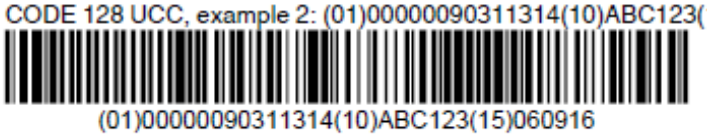
UPC-E : 03456781

EAN-SUP, EAN: 1234567891234, SUPP: 54321

CODE 128 : 0123456789 hello $%*@

CODE 128 UCC: 0191234567890121310100035510ABC123

CODE 128 UCC, example 2: (01)00000090311314(10)ABC123(15)060916



Barcode Interleaved 2 of 5: 41-1200076041-00

Barcode Interleaved 2 of 5, 2: 06110123456783

POSTNET, ZIP: 01234

POSTNET, ZIP+4: 012345678

Setting Next Page Number

It is sometimes convenient to be able to control page numbering in an MPL printout. For example, when you are batch printing invoices you might want each invoice to start with page number 1, instead of just continuing with increasing page numbers. Also, when you are including static PDFs, it is sometimes convenient to set the page number to account for the size of the included PDF document.

This functionality is provided by the `<nextpagenumber>` tag, which has one mandatory and nameless attribute value of type *EXPRESSION*. The semantics of this tag are that the value of the `PageNumber` variable will be set to the new value on the next page to be printed, after the `<nextpagenumber>` tag has been executed. The `PageNumber` variable controls the page numbers that are printed in an MPL layout.

The `<nextpagenumber>` tag was introduced in MPL 4 as of TPU 16 SP2.

Example 1

In the following example, the `PageNumber` variable will be reset to 1 after printing each invoice.

```
<paper cursor=InvoiceEditingHeader>
  "Invoice body"
  ...
  <nextpagenumber {1}>
<end paper>
```

Example 2

To see how you can use the `<nextpagenumber>` tag with `<includepdf>`, see “Example 2” in “Including static PDF documents.”

Including static PDF documents

MPL is a language for dynamically generating PDF documents based on the data that is in Maconomy. There are certain situations, however, when you want to include an already generated, static PDF into a Maconomy print, for example when attaching a customer satisfaction survey or a letter of fulfillment to an invoice.

The `includepdf` tag

```
<includepdf attributes>
  ...
<end includepdf>
```

has been designed to meet these exact needs. It enables you to include static PDF documents from the Maconomy Document Archive directly into an MPL print.

The following attributes are supported:

- `path` denotes a path to the document in the Document Archive that we want to include. It is of type *EXPRESSION,* meaning that the path can be calculated dynamically at run time, depending on the context that we are currently in. The path attribute is mandatory and nameless.
- `skipNewPageAfterPdf` specifies whether there should be a page break (new page) added after the included PDF. This attribute is of type *BOOLEAN* and defaults to `false`.

  There are two basic scenarios when including a PDF in an MPL print. We might want to include the PDF at the end of a `paper` tag content, for example when attaching a satisfaction survey document to an invoice. In this case the value of `skipNewPageAfterPdf` should be set to `true`, because the `paper` tag will make an implicit new page after each `paper` cursor record has been printed. On the other hand, when including a PDF somewhere in the middle of a `paper` tag, we want to make a page

break after the included PDF, so we should set the `skipNewPageAfterPdf` to `false`, which is its default value.

When including a PDF in an MPL print, a natural question arises: what about the page numbering? Should we account for the included PDF pages or not? MPL leaves it up to the layout developer, who can decide to either account for the included PDF document's number of pages or not. To this end, the `includepdf` tag brings a variable `noOfIncludedPdfPages` into its children's scope, where we can use the `<nextpagenumber>` tag to set the next page number to any value we wish.

The `<includepdf>` tag was introduced in MPL 4 as of TPU 16 SP2.

Example 1

When including a PDF somewhere in the middle of a `paper` tag, we would usually do something like this (provided that there is a MyExample.pdf document in the PDFs group in the Document Archive).

```
<paper>
  "First part of the print"
  ...
  <includepdf {"pdfs\MyExample.pdf"}>
  <end includepdf>
  ...
  "Second part of the print"
  <footer onLastPage+>
   ^{"Page " + PageNumber}
  <end footer>
<end paper>
```

Note that an automatic page break will be inserted after the included PDF document.

Example 2

Let us modify the previous example so that we account for the number of pages of the included PDF in the document page numbering. To this end, we will use the `<nextpagenumber>` tag together with the `noOfIncludedPdfPages` variable. We must remember that the `<nextpagenumber>` tag will set the page number for the next page to be printed after this tag has been executed. For that reason, we should postpone adding a page break after the included PDF to the point after the execution of the `<nextpagenumber>` tag. These considerations lead to the following piece of code (changes to the previous example appear in bold):

```
<paper>
  "First part of the print"
  ...
  <includepdf {"pdfs\MyExample.pdf"} skipNewPageAfterPdf+>
    <nextpagenumber {PageNumber + noOfIncludedPdfPages + 1}>
  <end includepdf>
  <newpage> --triggers nextpagenumber to take effect
  ...
  "Second part of the print"
```

```
  <footer onLastPage+>
   ^{"Page " + PageNumber}
  <end footer>
<end paper>
```

Example 3

When including a PDF document at the end of a `paper` tag, we should set the `skipNewPageAfterPdf` attribute to `true`, because the `paper` tag will make an implicit page break for each `paper` cursor record.

```
<paper>
  "The paper content"
  ...
  <includepdf {"pdfs\MyExample.pdf"} skipNewPageAfterPdf+>
  <end includepdf>
<end paper>
```

goto

The `goto` tag:

```
<goto attributes>
```

specifies moving the print cursor to a position on the page indicated by the attributes.

It is mandatory that a `goto` has exactly one of two attributes:

- `top` specifies the position on the page to which the cursor should move. The attribute has the type *LENGTH* and defines the distance from the top of the page (that is, including page margins).
- `bottom` is similar to `top`, but specifies the distance to the bottom of the page.

Example

```
<goto top=5cm>
```

Note that you can only move the print cursor forward. This means that a construction like:

```
<goto top=6cm> "Text1"
<goto top=5cm> "Text2"
```

will result in “Text2” being printed on the page that follows the page that contains “Text1.”

Together with the `stop` attribute, the `goto` tag enables you to make border constructions within all parenthetical tags:

```
<stack stop=5cm>
    ...
<end stack>
<goto bottom=5cm>
```

This construction will have the result that nothing (except footers) from the stack is printed below the 5cm margin to the page bottom. When the stack has been printed, the printing will continue 5cm from the page bottom.

title

Within Maconomy, a default title is associated with some fields and variables. In MPL, this default title can be accessed through the tag:

```
<title attributes>
```

It is mandatory that a title has exactly one of two attributes:

- `fieldname` has type *ID* and specifies the name of a field.
- `varname` has type *ID* and specifies the name of a variable.

If you want to specify the cursor from which the fieldname should be taken, you can use the following attribute:

- `cursor` specifies the name of the cursor from which the field is to be taken. The attribute has the type *ID*.

If you do not specify a cursor name, the value will be taken from the nearest cursor with a field of the specified name. Note that you can print titles “out of scope,” that is, you can print the default title of a field even though you are not within the scope of the cursor to which the field belongs. To do this, the title must be fully qualified with the cursor name(s).

Example

```
<title fieldname=JournalNumber cursor=Journal>
```

Often you will see a short form being used:

```
[cursorname.field]
[.field]
[VAR]
```

Example of usage “out of scope”:

```
<title fieldname=cursor1.cursor2.JournalNumber>
```

The `title` tag is closely related to the `text` tag, and you can therefore specify the same attributes as for `text`. These attributes all have the same meaning, except for:

- `title` has the type *STRING* and specifies a title, which is used if the system default title is empty.

Example

```
<title varname=VARNAME title="DefaultTitle">
```

or

```
[VARNAME]:"DefaultTitle"
```

All attributes that are nameless for the `text` tag are also nameless for the `title` tag.

Other attributes:

- `uppercase` has the type *BOOLEAN* and specifies whether the title should be writ- ten in uppercase. The default value is `false`.

Example

```
[.FieldName]:uppercase+
```

Length Constants

MPL enables you to define length constants. By assigning a name to a length that you want to use in more than one place, you ensure that any change that you make to the length will affect all occurrences.

MPL also has a number of predefined lengths that control the formatting. You can refer to these constants to align a length with an embedded length, or you can change the lengths to change the formatting of your print.

All length constants have an orientation (horizontal or vertical) and can only be used with the specified orientation.

Define

You can use the define tag to define a new length constant:

```
<define attributes>
```

Definitions with the `define` tag must be inserted in the beginning of parenthetical tags, that is, along with `default` statements. The defined constants apply within the parenthetical tag in which they are defined (and inside all embedded tags).

The tag has the following attributes:

- `name` specifies the name of the new constant. The attribute is mandatory and has the type *ID*.
- `value` specifies the value of the new constant. The attribute is mandatory and has the type *LENGTH*.
- `orientation` specifies the orientation of the new constant. The attribute has the type *ID* and can be assigned the values `vertical` or `horizontal`. The attribute is mandatory if the value is specified using a `grid` unit. If the attribute is not set, the orientation is derived from the first use of the constant.

Example

Consider the following:

```
<define name=nine value=9mm orientation=horizontal>
<define name=mylen value=1.5cm>
"Hello":indent=nine
"Hi!":indent=mylen
```

Both lengths can only be used horizontally: `nine` because it has been specified using the `orientation` attribute, and `mylen` because it is used in `indent`, which is a horizontal length. The resulting printout is:

Predefined Lengths

MPL has a number of predefined length constants. These lengths can be used in the same way as lengths defined using `define`, that is, they can be used as values in all length attributes. For example, you can use the radius of the island curves as margins:

```
<island leftmargin=IslandCornerRadius
   rightmargin=IslandCornerRadius>
```

```
   ...
<end island>
```

You often redefine these lengths (see the next section) and thus change the formatting of MPL layouts.

The following lengths are predefined in MPL:

- `InterColumnSpacing` specifies the distance between two columns if no column separators have been inserted between the columns. Moreover, the length defines the distance from a column to a column separator if any such has been inserted. It is the space that you can underline with the attributes to `hline`, namely `left` and `right`. The orientation of the length is horizontal and its default value is 4 points.
- `IslandTitleIndent` specifies the distance from the left-justified island title to the left edge of the island (and the distance from a right-justified island title to the right edge of the island). For further illustration of this length, see the figure in “Island Lengths” in “Tips and Tricks.” The default value is 5 points.
- `IslandCornerRadius` specifies the radius of the quarter circles that make up the corners of the islands (when the `rounded` attribute has not been set to `false`). The default value is 5 points.

The orientation of `IslandCornerRadius` is horizontal.

- `MinTextWidth` specifies the smallest width allowed for a (used) text block. If not set, the default value is 0 (zero) and text blocks will therefore never be wider than their actual width. The default value is 0.
- `MinFieldWidth` specifies the smallest width allowed for a field or a variable. As the contents of the field is not known during compilation/interpretation of the layout, this length ensures that fields always have a minimum width. The default width only provides a space for 5-6 characters with normal font which is rarely enough space for text fields. On the other hand it is also inexpedient to set a larger width than is actually needed; often the width of fields will be controlled by other attributes (for example, width and column stretchability). The default value is 18 points.
- `HlineSpacing` specifies the distance between lines in an `hline` with the multi attribute set to a higher value than 1. This constant is rarely used. The default value is 2 points.
- `HlineSkip` specifies the extra space after an `hline`. The default value of the constant is 0. This constant is rarely used.
- `HeaderSkip` specifies the extra space after headers, that is, the distance between a header and the main text (and between a page header and a block header). The default value is 4 points.
- `FooterSkip` specifies the extra space before footers, that is, the distance between the main text and a footer (and between a block footer and a page footer). The default value is 4 points.
- `MinBaseLineSkip` specifies the smallest distance between the baseline of text lines. Thus you set the line spacing by redefining this constant. Note that in certain cases the line spacing can seem smaller than `MinBaseLineSkip`. This usually occurs when two stacking tags of different height are placed alongside in a row. Here the baseline of the rows is the bottom text line by default. This means that the baseline of the row can be `MinBaseLineSkip` from the preceding baseline, without it being the case of the baseline of the top element. The default value is 10 points.

redefine

You can assign a new value to an existing length constant using this constant:

```
<redefine attributes>
```

The tag can occur in the same places as the `define` tag and has the same attributes:

- `name` specifies the name of the constant to which you want to assign a new value.

  The length constant should be known, meaning that it should either be predefined in MPL or be defined using a `define` tag in the scope where the `redefine` tag is specified. The attribute is mandatory and has the type *ID*.

- `value` specifies the new value that you want to assign to the constant. The attribute is mandatory and has the type *LENGTH*.

- `orientation` specifies the orientation of the new value of the constant. Because the constant keeps the orientation that has possibly been specified earlier, you should only use this attribute if the constant has not already been assigned an orientation.

  The attribute has the type *ID* and can be assigned the values `vertical` or `horizontal`. The attribute is mandatory if the value has been specified with a `grid` unit, and the length constant has not already been assigned an orientation. If `orientation` is not specified, and the constant does not have an assigned orientation, the orientation is derived from the first use of the constant.

If the constant has not already been assigned an orientation, a new orientation will not be tied to the old value. When the scope of the `redefine` tag is concluded, the constant will, therefore, still not have an assigned orientation.

The new value of the length constant will apply to the parenthetical tag in which the `redefine` tag is inserted. After the parenthetical tag, the original value applies.

Example 1

Consider the following:

```
<paper>
   <define name=mylen value=1cm>
   "1cm indent":indent=mylen
   <stack>
   <redefine name=mylen value=2cm>
   "2cm indent":indent=mylen
   <end stack>
   "1cm indent":indent=mylen
<end paper>
```

In the example the first and last text blocks are indented 1 cm, while the middle text block is indented 2 cm. Note how `stack` is exclusively used to create a local scope.

Example 2

In this example we use `redefine` to set some of the embedded length constants:

```
<redefine name=MinBaseLineSkip value=1cm> "Hello"
"Hello again"
<stack>
```

```
    <redefine name=InterColumnSpacing value=1cm>
    {:[|[]|[]|]
    <hline 2>;
    "1" "2";
    <hline 2>;
    }
    {
    "a" "b";
    }
<end stack>
{:[|[]|[]|]
    <hline 2>;
    "1" "2";
    <hline 2>;
}
```

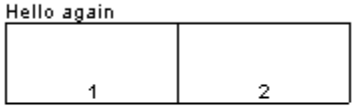
Redefining `MinBaseLineSkip` ensures that the line spacing is at least 1 cm (both between the simple text lines and the rows).

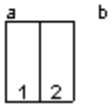
Redefining `InterColumnSpacing` changes the distance between columns to 1 cm. The difference is made more obvious by the fact that the same array is repeated after the stack in which the redefinition no longer applies.

The resulting printout is:

Hello

Hello again





Margins

The `margin` tag is part of the heading in the MPL definition. It should be specified immediately following the `page` tag and specifies the margins of the printout. If no margin tag is specified, the margin values from the Maconomy client window Paper Formats are used instead.

The tag is used as follows:

`<margins` *`attributes`*`>`

The tag has the following attributes which are all mandatory and have the type *LENGTH*:

- `top` specifies the top margin.
- `bottom` specifies the bottom margin.
- `left` specifies the left margin.
- `right` specifies the right margin.

Margins in Standard Prints

Some comments should be added to the use of the `margin` tag in connection with the standard prints. The behavior of this tag depends on the version of the current TPU and on the client that is using the standard printout.

Defaults

All attributes in all tags have a default value that is used if the attribute is not specified (not mandatory attributes). You can, of course, always change this value, but if you must do this often, it is easier to change the default value. A typical example is when you want to use another default font or font size. You can also use a style where islands always have the same inner margin and you therefore need to change the default value for the four attributes that decide the inner margins of the islands.

It often results in a more elegant output if the margin size is larger than zero. The disadvantage is the fact that rulers that are defined outside the island are not recognized inside the island if `leftmargin` or `rightmargin` has been set to a value that is other than zero.

You can change default values using the `default` tag that is specified in the beginning of parenthetical tags (for example, in the beginning of `paper` or in a repetition). The changing of default has an effect on the inner part of the parenthetical tag that is surrounded by the default statement (see the example below).

The syntax is:

```
<default attributes>
```

The tag has the following attributes that are all mandatory:

- `tag` specifies to what tag the attribute for which you want to change the default value belongs. In other words, the combination of `tag` and `attribute` identifies the attribute. The attribute has the type *ID*.
- `attribute` specifies for what attribute you want to change the default value. The attribute has the type *ID*.
- `value` specifies the default value for the specified attribute. The attribute type depends on what attribute is specified.

Because mandatory attributes must be specified, these have no default values and thus cannot be changed using the `default` tag.

Note that the defaults for `text` and `vline` will also affect column separators. You can specify default attribute values for the attributes in columns by using the `col` tag name (affecting both ruler and subruler definitions).

Example 1

The default font size for texts is 7 pt, whereas it is 9 pt for fields and variables. This results in an elegant layout when you print fixed texts in capitals, but it is not always as elegant if you want to print fixed texts in normal size.

The default font size can therefore be changed by writing:

```
<paper>
   <default tag=text attribute=fontsize value=9>
   ...
<end paper>
```

Because the default value is specified in the beginning of the `paper` tag, the default value will apply to the entire print definition.

As an exercise, you could insert this line into the time sheet layout defined earlier. Note that the island titles are also printed in 7 pt by default, and that it can also be useful to change this default value.

Example 2

The following example illustrates the scope of the `default` statement:

```
<paper>
   <island>
   <default tag=island
            attribute=leftmargin value=1cm>
   <default tag=island
            attribute=rightmargin value=1cm>
   <default tag=island
            attribute=topmargin value=5mm>
   <default tag=island
            attribute=bottommargin value=5mm>
   ...
   <island>
          "Hello"
          <island stretch- rightmargin=4cm>
             "Hi!"
          <end island>
   <end island>
   ...
   <end island>
<end paper>
```

Here the two inner islands will be assigned an inner margin, while the outer islands will not. The innermost island will have a right margin of 4 cm because of the specified attribute, while the other three margins are defined by the `default` statements.

Inheritance of attribute values

In MPL, tags can inherit values of certain attributes from their parent tags. The following section describe the rules guiding the inheritance of certain kinds of attributes.

Style Inheritance

In a layout you sometimes want to have different styles in different sections of the layout. For instance, you may want a different font for your page header or right justification in a column of an array. You can achieve this by specifying style attributes on parenthetical tags and ruler columns.

The attributes classified as style attributes are:

- justification
- fontname
- fontsize
- bold

- italic
- underline
- color
- rgb
- dateformat
- timeformat
- amountformat
- realformat
- integerformat
- booleanformat

In this section we will use the term *block* as a common name for all the parenthetical tags—`stack`, `array`, `repeating`, `conditional`, `canvas`, `island`, `page`, `frontpage`, `header`, `footer`, and `link`—including `layout`. Note, however, that `island` tags behave differently from the other block tags as described below.

Block Inheritance

The meaning of adding a style attribute to, for example, a `text` tag is to print the text with a given style. When you add a style attribute to a block, the result is that this style attribute is passed on to all tags within its scope:

```
<stack justification=center>
   "Text"
   cursor.field
   VAR
<end stack>
```

will give the same result as:

```
<stack>
   "Text":justification=center
   cursor.field:justification=center
   VAR:justification=center
<end stack>
```

Note, however, that inherited style attributes will never overwrite any attributes on the tag itself. Thus:

```
<stack
   justification=center> "Text"
   cursor.field:justification=left
   VAR
<end stack>
```

will give the same result as:

```
<stack>
   "Text":justification=center
   cursor.field:justification=left
   VAR:justification=center
```

```
<end stack>
```

Similarly, inherited style attributes do not overwrite default style definitions:

```
<stack justification=center>
   <default tag=field attribute=justification value=left> "Text"
   cursor.field
   VAR
<end stack>
```

will again give the same result as:

```
<stack>
   "Text":justification=center
   cursor.field:justification=left
   VAR:justification=center
<end stack>
```

Note that unlike the `default` definitions, the inherited style applies to all tags, regardless of type.

Column Inheritance

It is also possible to add style attributes to columns in ruler definitions:

```
<ruler Ruler1 [ [italic+] [fontsize=10] ]>
```

The result of this is that the first column is written in italic, and the second with font size 10.

Blocks placed in the columns will also inherit the tags and pass them on as described above. The rules about not overwriting tag and default attributes described above also apply to column inheritance.

However, column inheritance has precedence over general inheritance:

```
<ruler Ruler1 [ [] [fontsize=10] ]>
{ :Ruler1:fontsize=8
   "Text1" "Text2";
}
```

will result in “Text1” being printed with font size 8 and “Text2” with font size 10.

Islands

The `island` tag differs from the other parenthetical tags in that the style attributes specified for an island concern the island title and the island frame. Style attributes on islands are therefore not passed on to the elements within the island. The style attributes are, however, passed on to the island itself:

```
<stack justification=center fontsize=10>
   <island "IslandTitle" fontsize=20>
   "Text"
   <end island>
<end stack>
```

will give the same result as:

```
<stack>
   <island "IslandTitle" fontsize=20 justification=center
   titlejustification=center>
```

```
    "Text":fontsize=10
    <end island>
<end stack>
```

Style Precedence Summary

The style of a tag is decided as follows in order of precedence:

1. Style attributes on the tag itself.
2. Default definitions in the same scope as the tag.
3. Column style attributes for the column containing the tag, if any.
4. Style attributes for the surrounding parenthetical tag(s) including the `layout` tag.

Note that when a `color` attribute is inherited, it does not overwrite an `rgb` attribute, and vice-versa. Furthermore, where both the `color` and `rgb` attributes exist on a tag, the `rgb` attribute is always used in preference over the `color` tag.

Block Attributes

The attributes described in this section are defined on all non-fixed parenthetical tags: `stack`, `array`, `repeating`, `conditional`, `island`, and `canvas`.

- `keeptogether`: The attribute `keeptogether` has type *BOOLEAN* and indicates whether the block should be printed on the same page if possible. The default value is `false`. If the value is `true`, the block is moved to the top of the next page if the current page does not have room for it.

Example

```
<stack keeptogether+>
    "This block" "should be printed"
    "on the same page"
<end stack>
```

Note that for repeating tags, the actual number of elements inside the block is not known at the time of compilation. Therefore, adding a `keeptogether` to these tags will not guarantee that the entire repetition is written on the same page. It does however ensure that each element of the repetition is written on the same page if possible.

Example

```
    <repeating Cursor script="L_Cursor" keeptogether+>
           .fieldname
           "This field and this"
           "text should be printed"
           "on the same page, but the"
           "next field may be written"
           "on the next page"
    <end repeating>
```

- `movepos`: The attribute `movepos` has type *BOOLEAN* and indicates whether the position of the print cursor should move after the block has been printed. The default value is `true`. If the value is `false`, the tags following the block are printed on top of it.

This attribute enables you to print on top of background frames, watermark images, and so on.

Note that `movepos` only works within one page. Adding a `movepos-` to a block therefore automatically triggers that `keeptogether` is true.

Example

```
<stack movepos->
       "This text will be overwritten"
<end stack>
"This text overwrites the text in the block"
```

- `stop`: The attribute `stop` has type *LENGTH* and indicates a distance to the bottom of the page. When a `stop` attribute is added to a block, it means that no contents of the block are written below the point given by the stop value.

Some printers—among others, matrix printers—do not support this feature.

The `stop` attribute can, together with the `goto` tag, be used to create border behavior within parenthetical tags (see `goto`).

Example

```
<stack stop=100pt>
       ...
<end stack>
```

Standard Printouts in the Maconomy Clients for Windows and Java

When selecting “Print...” and “Print this” in the standard Maconomy client, margins are defined in MPL along with paper format and orientation. This is illustrated by the following MPL fragment:

```
<mpl 1>
<layout title="16 Columns Horizontal"
   print="Print Finance Report"
   originallayout="16 Columns Horizontal">
<page "A4" landscape>
<margins top=24pt bottom=30pt left=20pt right=20pt>
...
```

If `orientation` is left out, `portrait` is chosen as default. If `margins` is left out, the default is derived from the margins of the paper format. The paper format is mandatory.

In the Windows client, paper formats are defined in the window Paper Formats. This allows Maconomy users to define their own paper formats, and have MPL prints be formatted according to those definitions. On the other hand, paper formats need not be related to paper formats available on printers.

MPL is preformatted when it is installed, and all elements are given positions. This formatting is performed with no knowledge of the printer that will be used. In older versions of Maconomy (before TPU 27), the server formatted the print according to “standard” margins.

On Windows servers, the standard margins were derived from the default printer installed on the server. On UNIX they were given fixed values.

In later versions of Maconomy, printable elements are given positions relative to the paper edge. When the report is actually printed, it is adjusted to the physical margins of the selected printer. This ensures that prints look the same on all printers.

To ensure backward compatibility, a configuration option controls whether prints should be formatted in the new way or the old way. This option is set in the file Maconomy.ini located in the folder MaconomyDir/Definitions on the server. To place printable elements in the new way, set `RunTimePrinterDependence=true`.

Unless you have important reasons to do otherwise, you should always set this to `true`.

The meaning of the name is that the formatting of the report depends on the actual printer chosen at run time.

If a few of your reports have been aligned to specific printers in an old system, you can force the use of old formatting by setting the attribute `runtimeprinterdependence` to `false` on the `layout` tag:

```
<layout title="16 Columns Horizontal"
   print="Print Finance Report"
   originallayout="16 Columns Horizontal"
   runtimeprinterdependence->
```

If the report is printed from Maconomy client for the Java™ platform, PDF is produced and presented in Acrobat Reader. The PDF that Maconomy creates contains information on the dimensions of the paper format chosen. Acrobat Reader allows the user to choose that rotation and paper source should be selected according to these dimensions. Furthermore, you can scale the print; choosing not to scale ensures that the output matches the specifications of your MPL.

If the report is printed in the Windows client, you can displace the print in the Print Displacement window (found on the “File” menu). This is a last resort, only to be used if your print does not come out right on your printer. Print displacements are not saved as part of user settings.

The paper dimensions specified in MPL are ignored by the Windows client. Instead, you must manually choose paper format and orientation in the Page Setup window (on the “File” menu). The format chosen in the Page Setup window also determines the blue frame in Print Preview windows.

RGL

This is similar to standard prints as described above. RGL printouts are affected by the `RunTimePrinterDependence` option.

Universe Reports and the Analyzer

These are similar to standard prints as described above, except for the following: If a report is too wide, the paper size in the generated PDF grows with the content. In this case, using Acrobat Reader’s scaling possibilities is recommended.

These are not affected by `RunTimePrinterDependence`.That is, it always works in the new way.

Filling in Forms

Printouts from Maconomy are often used to fill in preprinted forms. Forms can also be combined with printouts with a free design, for example, combining a special offer with a tear-off coupon, or an invitation to a business event with a registration form. In Denmark and other European countries an invoice is often accompanied by a giro form that allows a direct transfer of funds to the issuer of the invoice. The format of a giro form is defined in every detail to allow banks to read the forms automatically. This section describes the design of a specific sort of giro form, but the techniques employed are readily applicable to other preprinted forms.

A Giro Form

In the following example, it is assumed that all paper margins have been set to “0.”

```
<margins left=0cm right=0cm top=0cm bottom=0cm>
```

Furthermore, it is assumed that the font “OCR-B” is installed (below we use font “SAdvOCR-B”). The following lines of MPL will fill in a preprinted giro form at the bottom of an invoice:

```
<border bottom=101.6mm>
<stack>
   <default tag=field attribute=fontname value="SAdvOCR-B">
   <default tag=var attribute=fontname value="SAdvOCR-B">
   <default tag=text attribute=fontname value="SAdvOCR-B">
   <default tag=field attribute=fontsize value=14>
   <default tag=var attribute=fontsize value=14>
   <default tag=text attribute=fontsize value=14>
   <canvas height=101.6mm>
   .ThePaymentCustomer
   :(5mm,26mm):width=33.5mm:right:fontname="Courier":fontsize=9
   .InvoiceNumber
   :(45mm, 26mm):width=15mm:right:fontname="Courier":fontsize=9
   .InvoiceDate
   :(66.5mm, 26mm):width=18mm:right:fontname="Courier":fontsize=9
   <stack (10mm,34mm)>
         <default tag=var attribute=fontsize value=9>
         <default tag=var attribute=fontname value="Courier">
         <default tag=var attribute=width value=70mm> PaymentAddress1
         PaymentAddress2 PaymentAddress3 PaymentAddress4 PaymentAddress5
         PaymentAddress6 PaymentAddress7 PaymentAddress8
   <end stack>
   <stack (91mm,17mm)>
         <default tag=var attribute=fontsize value=9>
         <default tag=var attribute=fontname value="Courier">
         <default tag=var attribute=width value=40mm>
            CompanyGiroNumber:bold+
            CompanyAddress1
            CompanyAddress2
            CompanyAddress3
            CompanyAddress4
```

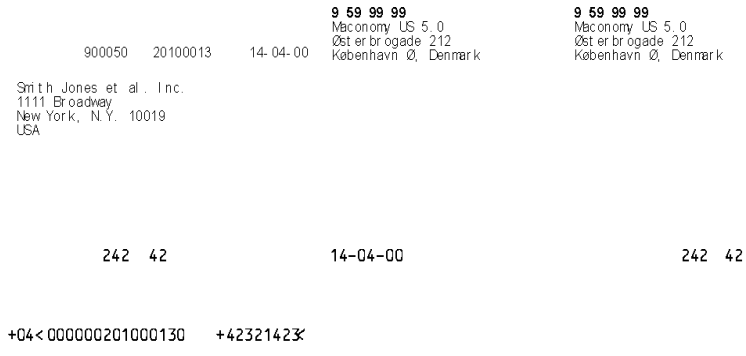
```
            CompanyAddress5
    <end stack>
    DollarAmount:(10.5mm,72mm):width=28.5mm:right
    CentAmount:(44mm,72mm):width=10mm:left
    .DueDate:(90mm,72mm):width=25mm:left
    -- The Receipt
    <stack (153mm,17mm)> -- the box of (148mm, 14.81mm)
           <default tag=var attribute=fontsize value=9>
           <default tag=var attribute=fontname value="Courier">
           <default tag=var attribute=width value=40mm> CompanyGiroNumber:bold+
           CompanyAddress1 CompanyAddress2 CompanyAddress3 CompanyAddress4
           CompanyAddress5
    <end stack> DollarAmount:(158.5mm,72mm):width=28.5mm:right
    CentAmount:(191.5mm,72mm):width=10mm:left "+04<":(7.62mm,90mm)
    .PayerIdentification:(17.78mm,90mm):width=5cm:left "+":(60.96mm,90mm)
    ExtAccountAccountNumberVar:(63.5mm,90mm):width=2cm:left "<":(81.28mm,90mm)
    <end canvas>
<end stack>
```

First a border is placed 101.6mm (the height of a giro form) from the bottom. This ensures that no elements before the border are printed on the preprinted giro form. Furthermore, it ensures that the stack that follows the border starts at exactly 101.6mm from the bottom of the paper.

The stack that follows the border has only one function: It ensures that the scope of the definitions in the stack is limited to the giro form. These definitions consist of a number of default declarations setting the standard font to “OCR-B” 14pt.

The formatting of the elements on the giro form is made by a canvas that places the individual texts and amounts correctly on the preprinted giro form. Note that the payment field and the payment receiver are stacks for which the content is simply stacked, but for which the stack itself is placed using coordinates in the canvas.

The result is as follows (slightly reduced in size):



Tips and Tricks

This section contains useful hints that can help you construct better and more elegant layouts using MPL.

Some of the most widely used layouts can initially seem difficult to construct using MPL. This section provides you with a number of tips and tricks and also gives you the necessary inspiration to get the most out of using MPL.

A layout can often be achieved using MPL—how you prefer to use MPL is naturally a matter of personal taste. But some methods can help to ensure that your MPL layouts are more easily maintained and updated.

Overlapping Fields

You often come across lines that need to include a debit field and a credit field. These lines can be constructed as follows:

```
{:[[stretch+][3cm][3cm]]
    .text
    .debit:zerosuppression+
    .credit:zerosuppression+;
}
```

Only one of the two fields can contain a value at a time, and you therefore waste space if you position them in separate columns. Instead you should let the two columns overlap each other.

Basically, this functionality is not supported by MPL, but if the two fields are placed in a canvas, they will overlap each other. You can then place the canvas in the same column:

```
{:[[stretch+][4cm]]
    .text
    <canvas height=10pt width=4cm>
    .debit:(0cm,0pt):width=3cm:zerosuppression+
    .credit:(1cm,0pt):width=3cm:zerosuppression+
    <end canvas>;
}
```

Paper Format Independence

Consider the following two MPL fragments:

```
{:[[75pt][stretch+][75pt]]
    .JobNumber JobName .ActivityNumber;
}
```

And:

```
{:[[75pt][380pt][75pt]]
    .JobNumber JobName .ActivityNumber;
}
```

These two definitions result in exactly the same formatting on a sheet of A4 paper with margins of 1 cm. Still, for several reasons, you should prefer the first definition:

- If you want to use the definition on another paper format (for example, landscape A4, US Letter), the first definition will automatically adjust to the paper format, whereas the other definition will have to be adapted, either because the entire sheet is not used, or because it has become too wide and cannot be compiled.
- If you change the margin in the definition or redefine the constant `InterColumnSpacing`, you will get an incorrect result.
- The first MPL fragment is a more logical representation than the second; that is, the length of `JobName` is unknown, and it should therefore have all excess space assigned to it, whereas the second fragment suggests that you should know for certain that 380 pt is an appropriate length for all occurrences of `JobName`.

Alignment of Headers and Footers

You often use headers in column headings and footers in totals. To align column headings with the contents, you can embed the repeating block in an array as in the following example:

```
{
   <repeating InvoiceLine script="InvoiceLineBlock">
   <header atstart+>
          "ITEM NO.""ORDERED"    "INVOICED"   "UNIT" "ITEM DESCRIPTION"  "UNIT
          PRICE"    "PRICE";
          <hline 7>;
   <end header>
   <footer>
          <hline 7>;
          ("Total Due":right):6   .TotalCurrency;
   <end footer>
   <conditional PricesOut script="PricesOut">
          .ItemNumber .NumberOrdered
   CorrectedNumberInvoiced .Unit
   ExternalItemText CorrectedUnitPrice
   CorrectedPriceWithoutDiscount;
   <end conditional>
   <end repeating>
}
```

You cannot use a similar technique for page headers and footers, since the `paper` tag cannot occur inside an array. If you want to align page headers or footers with the paper contents you must use rulers. An example of the use of this method can be seen in the following complete, however simple, layout for “Print Chart of Accounts”:

```
<mpl 2>
<layout title="Simple" print="Print_Chart_Of_Accounts"
   originallayout="Standard">
<page "A4">
<paper>
   <ruler main [[][stretch+][][15mm]]>
   <header onfirstpage+>
   {:main
          "ACCOUNT NO." "TEXT" "P&L, B/S" "TAX";
   }
```

```
    <end header>
    <repeating cursor=ACCOUNT>
    {:main
           .AccountNumber .AccountText
           .ProfitAndLossStatus .FinanceVATCode;
    }
    <end repeating>
<end paper>
```

Empty Stretchable Columns

In the footer of the time sheet print definition we used the `stretch` attribute to center the page number:

```
{:[[stretch+][][][][stretch+]]
    "" "-" PageNumber:center "-" "";
}
```

If you only want to insert a page number without hyphens on each page, `PageNumber:center` is of course sufficient, but by using a stretchable element you ensure that all three columns are centered equally. The purpose of this is to assign the necessary space to the two text blocks and the `PageNumber` variable, and all the remaining space is then equally distributed between the preceding and following columns.

In the example:

```
{:[[stretch+][][stretch+][][stretch+]]
    "" "a" "" "b" "";
}
```

we use the same method to ensure that “a” is positioned after a one-third indentation, and that “b” is positioned after a two-thirds indentation.

Stretchability and Embedding

If an array can be stretched—that is, at least one of its columns is stretchable—the array will stretch to the edges of the space that the surrounding elements provide. This is worth considering when arrays are contained in other columns. Columns in themselves are not stretchable, and the stretchability of the array will therefore not have any effect.

This point is illustrated in the following example:

```
<ruler main [[][]]>
{:main
    {:[[][stretch+]]
    "text";
    .field;
    }
    "another text";
}
```

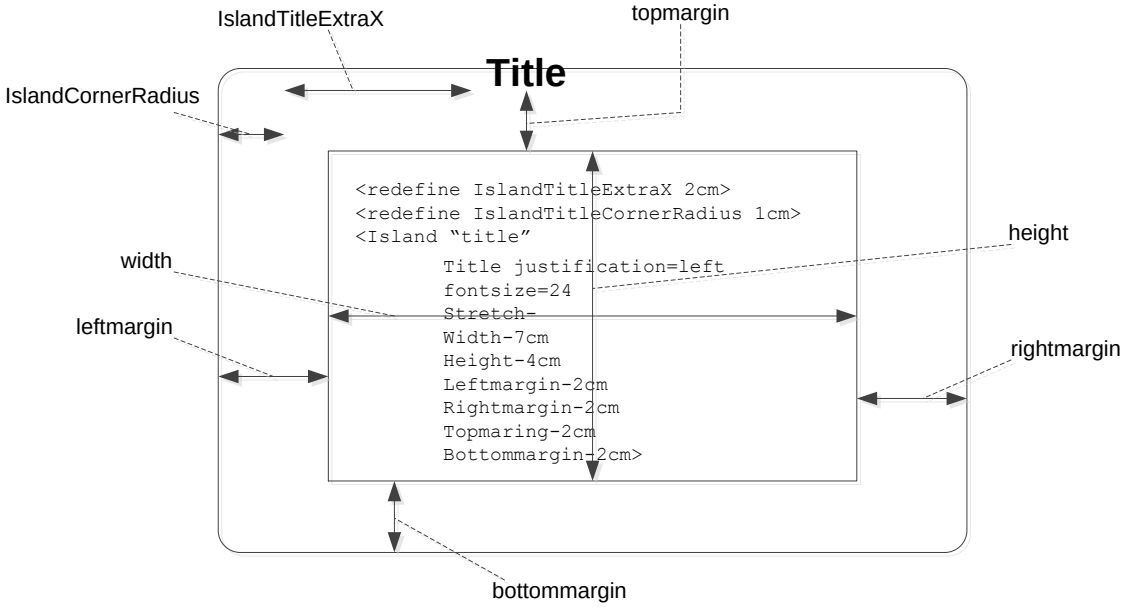
In this example, the array is contained in the first column of another array. Because the first column is not stretchable, the width of “text” plus the minimum width for fields is assigned to the inner array. If you change `main` to:

```
<ruler main [[stretch+][]]>
```

the column that contains the inner array will become stretchable. Therefore the column that contains `.field` can be stretched, and the field is assigned the width of the paper minus the width of “text” and “another text.”

Island Lengths

The formatting of an island is defined using a number of attributes and length constants. The following figure illustrates the functionality of the various lengths.



If an island cannot be stretched, it is assigned the greatest of the following values:

- 2 * IslandCornerRadius + 2 * IslandTitleIndent + the width of the title
- leftmargin + rightmargin + the width of the text (possibly as specified by the width attribute).

Fixed Frames and Watermarks

You cannot put repeating blocks inside islands. You can, however, put a fixed island frame around a repeating block by using the `movepos` attribute:

```
<ruler line [2pt[][][]]>
<repeating Employee script="L_EMPLOYEE">
   <header atstart+>
   <island height=4.5cm movepos->
   <end island>
   { :line
          "EMPLOYEE NO."   "NAME" "PHONE";
          <hline left- right->;
   }
   <end header>
   {:line
   .EmployeeNumber  .Name1 .Telephone;
   }
```

```
<end repeating>
```

In this example, the island frame is written in the header, but the print position is not moved. The header and content of the repeating block are thereby written inside the island frame. Note however that the frame is fixed, and that content from the repeating block therefore could exceed the frame, or text that follows the block could be written inside the block. This approach is therefore probably best suited for whole-page frames.

The same approach can be used for printing with background watermarks:

```
<ruler line [[3cm][stretch+][5cm fontsize=20]]>
<repeating Employee script="L_Employee">
   <header atstart+>
   <stack movepos->
          <image title="Images\MaconomyBackground.JPG" height=18cm>
   <end stack>
   <end header>
   {:line:keeptogether+
   .EmployeeNumber  .Name1 .Telephone;
   ""  .Name1 "";
   ""  .Name1 "";
   }
<end repeating>
```

Dynamic Frames

To put repeating blocks into a frame which expands with the size of the block, you can use a combination of the `hline` and `vline` tags and the `pagebottom` attribute:

```
<ruler line [||][stretch+][5cm]|]>
<redefine HeaderSkip 0pt>
<redefine FooterSkip 0pt>
<repeating Employee script="L_Employee">
   <header atstart+>
   {:line
          <skip 3mm>;
          <hline>;
          "EMPLOYEE NO."   "NAME" "PHONE";
          <hline left- right->;
   }
   <end header>
   <footer atend+ pagebottom->
   <hline>
   <end footer>
   {:line
   .EmployeeNumber  .Name1 .Telephone;
   }
<end repeating>
```

In this example, the first `hline` in the header creates the top of the frame, and the `vline` tags in the ruler create the sides of the frame. The bottom of the frame is created by the `hline` in the footer, and setting the `pagebottom` attribute to `false` in the footer ensures that its content

attaches itself to the frame. The two `redefine` tags are there to avoid space between header and vertical lines, respectively.

Using stop and goto

The `stop` attribute, together with the `goto` tag, enables you to write blocks on preprinted paper (for example, checks):

```
<repeating Employee script="L_Employee" stop=20cm>
   <header atstart+>
   ...
   <end header>
   <footer atend+ pagebottom->
   <hline>
   <end footer>
   {:line
   .EmployeeNumber  .Name1 .Telephone;
   }
<end repeating>
<goto bottom=10cm>
<hline>
"Check":color=red:italic+:fontsize=24:right
```

The `stop` attribute ensures that no content of the block is written below a line 20cm from the page bottom. After the loop has finished, the `goto` tag moves the printing position to a line 10cm from the page bottom.

Grammar

This section provides an easy-to-use overview of the syntax of MPL. The first section contains a definition of the language syntax in the form of a BNF (Backus-Naur Form) diagram, and the following section contains an overview of the MPL tags and their attributes.

Backus-Naur Form (BNF)

The syntax is described in a variant of the Backus-Naur Form. The notation (abbreviated BNF) is a precise method for describing valid language constructions.

Terminal symbols are written in the `typewriter font`, non-terminal symbols as regular text, and grammar primitives in *CAPITALS*.

Terminal symbols are characters that should be written as they appear. Non-terminal symbols are symbols that are defined by the grammar. Grammar primitives are similar to non-terminal symbols, but they are typically more fundamental and are defined less formally.

If a symbol *s* can be left out, it is written as s^0. If *s* can appear any number of times (0 or more), it is written as s*, and if *s* should only appear at least once (that is, once or more), it is written as s+. If either s_1 or s_2 is to appear, it is written as $s_1|s_2$.

In BNF, you would normally use the form [s], but because the square brackets occur in the grammar, that notation would make the grammar hard to read.

Syntax

The grammar primitives *ID*, *STRING*, *INTEGER*, *LENGTH*, and *BOOLEAN* are explained in “Attribute Values” in “Basic MPL.” *STRING2* is a string that is specified in apostrophes, for example, `'abc'`. *RGB* is a triple of integers denoting the percentage of the colors red, green, and blue, respectively, for example, (100,50,0).

Start	<mpl2>
	<`layout` Attributes>
	Page
	Margin^0
	Grid^0
	Frontpage^0
	Paper
Page	<`page` Attributes>
Margin	<`margins` Attributes>
Grid	<`grid` Attributes>

Frontpage <frontpage Attributes>

Definition*

Elem*

<end frontpage>

Paper : paper Attributes> Definition*

Elem+

<end paper>

| <paper Attributes> Definition* FrameOrg

<end paper>

Header : <header Attributes> Definition*

Elem*

<end header>

Footer : <footer Attributes> Definition*

Elem*

<end footer>

Canvas : <canvas Attributes> Definition*

Elem*

<end canvas>

Conditional : <conditional Attributes> Definition*

Elem*

<end conditional>

Repeating : <repeating Attributes> Definition*

Elem*

<end conditional>

Island : <island Attributes> Definition*

Elem*

<end island>

Stack : <stack Attributes> Definition*

Elem*

<end stack>

Span : <span Attributes> Definition*

Elem

<end span>

| (Definition* Elem) Shortattributes

Array : <array Attributes>

Elem+

<end array>

| { Shortattributes Elem+ }

Elem : Conditional

Paper : <paper Attributes> Definition*

Elem+

<end paper>

| <paper Attributes> Definition* FrameOrg

<end paper>

Header : <header Attributes> Definition*

Elem*

<end header>

Footer <footer Attributes> Definition*

Elem*

<end footer>

Canvas : <canvas Attributes> Definition*

Elem+

<end canvas>

Conditional : <`conditional` Attributes>

Definition*

Elem*

<end conditional>

Repeating : <`repeating` Attributes>

Definition*

Elem*

<end repeating>

Island : <`island` Attributes> Definition*

Elem*

<end island>

Stack : <`stack` Attributes> Definition*

Elem*

<end stack>

Span : <`span` Attributes> Definition*

Elem

<end span>

| (Definition* Elem) Shortattributes

Array : <`array` Attributes>

Elem+

<end array>

| { Shortattributes Elem+ }

Elem : Conditional

| Repeating

| Array

| Text

| Field

| Variable

| Island

| Stack

| Skip

| Canvas

| Line

| Border

| Newpage

| Header

| Footer

| Row

| Span

| Hline

| Vline

| Image

| Goto

| Title

Row : <row> Attributes>

Elem+

<end row>

| Elem+; Short attributes

Text : <text Attributes>

| *STRING* Short attributes

Text2 : <text2 Attributes>

| *STRING2* Short attributes

Field	:	<`field` Attributes>
	|	*ID*[0].*ID* Short attributes
Variable	:	<`var` Attributes>
	|	*ID* Short attributes
Title		<`title` Attributes>
	|	[*ID*] Short attributes
	|	*ID*[0].*ID* Short attributes
Line	:	<`line` Attributes>
Border	:	<`border` Attributes>
Newpage	:	<`newpage` Attributes>
Hline	:	<`hline` Attributes>
Vline	:	<`vline` Attributes>
	|	| Short attributes
Skip	:	<`skip` Attributes>
Image	:	<`image` Attributes>
Goto	:	<`goto` Attributes>
Definition	:	defaultdef
	|	RulerDef
	|	Setlength
Setlength	:	<`define` Attributes>
	|	<`redefine` Attributes>
Defaultdef	:	<`default` Attributes>
RulerDef	:	<`ruler` Attributes>
	|	<`subruler` Attributes>
Attributes	:	Attribute*
Shortattributes	:	(:Attribute)*

Attribute : *ID* = Attributevalue

| *ID*+

| *ID*-

| Attributevalue

Attribute value : BOOLEAN

| INTEGER

| STRING

| *ID*

| *RGB*

| Length

| (Length, Length)

| List

Paper : `<paper` Attributes>

Definition*

Elem+

<end paper>

Header : `<header` Attributes>

Definition*

Elem*

<end header>

Footer : `<footer` Attributes>

Definition*

Elem*

<end footer>

Canvas : `<canvas` Attributes>

Definition*

Elem+

<end canvas>

Conditional : `<conditional` Attributes>

Definition*

Elem*

<end conditional>

Repeating : `<repeating` Attributes>

Definition*

Elem*

<end repeating>

Island : `<island` Attributes>

Definition*

Elem*

<end island>

Stack : `<stack` Attributes>

Definition*

Elem*

<end stack>

Span : `<span` Attributes>

Definition*

Elem

<end span>

Array : `<array` Attributes>

Elem+

<end array>

| { Shortattributes Elem+ }

Elem : Conditional

| Repeating

| Array

| Text

| Field

| Variable

| Island

| Stack

| Skip

| Canvas

| Line

| Border

| Newpage

| Header

| Footer

| Row

| Span

| Hline

| Vline

| Image

| Goto

| Title

Row : `<row` Attributes>

Elem+

<end row>

| Elem+; Short attributes

Text : `<text` Attributes>

	|	*STRING* Short attributes
Text2	:	`<text2` Attributes>
	|	*STRING2* Short attributes
Field	:	`<field` Attributes>
	|	*ID⁰.ID* Short attributes
Variable	:	`<var` Attributes>
	|	*ID* Short attributes
Title	:	`<title` Attributes>
	|	[*ID*] Short attributes
	|	[*ID⁰ .ID*] Short attributes
Line	:	`<line` Attributes>
Border	:	`<border` Attributes>
Newpage	:	`<newpage` Attributes>
Hline	:	`<hline` Attributes>
Vline	:	`<vline` Attributes>
	|	| Short attributes
Skip	:	`<skip` Attributes>
Image	:	`<image` Attributes>
Goto	:	`<goto` Attributes>
Definition	:	Defaultdef
	|	RulerDef
	|	Setlength
Setlength		`<define` Attributes
	|	`<redefine` Attributes>
Defaultdef	:	`<default` Attributes>
RulerDef	:	`<ruler` Attributes>

	|	`<subruler` Attributes>
Attributes	:	Attribute*
Shortattributes	:	(:Attribute)*
Attribute	:	*ID* = Attributevalue
	|	*ID*+
	|	*ID*-
	|	Attributevalue
Attribute value	:	BOOLEAN
	|	INTEGER
	|	STRING
	|	*ID*
	|	*RGB*
	|	Length
	|	(Length, Length)
	|	List
	|	Ruler
	|	SubRuler
Length	:	LENGTH
	|	*ID*
Ruler	:	[(ColSep^0 Col)+ ColSep^0]
Col	:	[Attributes]
ColSep	:	LENGTH
	|	Vline
	|	Text
SubRuler	:	[Col+]
List	:	[*ID* (, *ID*)*]

Attribute List

The following table is a list of tags that are used in MPL and the attributes that belong to each tag. To improve readability, some attributes, which are shared between many tags, have been assembled into attribute groups.

The first column displays the tag name, and the second column displays attributes or groups of attributes that you can assign to the tag in question. Attribute group names are written in *italics*. If an attribute group is available in a tag, it means that all of the attributes that belong to that group are available in the tag. At the end of this section is a table of the attributes that are available within each attribute group.

The third, fourth, and fifth columns contain information about the use of the current tag: “M” stands for mandatory, “N” stands for nameless, and “S” specifies that the tag is nameless in a possible short form. The sixth column displays the attribute value type.

Tag	Attribute	M	N	S	Type
array	ruler pos indent height width baseline	 M	N N	S S	*ID >> Ruler* *POS* *LENGTH* *LENGTH* *LENGTH* *ID*
assign *(MPL 4)*	var value	M M	N N		*ID* *EXPRESSION*
border	top bottom *style*				*LENGTH* *LENGTH*
canvas	pos indent height width *style* *block*	M M M	N		*POS* *LENGTH* *LENGTH*
concat	pos indent width justification style *wrap*		N N	 S	*POS* *LENGTH* *LENGTH* *ID*
conditional	variable script indent height negate width baseline *style* *block*	M	N N		*ID* *STRING* *LENGTH* *LENGTH* *BOOLEAN* *LENGTH* *ID*

Tag	Attribute	M	N	S	Type
	expression *(MPL 4)*		N		*EXPRESSION*
cursor *(MPL 4)*	query name showmaincursor	M	N		*ID* *ID* *BOOLEAN*
default	attribute value tag	M M M			*ID* *Any*
define	name value orientation	M M	N N		*ID* *LENGTH* *ID*
eval *(MPL 4)*	expression zerosuppression pos indent width justification *style* *wrap*	M M	N N	 S	*EXPRESSION* *BOOLEAN* *POS* *LENGTH* *LENGTH* *ID*
field *(desupported in MPL 4)*	data cursor zerosuppression pos indent width	M M	N N	 S	*ID* *ID* *BOOLEAN* *POS* *LENGTH* *LENGTH*
footer	onlastpage attend script height *style*		 N		*BOOLEAN* *BOOLEAN* *STRING* *LENGTH*
frame	height width *style*				
framecolumn	style				
framerow	style				
grid	hor ver pagebottom	M M			*LENGTH* *LENGTH* *BOOLEAN*
goto	top bottom				*LENGTH* *LENGTH*

Tag	Attribute	M	N	S	Type
header	onfirstpage atstart script height *style*		 N N		*BOOLEAN* *BOOLEAN* *STRING* *LENGTH*
hline	multi columns left right *style*	 M 	 N 		*INTEGER* *INTEGER* *BOOLEAN* *BOOLEAN*
image	title fieldname varname cursorname justification width height scalewidth scaleheight expression *(MPL 4)*		N N N		*STRING* *ID* *ID* *ID* *ID* *LENGTH* *LENGTH* *BOOLEAN* *EXPRESSION*
island	title titlejustification titlecolor titlergb rounded pos indent height width fontname fontsize stretch italic underline bold topmargin bottommargin leftmargin rightmargin justification baseline *block*	 M 	N N 		*STRING* *ID* *ID* *RGB* *BOOLEAN* *POS* *LENGTH* *LENGTH* *LENGTH* *STRING* *INTEGER* *BOOLEAN* *BOOLEAN* *BOOLEAN* *BOOLEAN* *LENGTH* *LENGTH* *LENGTH* *LENGTH* *ID* *ID*
layout	title print originallayout	M M M	N		*STRING* *STRING* *STRING*
line	start end	M			*POS*

Tag	Attribute	M	N	S	Type
	style	M			*POS*
margins	top bottom left right	M M M M			*LENGTH* *LENGTH* *LENGTH* *LENGTH*
page	name orientation	M	N N		*STRING* *ID*
paper	cursor script *style*		N N		*ID* *STRING*
parameter	name vale	M M	N N		*ID* *EXPRESSION*
query	name	M	N		*ID*
redefine	name value orientation	M M	N N		*ID* *LENGTH* *ID*
repeating	cursor script groupby indent height width *style* *block*	M	N N N		*ID* *STRING* *LIST* *LENGTH* *LENGTH* *LENGTH*
row	align height *style* *block*		N	S	*ID* *LENGTH*
ruler	name value	M M	N N		*ID* *ID >> Ruler*
skip	height	M	N		*LENGTH*
span	columns	M	N	S	*LENGTH*
stack	indent pos script height width baseline *style*	M	N N		*LENGTH* *POS* *STRING* *LENGTH* *LENGTH* *ID*

Tag	Attribute	M	N	S	Type
	block				
subrulers	name value parent	M M M	N N		*ID* *SUBRULER* *ID*
text	title pos indent width justification fontname fontsize bold italic underline	M M	N N N	 S S	*STRING* *POS* *LENGTH* *LENGTH* *ID* *STRING* *INTEGER* *BOOLEAN* *BOOLEAN* *BOOLEAN*
text2	same attributes as text				
val *(MPL 4)*	name value type	M M	N N		*ID* *EXPRESSION* *ID*
var *(desupported in MPL 4)*	data zerosuppression pos indent width justification *style*	M M	N N	S S	*ID* *BOOLEAN* *POS* *LENGTH* *LENGTH* *ID*
var *(MPL 4)*	name value type	M M	N N		*ID* *EXPRESSION* *ID*
vline	justification style		N	S	*ID*
Col	stretch width style		 N	 S	*BOOLEAN* *LENGTH*
ColSpec	interval	M	N	S	*INTERVAL >>* *INTEGER*
title	fieldname varname cursorname				*ID* *ID* *ID*

Tag	Attribute	M	N	S	Type
	uppercase title width *style*			N	*STRING* *LENGTH*

Attribute Group List

The following is a table of attribute groups that can be used for several tags, namely the ones where the group name in question appears in italics in the Attribute column in the attribute list. In this table, the first column displays the attribute group name. The Attribute column shows the attributes that are available in tags where the attribute group in question is available.

Group	Attribute	M	N	S	Type
style	justification[2] fontname fontsize bold italic underline color rgb dateformat[3] timeformat amountformat realformat integerformat booleanformat		N	S	*ID* *STRING* *INTEGER* *BOOLEAN* *BOOLEAN* *BOOLEAN* *ID* *RGB* *STRING* *STRING* *STRING* *STRING* *STRING* *STRING*
block	keeptogether movepos stop				*BOOLEAN* *BOOLEAN* *LENGTH*
wrap	wrap lines height				*BOOLEAN* *INTEGER* *LENGHT*

MPL for Universe Reports

This section describes a number of MPL extensions that have been created to support Maconomy Universe Reports. Universe Reports are used in the Maconomy Portal to create dynamic, interlinked reporting tools using HTML or PDF to display output. The layout of such reports is based on MPL. However, you cannot use all MPL commands in Universe Reports. For a list of the MPL features that are not available in Universe Reports, see “Standard MPL vs. Reporting MPL.” Conversely, you cannot use the MPL extensions for Universe Reports in standard MPL used for the layout of printed reports, unless specifically stated.

[2] This attribute is nameless for the long form of the tag text, and the short form of the tags text, field, and var.

[3] Format attributes are used in MPL for Universe Reporting. See “Formats.”

A Universe Report is generated using MQL statements. The Maconomy Query Language (MQL) is a language for interacting with the Maconomy database. In structure, it is similar to SQL, but there are a number of differences.

- With MQL, you do not have to be aware of the relation structure of the database.
- The data model is separated from the command—in MQL, the data model is defined in universes.
- MQL is aware of the Maconomy types, and the types are validated before command execution.
- MQL is database-independent.

In MQL, data is selected using an `mselect` statement. For more information, see the "MQL Language Reference."

The extensions can be divided into three categories: links, tables, and charts. The following sections describe these extensions.

Links

Using the parenthetical tag `link`, you can add a hypertext link in the report. The syntax of links is the following:

```
<link attributes>
    ...
<end link>
```

The hyperlink that is specified in the attributes can be a regular URL, possibly with parameters; it can be a JavaScript; or it can be a link to a Portal component or another Universe report. The `link` tag has the following attributes:

- `href` specifies the address (URL) of the link. The attribute has the type *STRING*. If the tag is specified, the link opens a browser window to the specified URL. If a `parameters` attribute is specified, the parameters are added to the URL in the form described below.
- `script` specifies a piece of JavaScript code. The attribute has the type *STRING*. If the tag is specified, the specified script is executed when the link is clicked. You cannot combine this attribute with other attributes. For example:

```
field2:script="var x='Hello World';alert(x);"
```

- `component` specifies the name of a Portal component that will be opened when the link is clicked. The attribute has the type *STRING*. This attribute is usually used in conjunction with the `parameters` attribute (see below).
- `report` specifies the name of another report that will be called when the link is clicked. This attribute is usually used in conjunction with the `reportsetup` and `parameters` attributes (see below).
- `reportsetup` specifies how a linked report should be called. This attribute currently takes one parameter of the type *PARAMLIST*, namely `rpAction`. If the `reportsetup` attribute is not specified, the linked report will open with an empty selection criteria selection window. This attribute only has effect when used in conjunction with `report`. See the description of `parameters` below for an example.

  This parameter takes the following values:

  - `drill`: If this value is specified, the report will take the parameters specified through the `parameters` attribute and use them in the selection criteria specification window, if possible, and open the report directly, if possible (that is, if all mandatory selection fields

are completed). Furthermore, it will be possible to return to the original report (a drill-down path is displayed).

- `drillaround`: This value essentially does the same as `drill`, except that the old report is not entered in the drill-down path, but replaced by the new report. This is useful if two different reports present the same data in different ways and therefor exist at the same level in the drill-down hierarchy.
- `edit`: This value forces the user to specify selection criteria by opening the selection criteria specification window before drilling down.

- `parameters` specifies parameters that are to be added to the URL specified in the `href`, `component`, or `report` attributes. The attribute has the type *PARAMLIST*. The parameters can be the content of fields in the underlying MQL `mselect` statement, variables, or text. When this attribute is specified, the parameters are added to the URL (as specified in the `href`, `report`, or `component` attribute) in the following form:

```
<URL>?param1=value1&param2=value2
```

Example of the use of `parameters` in conjunction with `report` as a link from a table field:

```
.EmployeeNumber
        :report="Demo::ByJob"
        :parameters=
            [
                ParEmployeeNumber=.EmployeeNumber
            ]
:reportsetup=[rpAction="drill"]
```

- `transferparameters`: This attribute is of type *BOOLEAN*. If this attribute is set to `true`, all MRL parameters from the current report are automatically transferred as parameters in the link to the next report. This is useful if universe reports that consist of several MRL definitions share parameters. The attribute only has effect for report links.
- `target` specifies the target window of the link. The attribute is of type ID, and possible values are `self` (replace the current window or frame), `new` (show in a new browser window), and `rightside` (show to the right of the Portal). The attribute is ignored for report links and has no effect for script links. For `component` and `href` links, the default is `rightside`.

Note that the link attributes can be added to a number of different MPL tags. Apart from the `link` tag, the following tags accept the link attributes. Links are of course only relevant in reports in HTML or PDF formats:

- `text` (see “Texts”)
- `field` (see “Fields”)
- `var` (see “Variables”)
- `text2` (see “Alternative text tag”)
- `image` (see “Images”)
- `title` (see “Title”)
- `fields` (see “Tables”)

Tables

Tables are a very common feature in reports, and the generation of tables has been simplified with the MPL `table` tag. This tag is a form of preformatted template, where an HTML or PDF table is generated by iterating through a cursor. This section describes:

- Syntax of the table tag
- Layout of individual fields
- Attributes applicable to fields tag
- Attributes applicable to individual fields
- Table headers and footers
- Example

Syntax of the Table Tag

The general syntax of tables is the following:

```
<table cursor=cursorName>
   <fields>
   .field1
   .field2
         ...
   .fieldN
   <end fields>
<end table>
```

The attribute `cursor` is mandatory and should be the name of the cursor that contains the `mselect` result columns to be shown in the table. The cursor is defined in MQL using the `as cursor` keyword.

You can only use the `table` tag in stacking environments; that is, you cannot place a table in an array row or a canvas.

If you want to show all of the fields that are contained in the current cursor, the following short-hand is available:

```
<table cursor=cursorName>
   <fields>
   *
   <end fields>
<end table>
```

The order of the fields in the table is determined by the way in which the underlying `mselect` statement chose to sort the current cursor.

You can exclude fields or apply various formatting by combining the asterisk `(*)` notation with explicit field references (the `table` tag is left out from the examples from now on):

```
<fields>
   *
   .field5:hidden+
   .field6:color=red
<end fields>
```

This will display all fields with the exception of `field5`. Furthermore, `field6` is colored red.

There can be a maximum of one asterisk in a `fields` section. All fields in the current cursor that have not been explicitly referenced anywhere in the `fields` section will be placed in the location of the asterisk, in the order in which they appear in the cursor.

You can nest tables. If the underlying `mselect` statement has defined two cursors, you can add a new `table` tag within the first `table` tag using the new cursor name like this:

```
<table cursor=cursorName1>
   <fields>
   .field1
   .field2
   <end fields>
   <table cursor=cursorName2>
   <fields>
          .field1
          .field2
             ...
          .fieldN
   <end fields>
   <end table>
<end table>
```

This enables you to group entries within a table. For an example of this, see “Example” below.

Layout of Individual Fields

You can hide or apply special formatting to individual fields by applying attributes directly to the fields, for example:

```
<fields>
   .field1:bold+
   .field2:bold+
   .field3:hidden+
   *
   .field6:color=red
<end fields>
```

The following attributes can be applied to individual fields:

- href, script, parameters, component, report

  These tags allow you to add links to the fields in the table. For more information, see “Links.”

- Attributes in the attribute group “style”

  For more information, see “Attribute Group List.”

Note that you can place other elements than fields in the `fields` section of the `table` tag. Texts, titles, images, and stacks can also be inserted in the report columns. When using a stack, the stack can only contain elements of the same type, for example, a stack of images. For more information, see “Texts,” “Title,” “Images,” and “Stacks.” For instance, you can use a stack like this:

```
<fields>
```

```
    .field1:bold+
    .field2:bold+
    <stack>
    .field3
    .field4
    .field5
    <end stack>
    .field6:color=red
<end fields>
```

This will place the contents of fields 3 to 5 in the same table cell, written above one another.

Note that both the long form and the short form can be used when specifying tags. In the examples in this section, only the short forms are used.

Attributes Applicable to Fields Tag

Instead of applying attributes to the individual fields, you can add attributes to the `fields` tag. Attributes specified for individual fields will then overwrite the attributes specified in the `fields` tag. In the following example, all fields except `field6` are colored blue:

```
<fields:color=blue>
    *
    .field6:color=red
<end fields>
```

The attributes which you can use are the same as those mentioned above for the individual fields, with the exception of `hidden`. See also “Style Inheritance.”

Formats

The output formats for fields (and variables) are by default the formats that are set up by the client that is connected to the server. For Universe Reports, the client is the web server, so the format settings are defined by the M-Script format settings. You can change these formats locally in a report by using these format attributes:

- dateformat
- timeformat
- amountformat
- realformat
- integerformat
- booleanformat

These attributes are all of type *STRING*, and their values should be a (valid) format string for the type that they represent. See the M-Script manual for a description of format strings.

The format attributes are part of the *style* attributes (see “Attribute Group List”) and can therefore be added to any tag in MPL. They are inherited in the same way as *style* attributes.

Example

```
<fields dateformat="E:DDMMYY">
    .dateField1
    .dateField2:dateformat="E:YYDDMM"
```

```
<end fields>
```

Here `dateField1` will be written as "DD:MM:YY," whereas `dateField2` will be written as "YY:DD:MM".

As a special case, specifying the empty string as the value for a format attribute is interpreted as the default format attribute value.

Example

```
<fields dateformat="E:DDMMYY">
    .dateField1
    .dateField2:dateformat=""
<end fields>
```

Here `dateField1` will be written as "DD:MM:YY," whereas `dateField2` will be written in the default format.

A warning is issued if you specify an invalid format value or if you specify a format attribute on a field (or variable), where the type of the field does not match the type of the attribute (for example, a date format attribute on a time field). In both cases, the format attribute that caused the error is ignored.

Attributes Applicable to Individual Fields

All of the attributes mentioned in “Attributes Applicable to fields Tag” and “Formats” can be applied to individual fields as well. Apart from those, the following attributes can be applied to individual fields:

- headertitle

  Using this attribute, you can assign your own header to the column represented by the field. For instance, if you want to change the default header “Description” for the field `.Activity`, you can write the following:

```
.Activity:headertitle="Act. Name"
```

  This example will output the header “Act. Name” above the “Activity” column. The alternative to using the `headertitle` attribute is to specify a field in a separate header (see “Table Headers and Footers”). A separate header is, for example, useful if you want to add attributes to the header text. If, for instance, you add:

```
:color=redto the example above, the field will be colored red, not the header text.
```

- footerfunction

  Using this attribute, you can replace the default footer function with another function specified in the MQL `aggregate` option. The default function is `SUM`. If the specified function is not defined in the underlying `mselect` statement, no footer is displayed. For more information, see the MQL Reference manual. For example:

```
.RegTime:footerfunction="Max"
```

  This example will output the maximum number of registered hours below the “RegTime” column. The alternative to using the `footerfunction` attribute is to specify a field in a separate footer (see “Table Headers and Footers”).

- hidden

  Using this attribute, you can hide the current field. See “Syntax of the Table Tag” for an example of this.

- tooltip

  Using this attribute, you can add a tooltip text to the current field. The type of this attribute is *STRING*. The tooltip is naturally only displayed in HTML output, even though you can assign it in MPL layouts for printed output. For example:

```
.field2:tooltip="Click here to open the MyCustomer component showing the
current customer."
```

Table Headers and Footers

Headers and footers are enabled by default in tables. If you do not want a header or footer in a table or nested table, add the attribute `header-` and/or `footer-` to the `table` tag:

```
<table cursor=cursorName header- footer- >
```

However, you can also add your own headers and footers to the table using the parenthetical tags `header` and `footer`. The following example shows the syntax:

```
<header>
   "Title for field1"
   "Title for field2"
   *
   "Title for fieldN"
<end header>
<footer> "Total Time"
   *
   .RegTime$Sum:color=red
<end footer>
```

In the footer example, the field `.RegTime$Sum` is the result of the underlying MQL creating a sum field. Any function that is used in a footer must be made available by the underlying MQL statements (see also the description of the attribute `footerfunction` above).

For nested tables, headers can only be specified for the outermost table. Footers, however, can (must) be specified for each table. You can add multiple headers to a table by repeating the `header` tag. Note also that the use of the tags `header` and `footer` overwrites the specifications in the `headertitle` and `footerfunction` attributes.

You can make headings in a header span multiple columns using the same technique as for columns in regular MPL (see “Spanning Columns”). For example:

```
<header>
   "Title for field1"
   ("Title for field2 and field3"):2
<end header>
```

If a span exceeds the number of available columns, the span is automatically reduced to match the number of available columns.

Furthermore, you can use fields, titles, and images in headers and footers. For example:

```
<header>
   "Employee Name"
   *
   <image title="MyImages\regtime.png" scaleheight- height=100pt>
<end header>
```

You can add more headers to a table, simply by specifying multiple `header` tags. If you want to use the title of a field as a header, you can reference the field's title using the common MPL syntax: `[.field]`. You can also reference fields that are beyond the scope of the current cursor by qualifying the name with the full cursor path to the desired field, relative to the context that you are in. For example, if the outer table uses cursor `EmployeeCursor`, but you really want the title of a field in the `embedded cursor ActivityCursor`, you can use the following path within a `header` tag:

```
[EmployeeCursor.ActivityCursor.Activity]
```

Example

Below is a basic example of the use of the `table` tag. This example will be expanded as all of the features of this tag are explained.

```
<table cursor=EmployeeCursor>
   <fields>
   .EmployeeName
   .Activity
   .RegTime
   <end fields>
<end table>
```

This example will yield a simple list of employee registrations:

Employee Name	Description	Registered Time
Joe Daniels	Carpentry	20.0
Jack Johnson	Photo Copies	26.0
Jack Johnson	Consulting	25.0
Jack Johnson	Photo Copies	4.0
Sally Rogers	Photo Copies	5.0
		80.0

Now we want to avoid the repetition of the employee name, and we want different titles for our header. The titles are by default the visible title (label) of the referenced field. Fortunately, our `mselect` statement has defined another cursor as well, so we use a nested table with the cursor `ActivityCursor`:

```
<table cursor=EmployeeCursor>
<header>
   "Employee Name"
   "Act. Name"
   "Registered Time"
<end header>
   <fields>
   .EmployeeName
   <end fields>
```

```
    <table cursor=ActivityCursor>
    <fields>
           .Activity
           .RegTime
    <end fields>
    <end table>
<end table>
```

Employee Name	Description	Registered Time
Joe Daniels	Carpentry	20.0
		20.0
Jack Johnson	Photo Copies	26.0
	Consulting	25.0
	Photo Copies	4.0
		55.0
Sally Rogers	Photo Copies	5.0
		5.0
		80.0

To remove the subtotals from the table and leave only the grand total, add the attribute `footer-` to the inner table. Also add a bolded footer text to the first column:

```
<table cursor=EmployeeCursor>
<header>
    "Employee Name"
    "Act. Name"
    "Registered Time"
<end header>
<footer>
    "Total Time":bold+
<end footer>
    <fields>
    .EmployeeName
    <end fields>
    <table cursor=ActivityCursor footer->
    <fields>
           .Activity
           .RegTime
    <end fields>
    <end table>
<end table>
```

Employee Name	Description	Registered Time
Joe Daniels	Carpentry	20.0
Jack Johnson	Photo Copies	26.0
	Consulting	25.0
	Photo Copies	4.0
Sally Rogers	Photo Copies	5.0
		80.0

If you want to consolidate the activities (for example, to avoid multiple entries for “Photo Copies” for the same employee), an additional cursor is needed in the underlying `mselect` statement, which could then be incorporated in another nested table in MPL.

Charts

Charts are diagrams which depict report data graphically. Currently, MPL supports pie charts and bar charts.

Charts are currently considered one entity. This means that they cannot be broken into several pages in a PDF. If a chart does not fit on a page, the excess part is cut off. This is similar to specifying an image that is too large for one page in regular MPL. However, this restriction will be resolved in the near future.

Pie Charts

The syntax of a pie chart is as follows:

```
<piechart cursor=cursorName height=xxxpt>
   <legend>
   .field3
   <end legend>
   <fields>
   .field1
   .field2
   <end fields>
<end piechart>
```

This will draw two pie charts for `field1` and `field2`, respectively. The default title of the field is printed above each chart.

The pie chart definition is similar to the table definition, with the `legend` tag taking the place of the `header` tag. Therefore, the description of the pie chart will focus on where the specification is different from the table definition.

The `cursor` attribute is not mandatory, but is usually assigned a value. See “Cursorless Charts.” The cursor contains the columns available in the result of the underlying `mselect` statement. The attribute `height` is mandatory. The `height` attribute denotes the height of the chart excluding any legend.

The `legend` tag prints a legend that consists of colored boxes and the default text of the referenced field. You can in principle add any MPL inside the `legend` tag, but you should keep the legend simple. For instance, you can change the text that is displayed using the attribute `headertitle` (see the description in "Attributes Applicable to Individual Fields").

Also note in this connection that the `fields` section is not mandatory. This enables you to print the chart legend separately from the actual chart, even in another frame (see the example in "Frames").

The titles of the referenced fields are shown in the pie chart. You can add links to the fields, and you can add style information similar to ordinary table fields. The style is reflected in the title of the fields.

Note that the use of the `*` wildcard in the list of fields in the `fields` tag is not supported. The wildcard is ignored.

Bar Charts

The bar chart definition is very similar to that of the pie chart:

```
<barchart cursor=cursorName height=xxxpt>
   <legend>
   .field3
   <end legend>
   <fields>
   .field1
   <end fields>
<end barchart>
```

The `height` attribute specifies the height of the y axis of the chart.

You can use bar charts in two ways. In the preceding example, the title of the field that is referenced in the `legend` tag is printed along with a colored box below the x axis as a caption for the chart. The individual values of `field1` are used as bars in the chart.

If you add more fields in the `fields` section of the bar chart definition, a bar will be shown with the sum of the referenced columns—that is, a bar for each referenced field. Each bar will have a different color, which is also reflected in the bar chart legend.

The bar chart has two additional, optional attributes:

- `interbarspace`: This attribute is of the type *LENGTH*. If you, for example, assign a value of `2pt` to this attribute, the individual bars in the chart will be spaced 2 points apart. For example:

  ```
  <barchart cursor=EmployeeCursor height=100pt interbarspace=2pt>
  ```

- `maxbarwidth`: This attribute is of the type *LENGTH*. The bar width is by default calculated by dividing the width of the bar chart by the number of bars (and any space between). However, when the number of bars is low, this can produce very wide bars. To avoid this, MPL operates with a default maximum bar width of 20pt. You can change the maximum bar width by adding the attribute `maxbarwidth` to the `barchart` tag. For example:

  ```
  <barchart cursor=EmployeeCursor height=100pt interbarspace=2pt
  maxbarwidth=16pt>
  ```

Cursorless Charts

Even though charts are usually based on cursors, you can create them without using a cursor from an `mselect` statement, simply by omitting the `cursor` attribute from the `pie chart` or

`barchart` tags. When no cursor is used, the chart picks its data from the currently active row in the `MSelect` statement.

When you specify a chart without a cursor, the `headertitle` attribute and the `legend` tag change their meaning: `headertitle` (or, if omitted, the default title) will function as the legend beneath the x axis in bar charts, and the legend is used as a title and is printed above the chart.

Frames

In the Portal, *dashboards* are used for organizing multiple Portal components and reports into one view. Using MPL, you can show the top-level layout of Universe Reports in a dashboard-style layout using *frames*. This can, for example, be used for combining multiple reports in one view.

A frame has a fixed size. If the size is insufficient to hold the content of the frame, a scrollbar will appear in the HTML output from the report. In PDF output, overflow is clipped, and little red arrows will indicate that clipping has been performed.

You can only specify framesets (the sequence of frames) at the top level, in the `paper` tag. Dashboard-style framesets are organized using `framerow` and `framecolumn` tags, which can be nested arbitrarily. The `frame` tags are placed inside the frame rows and columns.

All the frame-related tags can take the *style* attributes (which will affect the content). For more information, see “Attribute Group List.”

framerow

A frame row is created using:

```
<framerow attributes>
    ...
<end framerow>
```

Frame rows can be nested in other frame rows or columns. Note that frames defined within a `framecolumn` without a surrounding `framerow` tag will be organized in rows.

framecolumn

A dashboard column is created using:

```
<framecolumn attributes>
    ...
<end framecolumn>
```

Frame columns can be nested in other frame columns or rows. Note that frames defined within a `framerow` without a surrounding `framecolumn` tag will be organized in columns.

frame

The frame itself is specified using the parenthetical `frame` tag:

```
<frame attributes>
    ...
<end frame>
```

The behavior of a frame can be modified using the following attributes:

- `height` is used to specify the height of the frame’s contents. If this attribute is not specified, the contents of the frame will determine its height. The attribute has the type *LENGTH*.

- `width` is used to specify the width of the frame's contents. If this attribute is not specified, the contents of the frame will determine its width. The attribute has the type *LENGTH*.

The *`framedefinition`* is any content—for example, text, a pie chart, or a table.

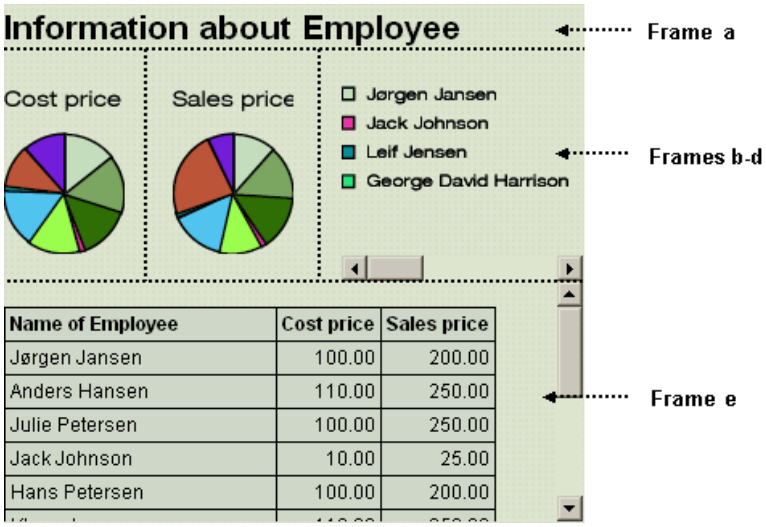
Example

The following is an example of a dashboard that is created by using MPL (annotations are described below):

```
<mpl 2>
<layout>
<paper>
   1   <framecolumn>
   2   <frame>
   "Information about Employee":fontsize=16:bold+
   <end frame>
   3   <framerow>
   4  <frame width=70pt>
          <piechart Employeecursor height=50pt >
             <fields>
                .costprice:headertitle="Cost price"
             <end fields>
          <end piechart>
   <end frame>
   <frame width=70pt>
          <piechart Employeecursor height=50pt >
             <fields>
                .salesprice:headertitle="Sales price"
             <end fields>
          <end piechart>
   <end frame>
   <frame width=100pt fontsize=7>
          <piechart Employeecursor height=70pt>
             <legend>
                .name1
             <end legend>
          <end piechart>
   <end frame>
   <end framerow>
   5  <frame height=100pt width=240pt>
          <table Employeecursor>
             <fields>
                .name1:headertitle="Name of Employee"
                .costprice:headertitle="Cost price"
                .salesprice:headertitle="Sales price"
             <end fields>
          <end table>
   <end frame>
   6  <end framecolumn>
```

```
<end paper>
```

This layout yields the following result, where the defined frames are marked:



Annotations

1. First, a frame column is defined.
2. In the column, frame *a* is inserted (the heading text).
3. Then a frame row within the frame column is defined, containing three frames (*b*-*d*). The frames are “stacked” horizontally within the row; that is, they are placed next to each other, moving right.

   Note that a specific width is applied to the frames within the frame row.
4. After the `<end framerow>` tag, a new frame is inserted. This is stacked in the column; that is, it is placed below the preceding frame. This frame (that contains the table) is given a fixed height and width; this is typically necessary because the size of a table cannot be calculated without knowing its content. It is not necessary to give a size to the frames that contain the pie charts (frames *b* and *c*). We have chosen to do it in this layout, however, to align the pie chart row with the table below.
5. The frame column is ended.

The PDF version of the same layout is shown below. Note the little red markers that indicate that part of the frame is missing.

Information about Employee

Cost price

Sales price

- Jørgen Jansen
- Jack Johnson
- Leif Jensen
- George David Harrison

Name of Employee	Cost price	Sales price
Jørgen Jansen	100.00	200.00
Anders Hansen	110.00	250.00
Julie Petersen	100.00	250.00
Jack Johnson	10.00	25.00
Hans Petersen	100.00	200.00
Klaus Jørgensen	110.00	250.00
Leif Jensen	10.00	25.00
Kurt Christensen	80.00	400.00
Michael Roberts	80.00	120.00

Calculating MPL Attribute Values Using M-Script

This section contains a description of how to modify your layouts on the fly by calling M-Script functions from the layout. This functionality was introduced to support Traffic Lighting, for example, assigning a certain color to the content of a field depending on the value of the field. However, the functionality has a number of additional uses.

To calculate the value of an attribute, assign the attribute value to an M-Script function and pass parameters to the function like this: `attributename=f(a1,...,an)` where `f` is any M-Script function name that is defined in the M-Script package that is associated with the current report, and the arguments `a1 - an` consist of fields or constants.

Do not use this functionality for complex M-Script calculations or large SQL queries, because performance will be too slow. To access the database, use the MQL part of the report.

You can achieve special layout effects by using global M-Script variables; for example you can color every other table row. However, because of the internal structure of the compiled MPL layouts, unpredictable side effects may occur, of which you should be aware.

Attributes and Return Value Types

The following table lists the MPL attributes for which a value can be calculated in M- Script and the expected type of the value that is returned by the M-Script function.

Attribute Name	Return Value Type
rgb | titlergb	struct: {r: v1, g: v2, b: v3} -where v_n are integers
color | titlergb | fontname | component | href | target | report | justification | `title` (on both `text` and `island` tags) | headertitle	*STRING*
bold | italic | underline | zerosuppression	*BOOLEAN*
fontsize	*INTEGER*
interbarspace | maxbarwidth	struct { val: v, unit: s } -where `v` is a *REAL* (the length value) and `s` is a *STRING* (`"pt"`, `"in"`, `"mm"` or `"cm"`)

Attribute Name	Return Value Type
pos	struct { x: v1, xunit: s1, y: v2, xunit: s2 }-where v_1 and v_2 are *REAL* (the length values) and s1 and s_2 are *STRING* ("pt", "in", "mm" or "cm")

Note that no automatic conversion is performed. A function that returns `{val:9,unit:"pt"}` to the attribute `interbarspace` will give a runtime error.

Instead, return `{val:real(9),unit:"pt"}` (or just `9.0`).

The type of the parameters that are passed to the M-Script function is the original database type.

Returning Null from an M-Script Function

To call an M-Script function without applying the return value to any tag, use the tag `functioncall`. This functionality is, for example, used for initializing and updating global variables in the M-script before other functions are called.

```
functioncall < functioncall call=f(…) >
```

- `call` is the only attribute, and it is mandatory. The short form is simply `f(…)`. If the tag is used in rows, it counts as an element, but nothing is shown. Note that `f` must be a function, but the value is ignored, so you might just return `null`.

Example

Consider the following example, which calculates the color of the header title depending on the value of the employee's degree of utilization:

```
<table EmployeeCursor>
   <fields>
   .employeenumber
   .Utilization:color=calcColor(.Utilization)
   <end fields>
<end table>
```

For every employee in the cursor `EmployeeCursor`, a row is built that consists of the employee number and the utilization of the employee. The color of the utilization percentage (the `color` attribute) is determined by the M-script `calcColor`, which takes the MQL field `.Utilization` as its parameter.

The M-script, which is defined in the M-Script package associated with the report, might look like this:

```
public function calcColor(util)
{
   if (util == null)
   return {r:0, g:0, b:0};
   if (util < 50)
   return {r:100, g:0, b:0};
   if (util < 70)
   return {r:100, g:100, b:0};
   if (util > 100)
```

```
    return {r:0, g:100, b:0};
    return {r:0, g:0, b:0};
}
```

If the utilization is `null` or between 70 and 100, the color black is returned. If the utilization is less than 50, red is returned; if it is less than 70, yellow is returned; if it is above 100, green is returned.

Note that the `util` value is also tested for `null`. This is to catch instances where the `calcColor` function is called with no value. For instance, in the preceding example, MPL automatically transfers a default `headertitle` attribute, as if the layout had said:

```
.Utilization:headertitle=<column name>:color=calcColor(.Utilization)
```

Therefore, the M-Script function is called implicitly by the `headertitle` attribute, but without a valid parameter. This case is caught by the M-script testing for `null` and then returning the color black.

Note that attributes (in the preceding example: the `color` attribute) are copied to headers and footers, and therefore the function call is copied as well. However, the parameter that is sent to the M-script might be out of scope in the header and footer. This is a good reason to include the possibility of a `null` value in your M-script.

Standard MPL vs. Reporting MPL

This section contains a description of the differences between standard MPL and the MPL extensions for Universe Reporting.

- Variables

  In MPL for Universe Reporting, no MSL variables are available. The only variables that are available are system variables, of which there currently is only one: `pagenumber`.

- Field sizes

  Because MPL for Universe Reporting is interpreted, rather than compiled and run as for standard prints, the real sizes of the fields are used when formatting. This means that you do not need to set the `width` attribute for fields, unless you want a column to be smaller or bigger than the text size. If you give an element in the `fields` section of a `table` tag a specific width using the `width` attribute, the output is truncated to that width.

- gotoand border

  The `goto` and `border` tags are currently not supported in MPL for Universe Reporting. If you use a `goto` or `border` tag in a layout, it will be ignored.

- Page width

  In standard MPL, an error is issued if the width of an element exceeds the content width (or height). For example, you will get an error or warning if you specify an island to be wider than the page. In MPL for Universe Reporting, these checks are omitted with regard to page width. This means that the MPL compiler does not check whether the layout exceeds the page width. You will, however, still get a warning/error if an element exceeds the width or height of another element, such as an island.

- layoutand page

  The attributes `title`, `print`, and `originallayout`for the `layout`tag are mandatory in standard MPL, but have no meaning in the reporting context and are consequently ignored if they are specified.

  In standard MPL, you must specify the `page` tag. This is not the case in MPL for Universe Reporting. If you omit the `page` tag, the page size is assumed to be A4. The page size is used for calculating the location of page breaks when creating PDF output, for calculating the width of islands, the maximum width of charts, and for right- and center-justification of elements.

- Warnings

  When standard layouts are imported (during installation or by running `MaconomyServer -UP`), warnings are active; that is, layouts with warnings will be compiled and installed. In all other cases warnings are treated as errors, and the import fails.

  In MPL for Universe Reporting, when you run a report for which the layout produces warnings, you will still get an output. However, when you install a report, all warnings are treated as errors.

- Miscellaneous

  The justification of islands has no effect on the HTML output of MPL for Universe Reporting. Islands are always right-justified.

MPL Version 3

MPL version 3 is a major new version of MPL; it breaks backward compatibility in some cases, and it provides new functionality. Since the entire MPL framework has been rewritten and moved to the Java platform, there are a number of areas (not only in the MPL language, but also in font handling and so forth) that have been affected by this. Additionally, a number of new features have been added to MPL 3.

The following sections describe the new futures as well as the changed functionality.

New Features

wrap Attribute for <text>, <field>, and <var> Tags

New in Maconomy version 12 is the ability to have some text fields that can contain multiple lines (using linefeed/newline characters). To be able to preserve this new functionality when printing, the `<text>`, `<field>`, and `<var>` tags now support a new Boolean attribute `wrap`. The attribute has type *BOOLEAN*.

When this attribute is `true`, text that is too long to be displayed in the width that was allocated to the `text`, `field`, or `var` tag will be wrapped, resulting in text that can be more than one line tall. Also any newline characters in the text will result in a line break.

For example, "This text\nis two lines long":wrap+ will result in this output:

This text
is two lines long

The behavior of a `<text>`, `<field>`, and `<var>` tag with the `wrap` attribute, when these attributes are defined:

- When `width` is not specified, the tag with `wrap+` will expand to fill all available width. See below for implications when used in a column of an array.
- When `height` is not specified for a `<var>` or `<field>` tag, the height of the tag becomes unknown (because it is not known at compile time how much text the tag will contain at run time). Therefore, when the tag is used in a context where a fixed height is required (the `canvas` tag, for instance, or a `stack` with a specific height set), you will get a compile error. For the `text` tag, the height is calculated automatically, based on what the width is set to (recall that it will expand to fill).
- `lines` is an additional attribute that you can use to specify the height of the tag in terms of the number of lines that can be shown. You can specify only `lines` or `height`, not both. The attribute has type *INTEGER*.

Furthermore, you should keep these points in mind:

- Because the content of wrapped `<field>` and `<var>` tags is unknown at compile time, the baseline for these is always set to the top line, meaning that they will be aligned with other items in a row using the first line as the baseline.
- Even in a `canvas` tag, you are not required to specify `width`. The tag will simply stretch from its horizontal position to the rightmost boundary of the canvas. However, you must specify `height` in a canvas.

Implicit Conversion to Stretching Column

When a wrapped `text` field or `var` occurs as a column in an array without an explicit width, and that column does not have any width set in a ruler definition, the ruler definition for that column is implicitly converted to become stretchable. This is done to make as much width available for multiline texts as possible.

In the following example, `ruler1` and `ruler2` are equivalent, because `ruler1` becomes `[[stretch+]]` by implicit conversion. Because the field in `ruler3` specifies a width, the column in `ruler3` is not made stretchable. The same applies to the column in `ruler4` because that column definition specifies a width. In effect, the arrays that use `ruler1` and `ruler2` will stretch to fit the entire the entire page width, while the arrays that use `ruler1` and `ruler2` will become 5 cm wide.

```
<ruler ruler1 [[]]>
<ruler ruler2 [[stretch+]]>
<ruler ruler3 [[]]>
<ruler ruler4 [[5cm]]>
<repeating MyCursor>
    {:ruler1
    .MultilineField:wrap+;
    }
    {:ruler2
    .MultilineField:wrap+;
    }
    {:ruler3
    .MultilineField:wrap+:width=5cm;
    }
    {:ruler4
    .MultilineField:wrap+;
    }
<end repeating>
```

The `wrap` attribute propagates upward in the hierarchy, so if, for example, a `stack` contains a wrapped text, and the `width` attribute is not specified on the `stack`, this `stack` will also be stretched and make columns stretch.

In a context where a subruler contains a tag with the `wrap` attribute on, the implicit stretching will affect the root ruler column definition if there is no explicit width set (as demonstrated above). In the following example, column 1 and 2 of `ruler1` will become stretchable because the first column of `ruler2` spans those two columns:

```
<ruler ruler1 [[][][]]>
<subruler ruler2 parent=ruler1 [[1:2]]>
{:ruler1
    ...
}
{:ruler2
    .MultilineField:wrap+;
}
```

concat

A new parenthetical tag, <concat>, is introduced. The <concat> tag is not a stacking tag; instead, it can concatenate text from <text>, <field>, and <var> tags to form one single text. This text behaves as if it had the wrap attribute set to true; that is, it will wrap if contents become too wide or newline characters are encountered.

You can specify style attributes on the concat tag; the tags within will inherit these. Only <text>, <field>, and <var> tags can be put inside concat. Most attributes on the <text>, <field>, and <var> tags inside a concat will be ignored (for example the width attribute), but most style attributes will be applied, for example:

```
<concat>"We confirm that " .Quantity1:bold+ " items have been ordered."<end
concat>
```

Indent and justification style attributes are ignored for tags inside the concat (but can be applied to the concat tag itself).

In addition to style attributes, you can specify these attributes for the concat tag:

- width specifies whether the concat tag should have a fixed width. If this attribute is not specified, it will expand to use all available width in its context. If it is used in a column in an array, the concat behaves like a <text>, <var>, or <field> tag with regard to implicit stretching of those columns; see “wrap Attribute for <text>, <field>, and <var> Tags.” The attribute has type *LENGTH*.
- height specifies a fixed height for the tag. If the height is not sufficient to display the contents when printing, only the lines that are completely visible within the available height will be shown. Alternatively, instead of using height, you can use the lines attribute instead. Only lines or height can be provided, not both. The attribute has type *LENGTH*.
- lines specifies the height of the tag in terms of number of lines that can be shown. Note that the height of the lines depends on the font that is associated with the concat tag. If other font sizes are used in the tags inside the concat tag, the number of lines actually shown may not match the specified number of lines. Only lines or height can be provided, not both. The attribute has type *INTEGER*.
- indent specifies an extra indentation of the text. The attribute has type *LENGTH*.
- pos specifies the positioning of a text element of a canvas. The attribute is nameless and has the type *POS*. It can only be used when the text appears on a canvas, and is mandatory, if so (see “Canvas” for further information about the canvas).

Furthermore, keep these points in mind:

- Because the content of a concat tag is generally unknown at compile time, the baseline for the concat tag is always set to the top line, meaning that it will be aligned with other items in a row using the first line as the baseline. Furthermore, using different font sizes and fonts inside a concat can reduce the accuracy of the baseline calculations.
- Even in a canvas tag, width need not be specified. The tag will simply stretch from its horizontal position to the rightmost boundary of the canvas. height must be specified in a canvas though.

Conditionals

The conditional tag has been modified in several ways. You can now use a conditional on fields (not just variables), and you can also use string fields or variables as a conditional. These two new attributes have been added to the conditional tag:

- `field` specifies the name of the field that must be `true` for the contents to be printed and has the type *STRING*. Note that unlike the `variable` attribute, this attribute cannot be nameless.
- `cursor`. If you want to specify the cursor from which the field should be taken, you can use this attribute. The attribute has the type *ID*. If you do not specify a cursor name, the value will be taken from the nearest cursor with a field of the specified name.

Furthermore, you can now negate a conditional (this functionality previously existed in MPL for Universe Reports only).

If a string variable or field is used in the conditional, the conditional will evaluate to `false` if the string is empty (that is, ""); otherwise, the conditional evaluates to `true`.

See also “Skipping False Conditionals” for the changed functionality for the `conditional` tag: a false conditional no longer skips space.

Paper Orientation Change

You can now switch the paper orientation of a layout when using the `newpage` tag in the outermost parenthetical tag (that is, the paper). This attribute is now recognized:

You can set the orientation to landscape or portrait. The new page after the page break will have this orientation.

Here is a simple example:

```
<paper>
...
<newpage orientation=landscape> -- switch to landscape for the rest of the print
<end paper>
```

Note that there are some side effects:

Page headers and footers will be shown using the least width available throughout the entire print; that is, they will not become wider when switching from portrait to landscape orientation.

Be careful when reusing ruler definitions across different paper orientations. As a general rule, avoid using the same ruler both on a page with landscape orientation and on a page with portrait orientation. Doing this can lead to rulers becoming too wide (in this case a compile error will be issued).

Varying Header/Footer Height

Headers and footers no longer need to have a fixed height. This allows for using wrapped texts and the `concat` tag in a `header` or a `footer`. The header and footer heights are calculated for every page at run time. This also means that they can potentially become taller than a page, leading to a runtime error.

Text Length Greater than 255 Characters

MPL 3 supports texts that are longer than 255 characters. Because fields and variables in Maconomy cannot yet be longer than 255 characters, this change is only visible when using the `text` or the `concat` tags.

Scopes

Definitions of lengths, defaults, and rulers in MPL (the `define`, `redefine`, `default`, `ruler`, and `subruler` tags) can now occur anywhere. They are no longer limited to be specified in the

beginning of a parenthetical tag (such as `stack`). The definitions take effect from the point of definition until the end of the parenthetical tag in which they occur. For example:

```
<stack>
   <stack>
   {:ruler1
         ...-- error: ruler1 is not yet defined
   }
   <ruler ruler1 [[][]]>
   {:ruler1
         ...-- fine: ruler1 is now defined
   }
   <ruler ruler1 [[][][]]> -- redefinition of ruler1
   {:ruler1
         ...-- fine: using the new definition of ruler1
   }
   <end stack>
   {:ruler1
         ...-- error: ruler1 is no longer defined
   }
<end stack>
```

goto is No Longer Allowed in Headers and Footers

You can no longer use the `goto` tag in headers and footers. However, because the positions of headers and footers and footers are (mostly) fixed on the page, it is often fairly easy to translate a `goto` tag into a `skip` tag that will place the contents in the same place on the page. For more precise control, you can apply a <`stack movepos->` with a `canvas` tag inside. Both solutions are demonstrated in this example:

```
<margins top=46pt bottom=30pt left=44pt right=20pt>
...
<header height=110pt>
   <stack movepos->
   <canvas>
         "At (44pt,146pt) of the page":(0pt,100pt)
   <end canvas>
   <end stack>
   "Line 1 in the header"
   "Line 2 in the header"
   <skip 80pt>
   "At (44pt,146pt) of the page"
<end header>
```

You must specify the height of the header manually, because the stack with `movepos-` does not influence the height of the header. The top margin of the page plays a role in determining the position as you can see in the preceding example; while we place the text at 100pt in the canvas, the real position on the page is 146pt (100pt+46pt top margin). Furthermore, the `<skip 80pt>` relies on each of the two lines being 10pt tall (the default baseline skip for small font sizes).

<newpage> in Row No Longer Allowed

You can no longer make a new page in a row, so this example:

```
{:ruler1
    ...
    <newpage>;
    ...
}
```

must be rewritten as:

```
{:ruler1
    ...
}
<newpage>
{:ruler1
    ...
}
```

Skipping False Conditionals

The functionality of the `conditional` tag has been improved in MPL 3 (see also “Conditionals”). An important change that can impact existing layouts is that when the content of a conditional is not shown (because the conditional evaluates to `false`), the conditional leaves no blank space where it would normally have been. For example, in MPL 2 a conditional such as:

```
{
    "First row";
    <conditional SomeVariable> Second row";
    <end conditional> "Third row";
}
```

would always produce three rows; even if the conditional evaluated to `false`, and the contents of the second row were not printed, there would be an empty row instead. Now, this row is skipped entirely. This applies to conditionals in general, no matter where they occur: If the content of a conditional is skipped, the surrounding block shrinks. This means that for an island such as this:

```
<island>
    <conditional SomeVariable>
    ...
<  end conditional>
    ...
<end island>
```

the island (including the island frame) will shrink and grow depending on whether the conditional inside is shown or not.

However, there are some pitfalls that you must be aware of. At compile time we include the potential height of conditionals (as if they were `true`) when calculating the heights of blocks. This is necessary to check whether they can possibly fit on pages or inside a user-specified height. Often this will not cause any problems, but if you rely on bottom baseline alignments between stacks containing conditionals, the result might not be as expected if the conditional content is not skipped.

Header and Footer Height was not Checked in MPL 2

If, for instance, the specified height of a header was less than the height actually occupied by the content, things would just be printed on top of the header, instead of giving an error message in MPL 2. This is no longer allowed. If you want to achieve this effect, you must use `movepos-` on a stacking block inside the header.

This snippet of MPL would work in MPL 2:

```
<header height=60pt>
   <stack height=80pt>
   …
   <end stack>
<end header>
```

but must be changed in MPL 3. You must either update the header height so that it can contain the contents, or you must put a `movepos-` attribute on the stack inside (if it is intended that the stack is higher than the header).

Too-Long Localized Strings are now Clipped

Widths of columns and other tags are calculated at compile time when importing an MPL layout. When dynamically localizing at runtime, this can lead to the localized text becoming too wide to fit the width that was originally calculated. Previously, this would result in this text overlapping other texts. In MPL 3 the text is now clipped to the size of the calculated width, so it will not overwrite other texts and will not be shown in its full length. There is one exception: In a canvas, the text is not clipped.

Length Orientations

The distinction between horizontal and vertical lengths has been mostly removed. For instance, this is now possible:

```
<define MyWidth MinBaselineSkip>
"Some text":width=MinBaselineSkip
```

Previously, this would not be possible because MinBaselineSkip is a vertical length, whereas the width attribute expects horizontal widths.

The only place where the orientation of lengths remains is with lengths that come from the `<grid>` tag. For instance specifying `"Some text":width=2.5grid` will use the horizontal grid length, whereas `height` attributes will use the vertical grid length (as expected).

Minor Changes

There are some additional changes that should not influence most existing prints:

- The `pagebottom` attribute of the footer tag is no longer supported. It had no effect previously anyway.
- The image tag now requires a `pos` attribute when placed in a canvas (like all other tags).
- The `HeaderSkip` length was, contrary to what the MPL manual says, not used between the page header and a block header. Now `HeaderSkip` is used.

Converting MPL 2 to MPL 3

Most layouts will work immediately just by changing the <mpl 1> or <mpl 2> version tag to <mpl 3>. The layout should of course always be tested.

Content Becomes too Wide

In some cases the converted layout will not work out of the box; width and height calculations have changed in MPL 3, so sometimes rulers become a few points (1pt=1/72 inch) wider in MPL 3. This might make the ruler too wide to fit the available width (as determined by the surrounding tag, for example, the paper tag). In this case you must reduce the ruler size by using smaller fonts, decreasing intercolumn spacing, or by providing more space for the ruler by, for example, decreasing page margins.

MPL Version 2 and MPL Version 3 Interoperability

In this section we describe some issues with using both MPL 2 and MPL 3 in one Maconomy system. These are not, as such, related to the MPL language.

Customization

You can customize an MPL 2 original layout so that it becomes an MPL 3 layout, both by shadowing the original MPL 2 layout or by importing it as a new layout.

However, after an original layout has been upgraded to MPL 3, customizing using MPL 2 is not recommended. You should upgrade existing customized MPL 2 layouts.

Print Layout Selection

When you specify Print Layout Selection rules (see the Maconomy Reference Guide), it is important that MPL 2 (or 1) and MPL 3 layouts do not participate in the same set of rules. If an MPL 3 layout is selected when MPL 2 is executing, or vice-versa, an error will occur. It is therefore quite important to verify that this cannot happen when you set up the layout selection rules. Also, updating a single layout that participates in layout selection rules requires updating all layouts that participate in the same rules.

M-Script printGetPDF Function

The M-Script Maconomy API function printGetPDF cannot mix MPL 2 (or 1) and MPL 3 print handles. See the"M-Script Maconomy API Reference" for more information.

Font Path Setup

The method for making additional fonts available in MPL 3 has changed slightly from MPL 2. See "Font Administration in Maconomy" in the Maconomy System Administrator's Guide.

MPL Version 4

MPL 4 is a new major version of MPL, which is a direct successor to and replacement for MPL 3. It introduces some new features, most notably custom database queries and expressions, both defined directly in an MPL layout. Because the concept of an expression is just a generalization of variable/field reference, the `<var>` and `<field>` tags have been replaced by a more general `<eval>` tag, which is used in MPL 4 for evaluating expressions. Note that this change breaks the backward compatibility of MPL 4 with respect to version 3. However, the TPUs as of M16sp0 (Maconomy 2.1) include a tool for automatic migration of MPL 3 layouts to version 4, so this should not be a problem in practice.

You can find the conversion tool in a TPU in ..\JavaMPL\MPL3To4MigrationTool.jar. Run it with the –help option to see the example usage.

This section describes in detail these new features in MPL 4, as well as possible backward compatibility issues with respect to MPL 3.

New Features

This section provides a closer look at expressions, standard functions, and custom database queries using MQL.

Expressions

In MPL 3, there were several ways of printing data on a layout. You could reference:

- A variable in the print environment, by means of the `<var>` tag.
- A cursor field in the print environment, by means of the `<field>` tag.
- A text literal.

In most programming languages these are just examples of atomic expressions, which can be combined with other expressions by means of arithmetic/relational operators, functions, and so on.

This uniform treatment of expressions is now enabled by embedding into MPL 4 the *Expression Language*, which is also used in other Maconomy specification languages like MDML or MWSL. For a detailed reference of the *Expression Language*, see “Functions and Expressions” in the Deltek Maconomy 2.1. MDML Language Reference Guide.

Expressions in MPL are always delimited by curly braces, that is, an opening bracket “{” and a closing bracket “}”. Here are a few examples of valid expressions:

- return a currency symbol for known currency names, otherwise use the currency name:

```
{ if currencyName = "USD" then "$"
  else if currencyName = "EUR" then "€"
  else if currencyName = "GBP" then "£"
  else currencyName }
```

  Note that the `if-then-else` construct is an expression, which means that it always returns a value.

- calculate an average of two variables, x and y, and turn them into a string that represents a percentage value:

```
{(x + y) / 2.0 * 100 + "%"}
```

- using standard function (evaluates to `"Monday"`):

```
{stringWeekday(date(2013,9,30))}
```

Every expression has a type, which can be determined while compiling an MPL 4 layout. It can be one of the following primitive types: *BOOLEAN*, *INTEGER*, *REAL*, *AMOUNT*, *DATA*, *TIME*, *STRING*, or an instance of a pop-up type, for example, *GenderType*.

You can use expressions as attribute values in the following tags:

- **conditional** — where the `expression` attribute of type *EXPRESSION* denotes a condition guarding the conditional tag, for example:

```
<conditional expression={overtime > 10.0}>, or just
<conditional {overtime > 10.0}>
```

- **image** — where the `expression` attribute of type *EXPRESSION* denotes a path to the image to be printed, for example:

```
<image expression={"MyLogos\Logo_" + VenderNoVar + ".png"}>
```

or just

```
<image {"MyLogos\Logo_" + VenderNoVar + ".png"}>
```

In addition to these tags, MPL 4 introduces a number of tags that enable you to perform custom calculations and bind their results to variables/values/ parameters. These include:

- **val** — where the `value` attribute of type *EXPRESSION* denotes a value to be bound to this value (that is, value or constant), for example:

```
<val name=workingDayTime value={8}>, or just
<val workingDayTime {8}>
```

- **var** — where the `value` attribute of type *EXPRESSION* denotes an initial value to be bound to this var (variable), for example:

```
<val name=sum value={0}>, or just
<val sum {8}>
```

- **assign** — where the `value` attribute of type *EXPRESSION* denotes a new value to be assigned to the given var (that is, variable), for example:

```
<assign var=sum value={x + 42}>, or just
<assign sum {x + 42}>
```

- **eval** — where the `expression` attribute of type *EXPRESSION* denotes an expression to be evaluated and printed out, for example:

```
<eval expression ={upperCase(s)}>, or
<eval {upperCase(s)}>
```

or in the short and in most cases preferred form:

```
^{upperCase(s)}
```

- **parameter** — where the `expression` attribute of type *EXPRESSION* represents an actual parameter value to which a query parameter in question will be bound when instantiating the query to a cursor, for example:

```
<cursor name=DraftInvoiceEntry query=DraftInvoiceEntryQuery>
  <parameter JobNumberPar {InvoiceEditingHeader.JobNumber}>
<end cursor>
```

For a more in-depth discussion of these tags and how the `expression` attribute is used in them, see the respective sections describing the tag in question.

Literal Values for Different Types

When declaring a `var` or a `val` or just using literals as values in expressions, it is useful to know how the different literals look for values of different types:

Type	Comma separated example values
INTEGER	`457, 77, -123`
REAL	`41.789, 99.4`
AMOUNT	`AMOUNT(99.74), AMOUNT(6.45)`
BOOLEAN	`true, false`
DATE	`DATE(2013, 12, 25), DATE(1987, 2, 15)`
TIME	`TIME(12, 23, 58), TIME(23, 15, 33)`
STRING	`"Text", "example string"`
POPUP	`GenderType'Male, CountryType'France`

Standard Functions

Along with the Expression Language, MPL 4 enables some of the Maconomy standard functions that are also available in other Maconomy Layout languages like MDML. This section lists all of the functions that are enabled in MPL 4, with an example use for each of them. For a more detailed reference on these functions, see the "MDML and Expression Language Standard Functions" document.

Let us first define some values to be used in the examples.

```
<val d  {date(2013, 10, 11)} >
<val d2 {date(2012, 12, 20)} >
<val t  {time(12, 0, 45)} >
<val t2 {time(10, 1, 18)} >
<val s  {"hello world!"}
```

Generally applicable functions			
Name	**Example**	**Result Value**	**Result type**
isNullOrEmpty	^{ isNullOrEmpty (t)}	No	Boolean

Time Functions			
Name	**Example**	**Result Value**	**Result type**
hour	^{hour(t)}	12	INTEGER
minute	^{minute(t)}	0	INTEGER
second	^{second(t)}	45	INTEGER
addHours	^{addHours(t, 5)}	5:00:45 PM	TIME
addMinutes	^{addMinutes(t, 10)}	12:10:45 PM	TIME
addSeconds	^{addSeconds(t, 1)}	12:00:46 PM	TIME
secondsBetween	^{secondsBetween(t2, t)}	7167	INTEGER
minutesBetween	^{minutesBetween(t2, t)}	119	INTEGER
hoursBetween	^{hoursBetween(t2, t)}	1	INTEGER

Date Functions			
Name	**Example**	**Result Value**	**Result type**
year	^{year(d)}	2013	INTEGER
month	^{month(d)}	10	INTEGER
day	^{day(d)}	11	INTEGER
intWeekday	^{intWeekday(d)}	5	INTEGER
stringWeekday	^{stringWeekday(d)}	friday	STRING
addDays	^{addDays(d, 7)}	2013-10-18	DATE
addMonths	^{addMonths(d, 20)}	2015-6-11	DATE
addYears	^{addYears(d, 10)}	2023-10-11	DATE
addPeriod	^{addPeriod(d, 1, 12, 12)}	2015-10-23	DATE
daysBetween	^{daysBetween(d2, d)}	295	INTEGER
monthsBetween	^{monthsBetween(d2, d)}	9	INTEGER
yearsBetween	^{yearsBetween(d2, d)}	0	INTEGER

String Functions			
Name	**Example**	**Result Value**	**Result type**
length	^{length(s)}	12	INTEGER
startsWith	^{startsWith(s, "hello")}	Yes	BOOLEAN
endsWith	^{endsWith(s, "ape")}	No	BOOLEAN
firstIndexOf	^{firstIndexOf(s, "ll")}	2	INTEGER
indexOf	^{indexOf(s, "ll", 1)}	2	INTEGER
lastIndexOf	^{lastIndexOf(s, "ll")}	2	INTEGER
contains	^{contains(s, "world")}	Yes	INTEGER
substring	^{substring(s, 4)}	o world!	STRING
trim	^{trim(s)}	hello world!	STRING
upperCase	^{upperCase(s)}	HELLO WORLD!	STRING
lowerCase	^{lowerCase(s)}	hello world!	STRING
replaceFirst	^{replaceFirst(s,"ll","dd")}	heddo world!	STRING
replaceAll	^{replaceAll(s, "l", "X")}	heXXo worXd!	STRING
replaceFirstRegEx	^{replaceFirstRegEx(s,"\\w","X")}	Xello world!	STRING
replaceAllRegEx	^{replaceAllRegEx(s,"\\w","X")}	XXXXX XXXXX!	STRING
matchRegEx	^{matchRegEx(s,"\\w+")}	No	BOOLEAN
format	^{format(123456.78, "##0.#####E0")}	123,45678E3	STRING
charAt	^{charAt("MPL4", 1)}	P	STRING

Conversion Functions			
Name	**Example**	**Result Value**	**Result type**
toInteger	toInteger(8.2)	8	INTEGER
toAmount	toAmount(1)	1.00	AMOUNT
toReal	toReal(amount(12.34))	12.34	REAL

In Maconomy, mathematical functions are polymorphic; that is, their return types depend on the types of the arguments that the function takes. As mentioned earlier, every expression in MPL 4 must have a well-defined type; therefore, in case of mathematical functions you must wrap them in a conversion function that indicates to the compiler the expected return type.

Mathematical Functions			
Name	**Example**	**Result Value**	**Result type**
max	^{toReal(max(8.2, 8.3))}	8.3	REAL
min	^{toInteger(min(-1, -2))}	-2	INTEGER
abs	^{toReal(abs(-amount(6.6)))}	6.6	REAL
sig	^{toReal(sign(-7))}	-1.0	INTEGER
floor	^{toAmount(floor(-7.4))}	-8.00	AMOUNT
ceiling	^{toAmount(ceiling(123.4))}	124.00	AMOUNT

Database Queries

MPL 4 allows for defining custom database queries against the standard Maconomy universes using MQL. For a detailed description, see “Database Queries.”

Backward Compatibility Issues

MPL 4 is not entirely backward-compatible with its immediate predecessor, MPL 3. This section describes the small incompatibilities between MPL 3 and 4. Note that the TPUs as of M16 SP0 (Maconomy 2.1) include a tool for the automatic migration of MPL 3 layouts to version 4, so this should not be a problem in practice.

You can find the conversion tool in a TPU in ..\JavaMPL\MPL3To4MigrationTool.jar. Run it with the –help option to see the example usage.

Field and Variable Reference Tags Desupported in MPL 4

As mentioned, in MPL 4 there existed separate tags that represent certain types of values to be printed, that is, field and variable references as well as static text. For completeness reasons, you could use these tags in their full forms, that is, `<field>` (see “Fields”), `<var>` (see Variables”), and `<text>` (see “Texts”). These long forms were, however, discouraged as overly verbose, and

short forms were recommended to be used instead (that is, `varName`, `.fieldName`, and `"text literal`"). In fact, in the entire standard application there was not a single use of these tags in their full form.

In the context of expressions, all of these tags were just particular examples of evaluating an expression and printing out its result. For this reason, `<field>` and `<var>` tags were desupported, and their short forms are now treated as any other instance of the short form for the `<eval>` tag, used for evaluating and printing out expressions. The `<text>` and `<text2>` tags, however, still exist in MPL 4 to account for the static text localization that they support. After text localization is implemented in the Expression Language and enabled in MPL 4 expressions, these tags will be desupported as well, and their short forms will be merged with the `<eval>` tag.

Since the full versions of `<field>` and `<var>` tags are very unlikely to occur in any layout, the only problem that you might encounter in practice is when you use their names in the `<default>` tag, for example:

```
<default tag=field attribute=justification value=right>
<default tag=var attribute=fontsize value=10>
```

In MPL 4, you would use the `<eval>` tag instead to set a default for field/variable references as well as other expressions to be printed out:

```
<default tag=eval attribute=justification value=right>
<default tag=eval attribute=fontsize value=10>
```

Note that the arguably error-prone distinction that allowed for setting different defaults for field and variable references has now been droppe; the default setting applies uniformly to all printable expressions apart from the old-style text literals that are still treated differently.

<var> Tag Means Variable Declaration in MPL 4

Because the old use of the `<var>` tag is desupported in MPL 4 and hence the name *var* became available, it has been used to denote a different concept in MPL 4, namely a variable definition. See “Variable Definitions” for a detailed description of the variable definition tag.

Tags Cannot Contain Spaces before the Tag Name

In MPL 3 it was legal to have a number of whitespaces in between the opening angle bracket “`<`” and the following tag name. For instance, it was legal to write `<  skip>`. MPL 4 is stricter in this respect, and white spaces are not allowed. Therefore, you must write, for example, `<skip>`.

Errors and Warnings

This section contains a list of all of the error messages and warnings that the MPL compiler can display. Error messages and warnings are saved to the file `PrintLayoutErrors.txt` in the Maconomy client folder when the layout is imported.

Some messages can be either an error or a warning. If the MPL layout is used in connection with running a Universe report, the interpretation of the layout is less strict, and a number of messages that in standard MPL are treated as errors are instead treated as warnings. This means that the layout in question can be compiled and run, and that output will be displayed when the layout is used for a Universe report. When the same layout is used in standard MPL, however, the layout will not be compiled, and no output will be printed.

During the import of standard layouts (either during installation or when running `MaconomyServer -UP`), all layouts with warnings will be compiled and installed.

An explanation of the individual error messages is given below each message, including any problem-solving suggestions. If a message can appear as a warning, it is noted also. Furthermore, a section of the messages below (under the heading “Warnings”) can only appear as warnings.

Basic Errors

- (#100) Only version 1 of MPL is supported.

  A version number different from 1 has been specified in the `mpl` tag. This message only appears on servers that do not support MPL 2.

- (#102) Syntax Error.

  The MPL definition is not coherent with the MPL syntax. The error messages only tell you that the error has been detected; the error itself can occur on a previous line.

- (#103) File '*ss*' does not exist.

  The referenced file does not exist.

- (#104) Wrong original print name '*ss*'.

  The name specified in the `print` attribute in the `layout` tag does not match the name of the layout that you are to which you are trying to create/import the MPL layout.

- (#105) Unknown original layout name '*ss*'.

  The name specified in the `originallayout` attribute in the `layout` tag does not exist or is not an original layout name.

- (#108) Wrong layout name '*ss*'.

  The name specified in the `title` attribute in the `layout` tag does not match the name of the layout that you are trying to import the MPL layout to.

- (#109) Layout name length cannot exceed 51 characters.

  The name specified in the `title` attribute in the `layout` tag is longer than 51 characters.

- (#110) Original layout '*ss*' is a Layout Designer layout.

  The layout that you are trying to use as the original layout is not one of the layouts that are created in the system, but have been designed using the Maconomy Layout Designer.

- (#111) Access to original layout '*ss*' is denied.

The `originallayout` attribute in the `layout` tag exists but the user trying to compile the layout does not have access to the layout.

- (#112) Layout shadows an original layout. In tag 'layout' the attribute 'original- layout' must refer to this layout.

  An MPL layout has been imported into an original layout. In this case the `originallayout` attribute must have the same value as the `title` attribute.

- (#113) Layout name cannot be empty.

  The value of the layout attribute `name` cannot be an empty string. In MPL 3, this error message has changed to "Layout title cannot be empty."

- (#114) Tag ’page’ is missing.

  The `page` tag has not been specified.

- (#115) Tag 'ss' is not allowed in standard prints.

  A tag that is only allowed in MPL for Universe Reporting has been specified for a standard print.

- (#116) MPL version dd and above is not supported by this compiler.

  The version of MPL specified in the layout is newer than the version that the MPL compiler supports.

  In MPL 4, the message associated with this error has slightly changed to:

  “Only MPL version 4 is supported by this compiler. Migrate all print layouts starting with **<mpl 3>** using the *MPL3to4MigrationTool* which is found in the TPU. Ask your system administrator for help.”

- (#117) Structure layout 'ss' was not found in the database.

  When you install Maconomy, every print layout exports its Structure layout (used for structure check comparison. See “Print Structure”) to the Maconomy database. If the structure layout for a particular layout was not found in the database, you should contact your system administrator because it is most likely a system set-up error.

- (#118) Invalid syntax of expression 'ss1'. Details: 'ss2'

  This message denotes a syntax error in the given expression. The detailed error message from the *Expression Language* compiler is given.

Lexical Errors

- (#200) Name is too long. Truncated to *dd*characters.

  Names longer than 100 characters are not allowed in MPL. A name can be a cursor, a database name, a variable name, an attribute name, or an attribute value of the type ID.

- (#201) Text is too long. Truncated to *dd*characters.

  Attribute values of the type *STRING* longer than 255 characters are not allowed in MPL.

- (#202) Incomplete text.

  A string has not been finished; that is, the string starts with “ or ‘, but has not been finished by “ or ‘. Maconomy will try to read on, but naturally cannot guess where the string should have been finished. This may cause confusing error messages later in the layout. Simply insert “ or ‘, and try to import the layout again.

- (#203) Error in integer '*ss*'.

  ss is an integer that is too big. Integers cannot exceed 21474836467.

- (#204) A real number must have between 1 and 3 digits after the decimal point.

  A real number of more than three decimals has been specified. Note that this is a syntax error (and is displayed as one) if no decimals are specified (as in '32').

- (#205) Error in real '*ss*'."

  ss is a real number that is too high or too small (the number is VERY high!).

Structure

- (#300) The script structure is invalid.

  The script structure of the MPL printout is not identical with the script structure of the original layout. See also ""Script Structure" in "Print Structure."

- The error message in MPL 4 has been changed to:

  The script structure of the layout is not identical to the script structure of the original layout. Expected script structure 'ss.' Actual script structure: 'ss2.'

- (#301) The stackless structure is invalid.

  The stackless structure of the MPL printout is not a subtree of the stackless structure of the original layout. See "Stackless Structures" in "Print Structure."

- The error message in MPL 4 has been changed to:

  'ss' violates the cursor structure of the original layout , as the corresponding repeating/paper with the same values of attributes does not exist in the original layout. This cursor block is valid only embedded in the following sequence of repeating/paper blocks 'ss'2.

Definitions

- (#400) Unknown paper name '*ss*' or orientation '*ss*' given.

  Either a paper name (the `name` attribute) has been specified in the `page` tag, which is not defined in the Maconomy client window Paper Formats, or the `orientation` attribute has been set to a different value than `landscape` or `portrait`. MPL distinguishes between capitals and regular characters in the paper name.

- (#401) Unknown paper name '*ss*' given.

  A paper name (the `name` attribute) has been specified in the `page` tag, which is defined in the Maconomy client window Paper Formats. MPL distinguishes between capitals and regular characters in the paper name.

- (#402) Grid used but not defined.

  The length unit `grid` has been used, but a `grid` tag has not been specified in the heading as a definition of grid lengths.

- (#403) Grid defined in terms of grid.

  You cannot use the length constant `grid` to define a grid.

- (#405) Unknown parent '*ss*'.

The `parent` attribute in the `subruler` tag refers to an unknown ruler. This can be due to one of following reasons:

- Misspelling of the parent ruler
- The scope defining the ruler is left
- The subruler has been defined in an indented tag, for example, an `island` with a fixed inner margin or a tag with a fixed `indent`. It is not necessarily the closest surrounding tag that is indented.

- (#406) The specification of the subruler is not ordered.

  The specification of the ruler in a `subruler` tag is invalid. The reason is that the referrals to the columns of the parent ruler are not increasing from left to right (see also “Subrulers” in “Arrays“).

- (#407) Subruler not allowed in 'array'.

  The `ruler` attribute defined in an `array` has been assigned a valid subruler value. This is not allowed. Subrulers must be named using the `subruler` tag.

- (#408) Unknown ruler identifier '*ss*'.

  An `array` tag refers to an unknown value. This can be due to one of the following reasons:

  - Misspelling of the ruler
  - The execution of the scope defining the ruler is left
  - The array has been defined in an indented tag, for example, an `island` with a fixed inner margin or a tag with a fixed `indent`. It is not necessarily the closest surrounding tag that is indented.

- (#409) Invalid format for ruler value.

  The ruler specified using the `value` attribute in the `ruler` tag is not correctly formatted. In other words, one or more columns have been specified as intervals (as in subruler values).

- (#410) Invalid format for subruler value.

  The `value` attribute in a `subruler` tag has been set to a value that is not recognized as a subruler. The reason for this is that there is a missing range in a column (*INTEGER* or *INTEGER*:*INTEGER*) or that `stretch` or `width` has been specified in one of the columns.

- (#411) Cannot set default values for tag '*ss*'.

  The value specified in the `tag` attribute in the `default` tag is not recognized. Either the tag does not exist or you have specified a standard value for one of the tags `grid`, `layout`, `margins`, `page`, or `paper`. You cannot specify a standard value for these tags as they should all be specified in the root of the structure. The `default` statement must be specified in a stacking tag.

- (#412) Unknown attribute '*ss1*' in tag '*ss2*'.

  The attribute name *ss1* specified in the `attribute` attribute in the `default` tag does not match the tag name *ss2* specified in the `tag` attribute. In other words, the *ss2* tag does not have an attribute with the name *ss1*. Check the attribute list or the section about the tag in question to find out which attributes are allowed in the tag. This message is a warning if it occurs in MPL for Universe Reports.

- (#413) Attribute '*ss1*' in tag '*ss2*' has type '*ss3*'. The given value had type '*ss4*'.

You have attempted to define a default value for the attribute *ss1* in the tag *ss2*. The specified value (in the `value` attribute) does not have the correct type for this attribute. The specified value has the type *ss3*, but should have had the type *ss4*.

- (#414) Attribute '*ss1*' is mandatory in tag '*ss2*'. Therefore it cannot be set.

  You have attempted to define a default value for the attribute *ss1* in the tag *ss2*. This attribute is mandatory and you are therefore not allowed to set a default value. This message is a warning if it occurs in MPL for Universe Reports.

- (#415) Negative paper width.

  The margins set using the margins tag are so large that the paper width minus the left and right margin have become less than 0.

- (#416) Negative paper height.

  The margins set using the margins tag are so large that the paper height minus the top and bottom margin have become less than 0.

- (#417) Interval specifiers cannot be zero.

  A range (or possibly just an integer) in a subruler definition contains the number 0.

- (#418) Interval specifiers must be less than or equal to the length of the parent ruler (*nn*).

  A range (or possibly just an integer) in a subruler definition refers to a column in the parent ruler that does not exist. This is true for `r3` and `r4` in the following example:

  ```
  <ruler r1 [[][][][]]>
  <subruler r2 parent=r1 [[1][2:3][4]]>
  <subruler r3 parent=r1 [[1][2][4:5]]>
  <subruler r4 parent=r2 [[1][4]]>
  ```

  The reason for this is that *r1* has four columns, but *r3* refers to column #5. Similarly, *r2* only has three columns, but *r4* refers to column #4.

Incorrect Use of Tags

- (#500) Header already defined in this '*ss*'.

  Each parenthetical tag (`paper`, `repeating`, `conditional`, or `stack`) can only be assigned one header.

- (#501) Footer already defined in this '*ss*'.

  Each parenthetical tag (`paper`, `repeating`, `conditional`, or `stack`) can only be assigned one footer.

- (#502) The tag '*ss1*' is not allowed in '*ss2*'.

  An invalid *ss1* tag has been specified in the *ss2* parenthetical tag. This error message may be the result of one of the following situations:

  - A header or footer has been specified in another tag than `paper`, `conditional`, `repeating`, or `stack`.
  - The `span` tag has been specified in another `span` tag.
  - The `newpage` tag has been specified in a row (possible deep within) - this is only allowed if newpage has been specified as the only tag in the row.

- (#503) The tag '*ss*' is not allowed in canvases.

A tag of the type `skip`, `newpage`, `repeating`, or `conditional` has been specified in a canvas. This is not allowed.

- (#504) The tag '*ss*' is not allowed in islands.

  An illegal tag (for example, `repeating`) has been specified in an island.

  You cannot use repetitions, tables, pie charts, and so on, in islands, because these constructions make it impossible for the MPL compiler to determine the size of the island.

- (#505) The tag '*ss*' is not allowed in stacking environments.

  The tags `span` and `vline` cannot be used in stacking tags.

- (#506) The tag '*ss*' is not allowed in headers or footers.

  You cannot use the following tags in headers and footers:

  - header
  - footer
  - repeating
  - conditional
  - newpage, or
  - stackwith scripts, headers, or footers.

- (#507) The tag '*ss*' is not allowed in frontpage.

  The tags `border`, `newpage`, `field`, `repeating`, and `conditional` cannot be specified in a frontpage.

- (#508) The tag '*ss*' can only be used in *ss*.

  Lines can only be used in `canvas`, and `border` can only be used on `paper`.

- (#509) *nn1*elements in row: *nn2*was expected.

  The compiler has encountered a row with *nn1* elements where it expected *nn2* elements. The reason for this is either that the array has a ruler with *nn2* columns or that all previous rows have had *nn2* columns.

- (#510) The tag '*ss*' in arrays should contain rows.

  This error is displayed if the MPL compiler expects that a header or footer contains rows (and not elements). The reason for this might be that the header or footer is specified in a stack, repetition or condition, which should contain rows.

- (#511) Tag 'stack' in *ss*cannot contain scripts, headers or footers.

  If the `script` attribute has been specified in a stack or a header or footer has been assigned to the stack, the stack cannot occur in rows, canvases, or frontpages.

- (#512) At most one conditional can appear in a row.

  A maximum of one condition can be specified in a row. Note that all conditions in the row are counted, regardless of whether or not they are embedded.

- (#513) No repeating tags can appear in a row.

  Repetitions cannot be specified in rows.

- (#514) Rulers are not allowed in '*ss*' containing rows.

You can only define rulers (using the ruler tag) in stacking tags that contain elements- not in stacking tags that contain rows.

- (#515) Rulers are not allowed in canvases.

  You cannot define rulers (using the ruler tag) in canvases.

- (#516) Attribute '*%s*' cannot be set in tag 'hline' appearing in stacking environments.

  `Hline` with spanning is not allowed in stacking environments. The *%s* attribute may be `columns`, `left`, and `right`.

- (#517) The tag '*ss*' is already defined in '*ss*'.

  A tag which can only be defined once in a context has been defined twice (for example, the `fields` tag has been specified twice in a table).

- (#518) Tag 'newpage' can only have orientation in paper.

  The `orientation` attribute of the `newpage` tag can only be used in the `paper` tag.

Fields, Variables, Cursors, and Expressions

- (#600) Variable '*ss*' is unknown.

  The specified variable *ss* is not recognized in this printout.

- (#601) Cursor '*ss*' is unknown.

  The specified cursor *ss* is not recognized in this printout.

- (#602) Field '*ss*' is unknown.

  The specified field *ss* has not been recognized in the position specified. A field is defined in a cursor. It can therefore only occur in the repetition or paper to which the cursor is assigned.

- (#603) Field '*ss1*' is unknown in cursor '*ss2*'.

  A field with an explicit cursor name has been specified (that is, either *ss2.ss1* or `<field title=`*ss1* `cursor=`*ss2*`>)`. The field *ss1* is not recognized in cursor *ss2*.

- (#604) Variable '*ss*' cannot be used in conditionals as its type is wrong.

  A variable which is not of the type *BOOLEAN*, *INTEGER* or *ENUMERATION* (or *STRING* for MPL 3) has been specified as the condition in a conditional.

- (#605) Ruler column index 'nn' out of bounds

  Too many elements were added to the ruler. Remedy: reduce the number of elements, change the definition of the ruler to include more elements, or use another ruler.

- (#606) Duplicate var/val definition: 'ss'.

  It is not allowed to define a new var/val with the same name as another val/var in the same scope.

- (#607) Error(s) found in expression {ss}. 'Detailed error message'.

  An error occurred while compiling the given expression. A detailed description of the error issued by the *Expression Language* compiler is given.

- (#608) Assignment to an unknown var 'ss'.

  The variable *ss* used in the **<assign>** tag is not defined in the current scope.

- (#609) Incompatible types. Cannot assign 'ss1' to 'ss2'.

  MPL is a strongly typed language, which means that every value in MPL has one of the following primitive types: *BOOLEAN*, *INTEGER*, *REAL*, *AMOUNT*, *DATA*, *TIME*, *STRING* or an instance of a popup type, for example, *GenderType*.

  Generally speaking, you can only assign to a variable a new value of the same type as the variable's type. The only exception to this rule is when assigning *AMOUNT* and *REAL*values to a variable of type *INTEGER*.

  When trying to assign a value of incompatible type to a variable, it usually denotes a semantic error in your code. There are, however, some corner cases when one has to convert between various numeric types. To this end, MPL 4 provides the following conversion functions: `toReal`, `toInteger`, and `toAmount`, taking a value of numeric type as a parameter and converting it to a value of the type specified in the name of the function. Note that these conversions might lose precision, for example, when converting a REAL to an INTEGER.

- (#610) Expression 'ss' cannot be used in conditionals as its type is wrong. Expected *BOOLEAN*, *INTEGER* or *STRING*. Got: 'ss2'.

  A condition defined for an MPL conditional must evaluate to a *BOOLEAN*, *INTEGER* or *STRING* value.

- (#611) Duplicate query definition 'ss'.

  A query with the same name 'ss' is already defined in the given scope. Consider renaming your query.

- (#612) Syntax error: invalid query body.

  The query tag is expected to contain a valid *MQL*query body.

- (#613) Cursor 'ss' requires a parameter 'ss2'.

  The MQL query that the cursor 'ss' refers to declares a parameter 'ss2'. Cursor 'ss' must supply an actual value of this parameter using the `<parameter>` tag.

- (#614) Duplicate cursor definition 'ss'.

  A cursor with the same name 'ss' is already defined in the given scope. Consider renaming your cursor.

- (#615) "Cursor 'ss' refers to an unknown query 'ss1."

  The query 'ss1' is not defined in the scope of cursor 'ss.'

- (#616) "Query 'ss' does not expect a parameter 'ss2.'

  The query that the current cursor refers to does not take a parameter named 'ss2.'

- (#617) "Invalid parameter type. Expected type: 'ss'. Got: 'ss2'.

  The parameter in question is expected to be of type 'ss', whereas the type of the given actual parameter value is 'ss2.'

- (#618) Duplicate parameter binding definition: 'ss.'

  The parameter 'ss' has been already supplied with a value.

- (#619) Error validating MQL query. Details: 'Detailed error message'.

  An error occurred while validating the MQL query. A detailed error message from the MQL compiler is given.

- (#620) Cursor name 'query' is reserved for a default top level MQL cursor name. Please use another name.

  As explained above, the cursor name “query” has a special meaning in MQL and hence is reserved.

Warnings

- (#650) Variable '*ss*' is unknown.

  The specified variable *ss* is not recognized in this printout.

- (#651) Cursor '*ss*' is unknown.

  The specified cursor *ss* is not recognized in this printout.

- (#652) Field '*ss*' is unknown.

  The specified field *ss* has not been recognized in the position specified. A field is defined in a cursor. It can therefore only occur in the repetition or paper to which the cursor is assigned.

- (#653) Field '*ss1*' is unknown in cursor '*ss2*'.

  A field with an explicit cursor name has been specified (that is, either *ss2.ss1* or `<field title=`*ss1* `cursor=`*ss2*`>)`. The field *ss1* is not recognized in cursor *ss2*.

- (#776) Attribute 'zerosuppression' can only be used with fields of type 'Amount', 'Integer' and 'Real'.

  The attribute `zerosuppression` has been specified for a field that is not a value field (that is, has the type Amount).

Attributes

- (#700) In '*ss1*': no nameless attributes of type '*ss2*'.

  You have attempted to specify a nameless attribute of the type *ss2* in the tag *ss1*. The error is displayed because the there are no nameless attributes of the type in question.

- (#701) In '*ss1*': no nameless shortattributes of type '*ss2*'.

  The tag *ss1* has been specified in short form combined with an attempt to specify a nameless attribute of the type *ss2*. The error is displayed because there are no nameless short attributes of the type in question.

- (#702) In '*ss1*': attribute '*ss2*' is mandatory.

  The mandatory attribute in the tag *ss1* has been left out.

- (#703) In '*ss1*': unknown attribute '*ss2*'.

  An unknown attribute, *ss2*, has been specified in the tag *ss1*.

- (#704) Attribute '*ss*' is defined more than once.

  The same attribute *ss* has been used more than once. Note that it can be specified as a nameless attribute or as a short form. In the following three examples the `title` attribute to the `text` tag has been specified more than once:

```
<text "Hello" "Hello"> "Hello":title="Hello"
<text "Hello" title="hello">
```

- (#705) Attribute '*ss*' in '*ss*' cannot be given with unit '*ss*'.

You have attempted to define a grid using lengths with the unit grid. This is not allowed as `grid` has not been defined at this point.

- (#706) In attribute '*ss1*': type 'Real' expected. Got '*ss2*'.

  The attribute *ss1* has the type *REAL*, but a value of the type *ss2* has been specified.

- (#707) In attribute '*ss1*': type 'Boolean' expected. Got '*ss2*'.

  The attribute *ss1* has the type *BOOLEAN*, but a value of the type *ss2* has been specified.

- (#708) In attribute '*ss1*': type 'Integer' expected. Got '*ss2*'.

  The attribute *ss1* has the type *INTEGER*, but a value of the type *ss2* has been specified.

- (#709) In attribute '*ss1*': type 'String' expected. Got '*ss2*'.

  The attribute *ss1* has the type *STRING*, but a value of the type *ss2* has been specified.

- (#710) In attribute '*ss1*': type 'ID' expected. Got '*ss2*'.

  The attribute *ss1* has the type *ID*, but a value of the type *ss2* has been specified.

- (#711) In attribute '*ss1*': type 'Length' or 'ID' expected. Got '*ss2*'.

  The attribute *ss1* has the type *LENGTH* (meaning that you should specify a length or a name of a length constant), but a value of the type *ss2* has been specified.

- (#712) In attribute '*ss1*': type 'Interval' expected (format 'Integer' or 'Integer:Integer'). Got '*ss2*'.

  This error message is displayed if you have specified the `interval` attribute (in the named version) with an incorrect value of the type *ss2* in a column or a subcolumn.

- (#713) In attribute '*ss1*': type 'Position' expected. Got '*ss2*'.

  The attribute *ss1* has the type *POS*, but a value of the type *ss2* has been specified.

- (#714) In attribute '*ss*': type 'List' expected. Got '*ss*'.

  The `groupby` attribute to the `repeating` tag has been incorrectly specified.

- (#715) In attribute '*ss*': type 'Ruler' or 'ID' expected. Got '*ss*'.

  A value of an incorrect type has been specified for the attribute *ss1* (either the `ruler` attribute to `array` or the `value` attribute to `ruler` or `subruler`). The attribute must be a ruler or an ID that refers to a defined ruler.

- (#716) Unknown justification '*ss*'.

  An invalid value for the `justification` attribute, *ss*, has been specified. The value must be either `left`, `center`, or `right`. Note that `justification` is nameless in some tags, meaning that if you specify a nameless value of the type *ID*, this may be interpreted as a `justification` value.

- (#717) Unknown length identifier '*ss*'.

  An unknown value for an attribute of the type *LENGTH*, *ss*, has been specified. The reason for this is that the name has been misspelled (the MPL compiler is case sensitive) or that the constant has not been defined using `define`.

- (#718) Conflicting orientations: identifier '*ss*' specifies a horizontal length, but is used in a vertical context.

  The length constant *ss* is horizontal but is used in a context where only vertical lengths can occur. The length constant *ss* is horizontal either because that it is so defined, or because it has been used previously in a horizontal context. Furthermore, length constants, as defined using `textwidth`, are horizontal.

- (#719) Conflicting orientations: identifier '*ss*' specifies a vertical length, but is used in a horizontal context.

  The length constant *ss* is vertical but is used in a context where only horizontal lengths can occur. The length constant *ss* is vertical either because that is so defined, or because it has been used previously in a vertical context. Furthermore, length constants, as defined using `textheight`, are vertical.

- (#720) The first coordinate of the pair is a vertical unit.

  A pair of coordinates (*x,y*) has been specified in which the first coordinate *x* is not horizontal as expected. The reason for this can be that a length with a `textheight` unit has been specified, or that a vertical length constant has been used (see the description of error message #719 for further information about vertical constants).

- (#721) The second coordinate of the pair is a horizontal unit.

  A pair of coordinates (*x,y*) has been specified in which the second coordinate *y* is not vertical as expected. The reason for this can be that a length with a `textwidth` unit has been specified, or that a horizontal length constant has been used (see the description of error message #718 for further information about horizontal constants).

- (#722) Attribute 'orientation' is mandatory for grid length definitions.

  If the length unit for the value specified in a `define` tag is `grid`, an orientation has to be specified. The rule only applies to the `redefine` tag if the redefined length constant has not already been assigned an orientation.

- (#723) Unknown orientation '*ss*'.

  This error message is displayed if a different value than `horizontal` or `vertical` has been specified in the `orientation` attribute to a define or redefine tag.

- (#726) Attribute 'zerosuppression' can only be used with fields of type 'Amount', 'Integer' and 'Real'.

  The attribute `zerosuppression` has been specified for a field which is not a value field (that is, has the type Amount).

- (#728) Attribute 'columns' in tag 'span' must be greater than 0.

  The attribute `columns` in the `span` tag has been set to 0.

- (#729) Attribute 'columns' in tag 'hline' must be greater than 0.

  The attribute `columns` in the `hline` tag has been set to 0.

- (#730) Unknown alignment: '*ss*'

  If the `align` attribute is specified in the `row` tag, it should be assigned one of the values `base`, `top`, `center`, or `bottom`.

- (#731) Unknown baseline: '*ss*'

  If the `baseline` attribute is specified in one of the tags `array`, `stack`, `conditional`, or `repeating`, it should be assigned one of the values `top` or `bottom`. If the attribute has been specified in the `island` tag, the allowed values are `title`, `top`, or `bottom`.

- (#732) Unknown script name '*ss*' (or script is empty).

  An unknown script name has been specified using the `script` attribute.

- (#733) Attribute 'indent' cannot be greater than zero in '*ss*' containing rows.

The indent attribute cannot be used in conditions, repetitions, stacks, and islands containing rows. Only a zero length is allowed (this is only relevant if you have set the standard values differently).

- (#734) Attribute 'height' can only be used in '*ss*' if it contains no repetitions

  The `height` attribute cannot be used in stacks, conditions, or repetitions if the MPL compiler cannot determine the height of the contents. This error message is displayed if the block contains repetitions, conditionals or wrapped fields and variables.

- (#735) Attribute 'stretch' cannot be false in 'island' containing rows.

  The `stretch` attribute cannot be set to `false` in islands containing rows.

- (#736) If 'indent' attribute is set, 'justification' cannot be '*ss*'.

  You can only use the `indent` attribute if the element (`field`, `variable`, `text`, or `island`) is left-adjusted. Note that islands are centered unless otherwise specified, and that certain fields and variables are right-adjusted (depending on type). In this case, it is thus necessary to set the `justification` attribute.

- (#737) Attribute 'baseline' cannot be set to 'title' unless a title is given.

  The `baseline` attribute in an `island` tag has been set to the value `title`, which means that the island's baseline is inherited from the island title. This error message is displayed if no island title has been specified.

- (#738) Horizontal margins cannot be greater than zero in 'island' containing rows.

  You cannot set the `leftmargin` and `rightmargin` attributes in the `island` tag if the island contains rows, because these rows share the ruler with the rows outside of the island. Only a zero length is allowed (this is only relevant if you have set the standard value differently).

- (#739) Length identifier '*ss*' already defined.

  You cannot use the `define` tag to define a length constant already defined. Instead you should use `redefine`. Note that the MPL compiler is case-sensitive.

- (#740) Length identifier '*ss*' not known.

  You have attempted to redefine a length constant, *ss*, using `redefine`, but the constant is unknown. Note that the MPL compiler is case sensitive.

- (#741) Exactly one of 'top' and 'bottom' attributes must be given in tag '*ss*'.

  The `border` or `goto` tag should contain one of the attributes `top` (the distance to the top edge of the page) or `bottom` (the distance to the bottom edge of the page).

- (#742) The '*ss*' attribute value cannot begin or end with blanks.

  The attributes `title, print`, and `originallayout` in the layout tag cannot begin or end with a space.

- (#743) The '*ss*' attribute value cannot contain '*cc*' characters.

  The `title` attribute in the `layout` tag cannot contain underscores, question marks or punctuation marks.

- (#744) Unknown color '*ss*'.

  The `color` attribute has been defined with an unknown color value.

- (#745) RGB values must be between 0 and 100.

The values in an `rgb` must be between 0 and 100.

- (#746) Exactly one of '*title*', 'varname' and 'fieldname' must be defined for tag 'image'.

  The `image` tag should contain one of the attributes `varname` (variable name), `fieldname` (field name) or `title` (string containing image document reference).

- (#748) Exactly one of '*varname*' and '*fieldname*' must be defined for tag 'title'.

  The `title` tag should contain one of the attributes `varname` (variable name) or `fieldname` (field name).

- (#749) Style attributes are not allowed on images.

  The `image` tag does not allow any of the style attributes : `justification`, `fontname`, `fontsize`, `bold`, `italic`, `underline`, `color`, `rgb`.

- (#750) Exactly one of attributes 'href', 'component', 'report' and 'script' must be given.

  A link should contain exactly one of the stated attributes.

- (#751) In attribute '*ss*': type 'PARAMLIST' expected. Got '*ss*'.

  An attribute value of another type than *PARAMLIST* has been specified for the attribute.

- (#753) Justification attribute is only allowed on images if a width is defined.

  To use `justification` on images the width must be defined.

- (#754) Attributes indent, keeptogether and stop are not allowed on stacks in tables.

  You cannot use these attributes on the `stack` tag if it is defined within a table.

- (#755) Unknown target '*%s*' (only 'self', 'new' and 'rightside' allowed).

  The `target` attribute on links only supports the listed targets.

- (#756) Format attribute '*ss*' does not match the type '*ss*' of '*ss*'.

  The type of the field/variable is not the same as the format specified (for example, an `amountformat` has been specified on a field of type *DATE*). This message is a warning if it occurs in MPL for Universe Reports.

- (#757) The content of format attribute '*ss*' in tag '*ss*' is not valid.

  The format specification is invalid. This message is a warning if it occurs in MPL for Universe Reports.

- (#758) In attribute 'rgb': type 'Triple' expected. Got 'ss'.

  The attribute `rgb` has the type *TRIPLE*, but a value of type *ss* has been specified.

- (#759) In tag 'ss': attribute 'pos' and 'indent' cannot be set together.

  Both attribute `pos` and attribute `indent` have been specified together in the same tag. This occurs if you specify the `indent` attribute on. Note: in the old engine specifying `indent` on, for example, a text tag in a canvas would result in this error "(#703) In 'text': unknown attribute 'indent'.

- (#760) Exactly one of 'variable' and 'field' must be defined for tag 'conditional'.

  When using the `conditional` tag, either the (possibly nameless) `variable` attribute must be set or the (non-nameless) `field` attribute (and possibly also the `cursor` attribute) must be set. The `variable` attribute cannot be specified together with the `field` and `cursor` attributes or vice-versa.

- (#761) In tag 'ss': attributes 'height' and 'lines' cannot be set together.

  When setting the `height` attribute on a `text`, `field`, or `var` tag, the `lines` attribute cannot be set at the same time (and vice-versa). Use only one of them to specify the desired height of the tag.

- (#762) In attribute 'ss': type 'Subruler' expected. Got 'ss'.

  When declaring a subruler, the value must be a *SUBRULER* type.

Sizes

- (#800) Tag '*ss*' too wide: element and indentation was *l1*, stacking environment only allows *l2*.

  An element, *ss*, was wider than the width allowed by the surrounding elements. The width of the element including its indent was *l1*, whereas the surrounding elements only allowed the total element width *l2*. This message is a warning if it occurs in MPL for Universe Reports.

- (#801) Tag '*ss*' too wide: element was *l1*, stacking environment only allows *l2*.

  An element, ss, was wider than the width allowed by the surrounding elements. The width of the element was l1, whereas the surrounding elements only allowed the total element width l2. This message is a warning if it occurs in MPL for Universe Reports.

  A common reason for this error is text that was added to a ruler that does not fit the field size allocated on the ruler. In response to this, MPL3 extends the ruler, thus making it too large for the page. The solution is to adjust the ruler to accommodate for the size of the text, or to reduce the size of the added text. Note that the line number that prefixes the error message points to the added text, not the ruler definition.

- (#802) Contents of '*ss*' too high: content was *l1*, 'height' attribute only allows *l2*.

  The contents of the tag *ss* is higher than the height allowed by *ss* in its `height` attribute. This error message may also be displayed for the `frontpage` attributes, if the contents of `frontpage` is higher than the height allowed by the paper format. It could also be because the required row height is greater than the available height. This message is a warning if it occurs in MPL for Universe Reports.

- (#803) Contents of 'array' too wide: content was *l1*, 'width' attribute only allows *l2*.

  The width attribute has been specified as *l2* for an array, but the sum of the columns is *l1*, which is too wide. This message is a warning if it occurs in MPL for Universe Reports.

- (#804) Tag '*ss*' too wide: its far right was at *l1*, canvas only allows *l2*.

  The positioning and width of the tag *ss* results in a right margin edge of *ss* extending beyond the right side of the canvas *ss* is placed in. This message is a warning if it occurs in MPL for Universe Reports.

- (#805) Tag '*ss*' too high: its bottom was at *l1*, canvas only allows *l2*.

  The positioning and height of the tag *ss* results in a bottom edge of *ss* below the bottom of the canvas *ss* is placed in. This message is a warning if it occurs in MPL for Universe Reports.

- (#806) The 'height' *l1*given in 'footer' cannot hold its content (height *l2*).

  A footer height has been specified using the `height` attribute, but the contents of the footer is higher than allowed.

- (#807) The height of 'footer' (*I1*) is greater than the height of the page (*I2*).

  A footer height has been specified using the `height` attribute, which is larger than the page size of the current paper format.

- (#808) The 'height' *I1*given in 'header' cannot hold its content (height *I2*).

  A header height has been specified using the `height` attribute, but the contents of the header is higher than allowed.

- (#809) The height of 'header' (*I1*) is greater than the height of the page (*I2*).

  A header height has been specified using the `height` attribute, which is greater than the page size of the current paper format.

- (#810) The border '*ss*' is placed outside the margins of the page.

  A `border` or `goto` position has been specified using the `top` or `bottom` attribute, which has resulted in positioning of the `border`/`goto` outside the paper margin (possibly outside the paper). This message is a warning if it occurs in MPL for Universe Reports.

- (#811) Canvas element is overlapping with another element (*%s*).

  %s is a line number. This error message is given to help you identify too long localization strings, and can occur when running `MaconomyServer -ULP`. The `ULP` parameter localizes MPL layouts before they are installed on the server. This message is a warning if it occurs in MPL for Universe Reports.

- (#812) Position of 'goto' in '*ss*' was above previous element.

  When using `goto` in a header you cannot go to a position above a previous element in the header, because that would imply a page break, which cannot occur in a header.

- (#813) Position of 'goto' in footer did not make room for ensuing elements.

  When using `goto` in a footer, there has to be room for the elements following the `goto` on the same page.

- (#814) Tag 'goto' is not allowed in '*ss*' with attribute '*ss*'.

  In headers and footers, `goto` is not allowed if attribute `atstart` or `atend` is set.

- (#815) The height of a frontpage must be fixed.

  When using a `field` or `var` tag with the `wrap` attribute set to `true`, or when using a `concat` tag, the height is not fixed. The frontpage must have a fixed height, so you must specify either the `height` or the `lines` attribute on those tags when used in the frontpage.

- (#817) A stack with movepos=false cannot be larger than the page height.

  When using a stack tag with the attribute `movepos` set to false, this stack's height cannot be larger than the available page height.

- (#818) Tag 'ss' must specify height when used in a canvas.

  When using a `field` or `variable` tag with the attribute `wrap` set to `true`, or when using the `concat` tag, the `height` or `line` attribute must be specified as we cannot calculate the height of these tags. Note that if the `width` attribute is not specified for these tags, they will fill the amount of horizontal space available for the canvas.

MDL and MPL Preprocessor

This document describes the preprocessor functionality in MDL and MPL. This was introduced in TPU 53.

The preprocessor functionality is also available from M-Script. For more information see the M-Script Language Reference Manual.

Introduction

The MDL and MPL preprocessor functionality allows code sections of MDL and MPL (both in standard prints and universe reports) to be dependent on add-ons as well as system parameters and system information. This means that you can, for example, define a dialog that only shows a certain island if a certain system parameter has been marked, or a printout that only shows certain information if a certain add-on has been installed.

The main reason for using this is to leave out parts of layouts that are irrelevant in certain setups. This functionality may be used mostly by the Deltek R&D department, but can be used by any layout and MRL report developer.

Versions

The preprocessor is available in MDL and MPL as of TPU 53.

To be able to import MDL layouts using preprocessor directives, a Maconomy Windows client version 4.3.0 is necessary. The preprocessor will, however, work correctly on older clients, and MPL with preprocessor directives can be imported with older clients as well.

Application version 8.0SP11 is necessary for automatic recompilation of MPL after changes to system parameters and system information. This application version also renames all system parameters such that pseudo localization tags (@) are removed.

Preprocessor Options

The preprocessor is applied to MDL and MPL before the relevant compiler is invoked. This means that the preprocessor directives can occur anywhere in the MDL or MPL syntax.

Syntax

The syntax for preprocessor options is as follows:

```
#if <expression>
   ...
#endif
```

and

```
#if <expression>
   #else
#endif
```

The # directives must occur as the first token on a line.

```
<expression>
```

is one of

```
addon(<number>)
```

```
systemparameter.<systemparametername>
systeminformation.<systeminformationfield>
```

For systems that run with Danish kernel language, this last option would be:

```
systemoplysning.<systeminformationfield>.
```

Note that a number of system parameters have an "@" as the first character. If this is the case, you must enclose the entire system parameter in a pair of backslashes, as illustrated in the following examples.

```
## WRONG:
#if systemparameter.@AllowChangeofVATOnInvoiceLines
    ...
## CORRECT:
#if systemparameter.\@AllowChangeofVATOnInvoiceLines\
    ...
```

The system parameter or the system information field must be of the type Boolean. Otherwise, an error message is produced (this can be viewed in the `LayError.txt` resp. `PrintLayoutErrors.txt` file, which is placed in the Maconomy client folder). If an add-on, a system parameter, or a system information field does not exist, it is treated as `false`.

Examples

Write a Text if an Add-On is Set

```
<island "Add-on">
    #if addon(65)
    "Add-on 65 is set"
    #endif
<end island>
```

Write a Text if an Add-On is Set, Otherwise Another

```
<island "Add-on">
#if addon(65)
    "Add-on 65 is set"
#else
    "Add-on 65 is not set"
#endif
<end island>
```

Write in Italics if an Add-On is Set

The preprocessor directives can be used everywhere in the layout.

```
"Hi"
#if addon(65)
    :italic+
#endif
```

Write a Text if a System Parameter is Set

```
#if systemparameter.UseDailyTimeSheets
    "We use daily time sheets"
#endif
```

Write a Text if a System Information Field is Set

```
#if systeminformation.DifferentialVAT
   "We use differential tax"
#endif
```

Negate a Boolean Criterion

There is no syntax for negating a Boolean criterion. Instead, you write:

```
#if systemparameter.UseDailyTimeSheets
#else
   "We don't use daily time sheets"
#endif
```

Warning

You should be aware that using the preprocessor functionality increases the risk of introducing illegal MDL/MPL layouts in the system. Consider the following fragment:

```
#if systemparameter.UseDailyTimeSheets
   "Add-on 65 is set"
#else
   <island "title> .unknownfield <end stack>
#endif
```

The fourth line contains three errors: Missing `"` after `title`, reference to an unknown field, and an attempt to match `<island>` with `<end stack>`.

Nevertheless, if the system is set up to use daily time sheets, this MDL will be validated with no problems. This is because the preprocessor is invoked prior to invoking the MDL compiler, which would otherwise detect the problems.

Now suppose that the system parameter “Use Daily Time Sheets” is changed to `false`. Now the layout will suddenly be invalid, and users will not be able to open the window for which the layout is defined.

It is therefore recommended that you test all MDL/MPL thoroughly before any preprocessor directives are inserted.

Deltek is the leading global provider of enterprise software and information solutions for professional services firms, government contractors, and government agencies. For decades, we have delivered actionable insight that empowers our customers to unlock their business potential. Over 14,000 organizations and 1.8 million users in approximately 80 countries around the world rely on Deltek to research and identify opportunities, win new business, optimize resource, streamline operations, and deliver more profitable projects. Deltek – Know more. Do more.®

deltek.com