

# MUIbase

---

A relational programmable database  
Version 2.9

27 May 2010

Steffen Gutmann

---

Copyright © 2010 Steffen Gutmann

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

# Table of Contents

<b>1</b>	<b>MUIbase Copying Conditions</b>	<b>1</b>
1.1	Donation	1
1.2	Distribution	1
1.3	Mailing List	1
1.4	Disclaimer	1
1.5	Third Party Material	1
1.5.1	Windows, Mac OS and Linux Versions	2
1.5.2	Amiga Version	2
<b>2</b>	<b>Welcome to MUIbase</b>	<b>4</b>
<b>3</b>	<b>Getting Started</b>	<b>5</b>
3.1	Installing MUIbase on Windows	5
3.2	Installing MUIbase on Mac OS	5
3.3	Installing MUIbase on Linux	5
3.4	Installing MUIbase on Amiga	6
3.5	Updating from a Previous Version	6
3.6	Starting MUIbase	7
3.7	Quitting MUIbase	7
3.8	Filename Conventions on Windows, Mac OS and Linux	7
<b>4</b>	<b>Tutorial</b>	<b>9</b>
4.1	How MUIbase Works	9
4.2	Starting with a Project, the Structure Editor	9
4.3	Adding a Table	9
4.4	Adding an Attribute	10
4.5	Displaying the Project	10
4.6	Adding two Reference Attributes	11
4.7	Adding Records	12
4.8	Filter	12
4.9	Queries	13
4.10	Adding a Table with a Memo and a Button Attribute	13
4.11	Programming MUIbase to do a Pedigree	14
4.12	Programming MUIbase to List a Person's Children	15
<b>5</b>	<b>Basic Concepts</b>	<b>18</b>
5.1	Projects	18
5.2	Tables	18
5.3	Records	18
5.4	Attributes	19
5.5	Attribute Types	19
5.5.1	String Attributes	19

5.5.2	Integer Attributes .....	19
5.5.3	Real Attributes .....	20
5.5.4	Boolean Attributes .....	20
5.5.5	Choice Attributes .....	20
5.5.6	Date Attributes .....	20
5.5.7	Time Attributes .....	20
5.5.8	Memo Attributes .....	21
5.5.9	Reference Attributes .....	21
5.5.10	Virtual Attributes .....	21
5.5.11	Buttons .....	21
5.6	Table of Attribute Types .....	21
5.7	Memory Consumption .....	22
5.8	Relationships .....	23
5.8.1	One to one Relationships .....	23
5.8.2	One to many Relationships .....	23
5.8.3	Many to many Relationships .....	24
5.9	User Interface .....	25
5.9.1	Windows .....	25
5.9.2	Masks .....	26
5.9.3	Panels .....	26
5.9.4	Attribute Objects .....	26
5.9.5	Text Objects .....	26
5.9.6	Images .....	26
5.9.7	Space Objects .....	27
5.9.8	Groups .....	27
5.9.9	Balance Objects .....	27
5.9.10	Register Groups .....	27
<b>6</b>	<b>Managing Projects .....</b>	<b>28</b>
6.1	File Format .....	28
6.2	Info .....	28
6.3	New Project .....	28
6.4	Clear Project .....	28
6.5	Open Project .....	29
6.6	Save Project .....	29
6.7	Admin and User Mode .....	30
6.8	Check Data Integrity .....	30
6.9	Swap Records .....	30
6.10	Close Project .....	31

<b>7</b>	<b>Preferences</b>	<b>32</b>
7.1	User Settings	32
7.1.1	Formats	32
7.1.2	External Editor	32
7.1.3	External Viewer	33
7.1.4	Extra Buttons in Tab Chain	33
7.1.5	Advance on Enter	33
7.1.6	Confirm Quit	34
7.1.7	MUI	34
7.2	Project Settings	34
7.2.1	Record Memory	34
7.2.2	Record Delete Requester	35
7.2.3	Paths Relative to Project	35
7.2.4	Confirm Save & Reorg	35
7.2.5	Program Source	35
7.2.6	Cleanup External Program Source	35
7.2.7	Program Debug Information	36
7.2.8	Obsolete Functions	36
7.2.9	Sort Trigger Functions	36
7.2.10	Program Include Directory	36
7.2.11	Program Output File	36
7.3	Save as Default	37
<b>8</b>	<b>Record-Editing</b>	<b>38</b>
8.1	Active Object	38
8.2	Adding Records	38
8.3	Changing Records	38
8.3.1	String Attributes with Pop-up Button	38
8.3.2	Entering Integer Values	39
8.3.3	Entering Boolean Values	39
8.3.4	Entering Choice Values	39
8.3.5	Entering Date Values	39
8.3.6	Entering Time Values	39
8.3.7	Memo Context Menu	39
8.3.8	Entering Reference Values	40
8.3.9	Entering NIL Value	40
8.4	Deleting Records	40
8.5	Browsing Records	40
<b>9</b>	<b>Filter</b>	<b>42</b>
9.1	Record Filter	42
9.1.1	Filter Expression	42
9.1.2	Changing Filters	42
9.1.3	Filter Examples	43
9.2	Reference Filter	43

<b>10</b>	<b>Order</b>	<b>44</b>
10.1	Empty Order	44
10.2	Order by Attributes	44
10.3	Order by a Function	45
10.4	Changing Orders	45
10.5	Reorder all Records	46
<b>11</b>	<b>Search For</b>	<b>47</b>
11.1	Search Requester	47
11.2	Forward/Backward Search	47
11.3	Search Pattern Examples	48
<b>12</b>	<b>Import and Export</b>	<b>49</b>
12.1	File Format	49
12.2	Sample Import File	49
12.3	Importing Records	50
12.4	Exporting Records	50
<b>13</b>	<b>Data Retrieval</b>	<b>51</b>
13.1	Select-From-Where Queries	51
13.2	Query Editor	51
13.3	Printing Queries	52
13.4	Query Examples	53
<b>14</b>	<b>Structure Editor</b>	<b>56</b>
14.1	Table Management	56
14.1.1	Creating Tables	56
14.1.2	Changing Tables	57
14.1.3	Deleting Tables	57
14.1.4	Sorting Tables	57
14.2	Attribute Management	57
14.2.1	Creating Attributes	57
14.2.2	Type Specific Settings	58
14.2.3	Label Editor	59
14.2.4	Copying Attributes	59
14.2.5	Changing Attributes	59
14.2.6	Deleting Attributes	60
14.2.7	Sorting Attributes	60
14.3	Display Management	60
14.3.1	Display Field	61
14.3.2	Panel Editor	62
14.3.3	Attribute Object Editor	62
14.3.4	Type Specific Settings	63
14.3.5	Text Editor	67
14.3.6	Image Editor	67
14.3.7	Space Editor	67
14.3.8	Group Editor	68

14.3.9	Register Group Editor .....	68
14.3.10	Window Editor .....	68
14.4	Export Structure .....	69
<b>15</b>	<b>Programming MUIbase .....</b>	<b>70</b>
15.1	Program Editor .....	70
15.2	External program source .....	70
15.3	Preprocessing .....	71
15.3.1	#define .....	71
15.3.2	#undef .....	71
15.3.3	#include .....	71
15.3.4	#if .....	72
15.3.5	#ifdef .....	72
15.3.6	#ifndef .....	72
15.3.7	#elif .....	72
15.3.8	#else .....	73
15.3.9	#endif .....	73
15.4	Programming Language .....	73
15.4.1	Why Lisp? .....	73
15.4.2	Lisp Syntax .....	73
15.4.3	Kinds of Programs .....	74
15.4.4	Name Conventions .....	74
15.4.5	Accessing Record Contents .....	74
15.4.6	Data Types for Programming .....	75
15.4.7	Constants .....	76
15.4.8	Command Syntax .....	77
15.5	Defining Commands .....	77
15.5.1	DEFUN .....	77
15.5.2	DEFUN* .....	78
15.5.3	DEFVAR .....	78
15.5.4	DEFVAR* .....	79
15.6	Program Control Functions .....	79
15.6.1	PROGN .....	79
15.6.2	PROG1 .....	79
15.6.3	LET .....	79
15.6.4	SETQ .....	80
15.6.5	SETQ* .....	80
15.6.6	FUNCALL .....	81
15.6.7	APPLY .....	81
15.6.8	IF .....	81
15.6.9	CASE .....	81
15.6.10	COND .....	82
15.6.11	DOTIMES .....	82
15.6.12	DOLIST .....	83
15.6.13	DO .....	83
15.6.14	FOR ALL .....	84
15.6.15	NEXT .....	84
15.6.16	EXIT .....	84

15.6.17	RETURN.....	85
15.6.18	HALT.....	85
15.6.19	ERROR.....	85
15.7	Type Predicates.....	86
15.8	Type Conversion Functions.....	86
15.8.1	STR.....	86
15.8.2	MEMO.....	87
15.8.3	INT.....	87
15.8.4	REAL.....	88
15.8.5	DATE.....	88
15.8.6	TIME.....	89
15.9	Boolean Functions.....	89
15.9.1	AND.....	89
15.9.2	OR.....	90
15.9.3	NOT.....	90
15.10	Comparison Functions.....	90
15.10.1	Relational Operators.....	90
15.10.2	CMP.....	91
15.10.3	CMP*.....	91
15.11	Mathematical Functions.....	91
15.11.1	Adding Values.....	91
15.11.2	Subtracting Values.....	92
15.11.3	1+.....	92
15.11.4	1-.....	92
15.11.5	Multiplying Values.....	93
15.11.6	Dividing Values.....	93
15.11.7	DIV.....	93
15.11.8	MOD.....	93
15.11.9	MAX.....	93
15.11.10	MIN.....	93
15.11.11	ABS.....	94
15.11.12	TRUNC.....	94
15.11.13	ROUND.....	94
15.11.14	RANDOM.....	94
15.11.15	POW.....	94
15.11.16	SQRT.....	95
15.11.17	EXP.....	95
15.11.18	LOG.....	95
15.12	String Functions.....	95
15.12.1	LEN.....	95
15.12.2	LEFTSTR.....	95
15.12.3	RIGHTSTR.....	96
15.12.4	MIDSTR.....	96
15.12.5	SETMIDSTR.....	96
15.12.6	INSMIDSTR.....	96
15.12.7	INDEXSTR.....	96
15.12.8	INDEXSTR*.....	97
15.12.9	INDEXBRK.....	97



15.12.10	INDEXBRK*	97
15.12.11	RINDEXSTR	97
15.12.12	RINDEXSTR*	97
15.12.13	RINDEXBRK	97
15.12.14	RINDEXBRK*	98
15.12.15	REPLACESTR	98
15.12.16	REPLACESTR*	98
15.12.17	REMCHARS	98
15.12.18	TRIMSTR	98
15.12.19	WORD	99
15.12.20	WORDS	99
15.12.21	STRTOLIST	99
15.12.22	LISTTOSTR	99
15.12.23	CONCAT	100
15.12.24	CONCAT2	100
15.12.25	COPYSTR	100
15.12.26	SHA1SUM	100
15.12.27	UPPER	101
15.12.28	LOWER	101
15.12.29	ASC	101
15.12.30	CHR	101
15.12.31	LIKE	101
15.12.32	SPRINTF	102
15.13	Memo Functions	104
15.13.1	LINE	104
15.13.2	LINES	104
15.13.3	MEMOTOLIST	104
15.13.4	LISTTOMEMO	105
15.13.5	FILLMEMO	105
15.13.6	FORMATMEMO	105
15.13.7	INDENTMEMO	106
15.14	Date and Time Functions	106
15.14.1	DAY	106
15.14.2	MONTH	106
15.14.3	YEAR	106
15.14.4	DATEDMY	106
15.14.5	MONTHDAYS	107
15.14.6	YEARDAYS	107
15.14.7	ADDMONTH	107
15.14.8	ADDYEAR	107
15.14.9	TODAY	108
15.14.10	NOW	108
15.15	List Functions	108
15.15.1	CONS	108
15.15.2	LIST	108
15.15.3	LENGTH	108
15.15.4	FIRST	109
15.15.5	REST	109

15.15.6	LAST .....	109
15.15.7	NTH .....	109
15.15.8	APPEND .....	109
15.15.9	REVERSE .....	109
15.15.10	MAPFIRST .....	110
15.15.11	SORTLIST .....	110
15.15.12	SORTLISTGT .....	110
15.16	Input Requesting Functions .....	111
15.16.1	ASKFILE .....	111
15.16.2	ASKDIR .....	111
15.16.3	ASKSTR .....	111
15.16.4	ASKINT .....	111
15.16.5	ASKCHOICE .....	112
15.16.6	ASKCHOICESTR .....	113
15.16.7	ASKOPTIONS .....	113
15.16.8	ASKBUTTON .....	114
15.16.9	ASKMULTI .....	114
15.17	I/O Functions .....	116
15.17.1	FOPEN .....	116
15.17.2	FCLOSE .....	117
15.17.3	stdout .....	117
15.17.4	PRINT .....	117
15.17.5	PRINTF .....	117
15.17.6	FPRINTF .....	118
15.17.7	FERROR .....	118
15.17.8	FEOF .....	118
15.17.9	FSEEK .....	118
15.17.10	FTELL .....	119
15.17.11	FGETCHAR .....	119
15.17.12	FGETCHARS .....	119
15.17.13	FGETSTR .....	119
15.17.14	FGETMEMO .....	120
15.17.15	FPUTCHAR .....	120
15.17.16	FPUTSTR .....	120
15.17.17	FPUTMEMO .....	120
15.17.18	FFLUSH .....	120
15.18	Record Functions .....	120
15.18.1	NEW .....	121
15.18.2	NEW* .....	121
15.18.3	DELETE .....	121
15.18.4	DELETE* .....	122
15.18.5	DELETEALL .....	122
15.18.6	GETMATCHFILTER .....	122
15.18.7	SETMATCHFILTER .....	122
15.18.8	GETISSORTED .....	123
15.18.9	SETISSORTED .....	123
15.18.10	RECNUM .....	123
15.18.11	COPYREC .....	123

15.19	Attribute Functions.....	124
15.19.1	ATTRNAME.....	124
15.19.2	MAXLEN.....	124
15.19.3	GETLABELS.....	124
15.19.4	SETLABELS.....	124
15.20	Table Functions.....	125
15.20.1	TABlename.....	125
15.20.2	GETORDERSTR.....	125
15.20.3	SETORDERSTR.....	125
15.20.4	REORDER.....	126
15.20.5	REORDERALL.....	126
15.20.6	GETFILTERACTIVE.....	126
15.20.7	SETFILTERACTIVE.....	127
15.20.8	GETFILTERSTR.....	127
15.20.9	SETFILTERSTR.....	127
15.20.10	RECORDS.....	128
15.20.11	RECORD.....	128
15.20.12	SELECT.....	128
15.21	GUI Functions.....	129
15.21.1	SETCURSOR.....	129
15.21.2	GETWINDOWOPEN.....	129
15.21.3	SETWINDOWOPEN.....	129
15.21.4	GETVIRTUALLISTACTIVE.....	129
15.21.5	SETVIRTUALLISTACTIVE.....	130
15.22	Project Functions.....	130
15.22.1	PROJECTNAME.....	130
15.22.2	CHANGES.....	130
15.22.3	GETADMINMODE.....	130
15.22.4	SETADMINMODE.....	131
15.22.5	ADMINPASSWORD.....	131
15.23	System Functions.....	131
15.23.1	EDIT.....	131
15.23.2	EDIT*.....	131
15.23.3	VIEW.....	131
15.23.4	VIEW*.....	132
15.23.5	SYSTEM.....	132
15.23.6	SYSTEM*.....	132
15.23.7	STAT.....	132
15.23.8	TACKON.....	132
15.23.9	FILENAME.....	133
15.23.10	DIRNAME.....	133
15.23.11	MESSAGE.....	133
15.23.12	COMPLETEmax.....	133
15.23.13	COMPLETEADD.....	134
15.23.14	COMPLETE.....	134
15.23.15	GC.....	134
15.23.16	PUBSCREEN.....	135
15.24	Pre-Defined Variables.....	135

15.25	Pre-Defined Constants .....	135
15.26	Functional Parameters .....	136
15.27	Type Specifiers .....	136
15.28	Semantics of Expressions .....	137
15.29	Function Triggering .....	138
15.29.1	onOpen .....	138
15.29.2	onClose .....	138
15.29.3	onAdminMode .....	139
15.29.4	onChange .....	139
15.29.5	New Trigger .....	139
15.29.6	Delete Trigger .....	140
15.29.7	Comparison Function .....	140
15.29.8	Attribute Trigger .....	141
15.29.9	Programming Virtual Attributes .....	142
15.29.10	Compute Enabled Function .....	142
15.29.11	Double Click Trigger .....	143
15.29.12	Compute Listview Labels .....	143
15.29.13	Compute Reference Records .....	144
15.30	List of Obsolete Functions .....	144
<b>16</b>	<b>ARexx Interface .....</b>	<b>145</b>
16.1	Port Name .....	145
16.2	Command Syntax .....	145
16.3	Return Codes .....	146
16.4	Quit .....	146
16.5	Hide .....	147
16.6	Show .....	147
16.7	Info .....	147
16.8	Help .....	147
16.9	Compile .....	147
16.10	Connect .....	148
16.11	Disconnect .....	148
16.12	Connections .....	148
16.13	Eval .....	149
16.14	Transaction .....	150
16.15	Commit .....	150
16.16	Rollback .....	150
	<b>Acknowledgments .....</b>	<b>151</b>
	<b>Author .....</b>	<b>152</b>
	<b>Function Index .....</b>	<b>153</b>
	<b>Concept Index .....</b>	<b>156</b>

# 1 MUIbase Copying Conditions

MUIbase is Copyright © 1998-2010 Steffen Gutmann. All rights reserved.

MUIbase is open source software distributed under the terms of the GNU General Public License (GPL).

## 1.1 Donation

MUIbase is completely free of charge. However, if you like the project and would like to support it then you are welcome to make a donation. Please visit <http://www.sourceforge.net/projects/muibase> for information on how to donate to MUIbase.

## 1.2 Distribution

Binaries of the latest version of MUIbase can be down-loaded from <http://muibase.sourceforge.net>. Source code is hosted under <http://www.sourceforge.net/projects/muibase>.

You are granted the right to share MUIbase with others, as long as you distribute the MUIbase archive exactly as you received it, with all associated files included.

Under no circumstances may you charge a copying fee or shipping cost for distributing the MUIbase archive without express written consent from the copyright holder.

## 1.3 Mailing List

A mailing list for discussion of MUIbase-related issues has been setup under <http://www.groups.yahoo.com/groups/muibase-list>. Everybody is welcome to join. Please send bug reports or feature requests to this mailing list.

## 1.4 Disclaimer

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## 1.5 Third Party Material

MUIbase uses external libraries and other material depending on the operating system.

### 1.5.1 Windows, Mac OS and Linux Versions

MUIbase for Windows, Mac OS and Linux uses the Gimp Toolkit (GTK+) version 2.6 or higher, copyright © 2007-2008 The GTK+ Team. GTK is a toolkit for developing graphical user interfaces (GUI) and is distributed under the GNU Library General Public License (LGPL). For more information, see <http://www.gtk.org>.

The Windows install script for MUIbase is based on the install script for The Gimp which is copyright by Jernej Simoncic.

### 1.5.2 Amiga Version

The Amiga version of MUIbase uses

MUI - MagicUserInterface

Copyright © 1992-2008 Stefan Stuntz

MUI is a system to generate and maintain graphical user interfaces. With the aid of a preferences program, the user of an application has the ability to customize the outfit according to his personal taste.

MUI is distributed as shareware. To obtain a complete package containing lots of examples and more information about registration please look for a file called "muiXXusr.lha" (XX means the latest version number) on your local bulletin boards or on public domain disks.

If you want to register directly, feel free to send

DM 30.- or US\$ 20.-

to

Stefan Stuntz  
Eduard-Spranger-Straße 7  
80935 München  
GERMANY

Support and online registration is available at

<http://www.sasg.com/>

MUIbase for Amiga uses NList.mcc, copyright © 2001-2008 NList Open Source Team. See <http://www.sourceforge.net/projects/nlist-classes> for more info or latest version.

MUIbase for Amiga uses BetterString.mcc, copyright © 2005-2009 BetterString Open Source Team. See <http://www.sourceforge.net/projects/bstring-mcc> for more info or latest version.

MUIbase for Amiga uses TextEditor.mcc, copyright © 2005-2009 TextEditor Open Source Team. See <http://www.sourceforge.net/projects/texteditor-mcc> for more info or latest version.

MUIbase for Amiga uses codesets.library, copyright © 2005-2009 codesets.library Open Source Team. See <http://www.sourceforge.net/projects/codesetslib> for more info or latest version.

Some icons used in the MUIbase for Amiga distribution are from the DefaultIcons set and are copyright © Michael-Wolfgang Hohmann and Angela Schmidt.

## 2 Welcome to MUIbase

MUIbase is a fast and flexible database system with graphical user interface (GUI) and programmability. It is targeted for advanced desktop users who want to manage data in a comfortable and powerful way. MUIbase is able to manage any kind of data, e.g. addresses, CD series, movies, photo collections, your family tree, your income and expense, and much more. The power of MUIbase lies in its clear and powerful graphical user interface and its programming capabilities. Programming MUIbase allows you to process data in various ways, e.g. automatic calculations upon user input, generation of reports, import and export of data, etc. For example it can be used for calculating the total amount of income, or the total amount of recorded time on a CD, or to automatically create and print serial letters to your customers.

MUIbase offers the following features:

- Unlimited number of projects, tables, attributes, and records.
- Attributes can be of type string, memo (multi line text), integer, real, date, time, Boolean, choice (one item out of many items), reference (easy way to reference a record of another table), button (for starting MUIbase programs), and virtual (compute value on the fly).
- The string type can also manage lists of strings, files, and fonts. A string can refer to an external image which is displayed in the graphical user interface.
- Dynamic loading of records. Records which are not needed may be flushed from memory (e.g. when memory is low).
- Programmability. With the easy and powerful MUIbase programming language complex tasks can be implemented. The language also includes a `SELECT FROM WHERE` query for comfortable and fast data retrieval.
- Ordering of records by any combinations of attributes.
- Flexible and powerful search and filter facility.
- Query editor which allows entering and managing of `SELECT FROM WHERE` queries. The queries can be saved and the results can be printed.
- Import and export facility.
- Full documentation including user and programming manual in HTML and PDF (plus AmigaGuide on the Amiga).
- The Amiga version provides a powerful ARexx interface for accessing MUIbase from other programs. The ARexx interface offers a transaction mechanism similar to other relational databases.
- Operating system independence. MUIbase is available for Windows, Mac OS, Linux and Amiga. The source code is available as part of a Source Forge project.



## 3 Getting Started

This chapter describes what the hardware and software requirements are to run MUIbase on your computer, the procedure for installing and updating MUIbase, and how to start and quit MUIbase.

### 3.1 Installing MUIbase on Windows

MUIbase for Windows runs on Intel 386 compatible computers that use the 32bit versions of Windows 2000, Windows XP or Windows Vista. MUIbase requires the GTK+ libraries for Windows to be installed on your system. Usually, these libraries are not present in most Windows installations unless you have installed other software that also requires them, e.g. Gimp. The hard disk space required for MUIbase is moderate (less than 10MB).

The MUIbase installer is distributed as a Windows executable, e.g. ‘`MUIbase-2.9-setup.exe`’ which you can download from the MUIbase home page <http://muibase.sourceforge.net>. When running the installer, your system will first be checked for the necessary GTK+ libraries. In case they cannot be found on your computer, the installer prompts a message allowing to download and install these libraries from the Internet. The installer then resumes to installing MUIbase where you will be asked several options about where to install MUIbase and which configuration to use. In case you have a previous MUIbase version installed on your computer, the installation procedure allows to update it to the new version.

After a successful installation your Windows ‘**Start**’ menu should contain a new entry for launching MUIbase. You can now remove the MUIbase installer as it is no longer needed.

### 3.2 Installing MUIbase on Mac OS

MUIbase for Mac OS supports Mac OS X Tiger (10.4) and Mac OS X Leopard (10.5). The software is distributed as universal binaries containing both, a PowerPC and an Intel version.

A disk image of MUIbase, e.g. ‘`MUIbase-2.9.dmg`’ can be downloaded from the MUIbase home page <http://muibase.sourceforge.net>. After downloading and opening the image, drag and drop ‘`MUIbase.app`’ into your computer’s ‘**Applications**’ folder. This installs MUIbase and all necessary libraries including GTK+ on your system.

The default appearance of MUIbase for Mac OS uses the default GTK engine. If you rather prefer a more native look and feel then rename the file ‘`gtkrc-quartz`’ in directory ‘`MUIbase.app/Contents/Resources/etc/gtk-2.0`’ into ‘`gtkrc`’ and restart MUIbase.

Note that this setting is experimental and the appearance of some GUI elements may be broken.

### 3.3 Installing MUIbase on Linux

You can install and run MUIbase on an Intel 386 compatible computer running the Linux operating system. Your system should have a recent X11 server and GTK+ (version 2.6 or higher) installed. Required hard disk space is moderate (less than 10MB).

MUIbase for i386 Linux is distributed both, as a Debian package (e.g. ‘`muibase_2.9_i386.deb`’ for Ubuntu or Debian) and as an RPM archive (e.g.

‘*MUIbase-2.9-i386.rpm*’ for Fedora or Mandriva). Both versions can be downloaded from the MUIbase home page <http://muibase.sourceforge.net>.

Usually MUIbase installs or updates itself automatically after downloading it on your computer. In case the installation process does not start automatically, the following commands executed as super user (root) in a command shell install or update MUIbase on a Debian-based Linux computer:

```
dpkg --install muibase.deb
```

where *muibase.deb* is the downloaded MUIbase Debian package.

On a Redhat-based system (RPM), the commands are:

```
rpm -Uvh MUIbase.rpm
```

where *MUIbase.rpm* is the downloaded RPM archive.

After a successful installation you should see a new menu item in the ‘*Applications - Office*’ menu tree of your (Gnome or KDE) desktop. At this time the downloaded archive is no longer necessary and can be removed.

### 3.4 Installing MUIbase on Amiga

MUIbase for Amiga runs on Amiga OS version 3.0 or higher and requires an 68020 processor or better. Furthermore, MUI version 3.8 or higher must be present on your system. A minimum of 4MB RAM and 10MB of free hard disk space are needed. The m68k binary of MUIbase for Amiga has been reported to run stable on UAE, MorphOS and Amiga OS4. PowerPC binaries for MorphOS and Amiga OS4 are also available as part of the distributed archive. A seprate distribution contains an AROS version.

MUIbase is distributed as an lha archive, e.g. ‘*MUIbase-2.9.1.lha*’ and can be downloaded from the MUIbase home page <http://muibase.sourceforge.net>. In order to install MUIbase on your computer unpack this archive to a temporary directory. Do not unpack it to the target directory!

Double click the MUIbase installer script ‘*Install-MUIbase*’ and follow its instructions. The script asks for a directory where the software should be copied to. Do not enter the directory where you have unpacked the MUIbase archive to. The script is also capable of updating an existing MUIbase installation by entering the directory where you previously installed MUIbase.

After successful installation you will find a (new) drawer on your system containing the MUIbase program along with all necessary files and some sample projects. In addition an assign ‘*MUIbase:*’ referring to this drawer has been added to your system file ‘*S:User-Startup*’. It is safe to remove the lha archive and the temporary files as they are no longer needed.

### 3.5 Updating from a Previous Version

When installing MUIbase, you can upgrade from a previous version of MUIbase or reinstall it as described above. During the new installation, all necessary files are replaced by new ones. This includes the sample projects in the ‘*Demos*’ directory. It is therefore not recommended to place your own projects into this directory nor to use any of the sample projects for managing your own data as they are overwritten when reinstalling or upgrading MUIbase.

The recommended way to manage your own projects is to place them into a separate directory independent of the MUIbase installation directory.

## 3.6 Starting MUIbase

MUIbase can be started either from your graphical environment or from a command shell. On Windows you find MUIbase under the ‘**Start**’ menu. On Mac OS, you launch MUIbase using ‘**Finder**’ from the ‘**Applications**’ directory. If you are running Gnome or KDE on Linux, you can start MUIbase by choosing the corresponding item from the ‘**Applications - Office**’ menu. On the Workbench (Amiga) you double click the MUIbase icon or the icon of a MUIbase project which is then automatically loaded after starting MUIbase (you can also select several MUIbase projects by shift clicking them and double clicking the last one).

When starting MUIbase from a command shell on Windows, Linux or Amiga you can include command line arguments and optional projects to load. There are two basic ways to start MUIbase from a command shell:

```
MUIbase [project1 ...]  
MUIbase -n
```

The first form starts MUIbase and loads the optional projects specified by their filenames *project1* .... The second form starts MUIbase without graphical user interface. This can be useful for running MUIbase as a server in the background and accessing it through its external interface (e.g. ARexx on Amiga, see [Chapter 16 \[ARexx interface\]](#), page 145).

## 3.7 Quitting MUIbase

To quit MUIbase select menu item ‘**Project - Quit**’ or close all opened projects.

## 3.8 Filename Conventions on Windows, Mac OS and Linux

MUIbase for Windows, Mac OS and Linux has a few special conventions for filenames that are usually not found in other software on those systems.

When MUIbase is starting, an environment variable ‘**MUIbase**’ is set referring to the installation directory of MUIbase. On Windows this is the directory specified during the software installation. On Mac OS, it is directory ‘**/Applications/MUIbase.app/Contents/Resources/share/MUIbase**’. On Linux, it is directory ‘**/usr/share/MUIbase**’.

When interpreting filenames (either entered by the user or the filenames present in this documentation) a filename is examined for the occurrence of environment variables. If a filename contains a dollar sign ‘**\$**’ then the characters following it until the first non-alpha-numeric character are interpreted as an environment variable and, if the environment variable is set, the part of the filename is replaced by the contents of the environment variable (otherwise the part of the filename is left unchanged including the ‘**\$**’ sign). You can use parentheses around the environment variable if it contains non-alpha-numeric characters. For example ‘**\$HOME/data**’ expands to the path where ‘**\$HOME**’ has been replaced with the path name of your home directory.

Furthermore, Amiga-style ‘**assign names**’ are handled in the following way. If a filename contains a colon ‘**:**’ then everything before the colon is treated like an environment variable

(called ‘**assign name**’) and is replaced by its contents if the environment variable is set. ‘**Assign names**’ should be at least 2 characters in length, in order to distinguish them from drive letters on Windows.

These conventions allow, for example, to refer to the movie sample database inside the MUIbase installation directory as ‘**MUIbase:demos/Movie.mb**’.

Another example is when you set an environment variable, say ‘**EXTERNAL\_DATA**’, referring to the directory where you store external data like images, etc. of one of your projects. You can then access data inside this directory through ‘**EXTERNAL\_DATA:some-file**’ and store such filenames in the database project.

## 4 Tutorial

The making of a family-tree database.

This chapter contains a brief tutorial describing the basic usage of MUIbase. Within the tutorial a small project is developed that allows the managing of your family tree. The project that results after applying all steps in this tutorial can be found as the ‘FamilyTree.mb’ project located in the ‘Demos’ directory of your MUIbase installation.

### 4.1 How MUIbase Works

MUIbase could be said to operate in two different modes, record-editing and structure-editing mode.

The record-editing mode is where you add, change, or delete records.

The structure-editing mode lets you edit how your database should look like and what tables and attributes it contains.

Besides this there is the program editor where you enter program functions that are executed automatically when entering data or explicitly when pressing a program button.

### 4.2 Starting with a Project, the Structure Editor

To create a database you first need to define it’s contents. In MUIbase this is done in the structure editor which is opened by choosing ‘Structure editor’ in the ‘Project’ menu. Here you find three different sections:

‘Tables’     Add, change, or delete the tables of your project.

‘Attributes’

             Add, change, or delete attributes of the currently selected table.

‘Display’    Specify the graphical layout of your database, i.e. how it should be displayed.

### 4.3 Adding a Table

First we need a table, press the ‘New’ button just below the list view in the ‘Table’ section. You will then see a dialog which asks you to enter some data:

‘Name’        The name of the table. The name must begin with an uppercase letter and can consist of up to 20 characters. The name can be changed later. In this tutorial we set the name to ‘Person’ since it’s going to hold all the person’s names in this database.

‘Number of records’

             A table can either consist of only one or of an unlimited number of records. In our case it should be set to unlimited since we are going to add more than one person.

‘Trigger functions’

             The adding and deletion of records can be controlled by program functions. This is where you set which function to call. Since we don’t have written any program functions yet, we leave the fields empty.

When this is done, just press the ‘OK’ button and we have our first table, ‘Person’.

## 4.4 Adding an Attribute

Then we need a string attribute for that table, press **New** in the attributes section. Attributes also need some settings:

- 'Name'**        The same as for a table: first letter must be uppercase and altogether a maximum of 20 characters. We set the name of this attribute to **'Name'** as it will contain the names of the persons we are about to add.
- 'Type'**        Here we choose what type this attribute should be. There are a couple of different ones but for this attribute we use a string attribute.
- 'Max length'**    Here you can define the maximum number of characters a user can enter for the string. We set this to 30.
- 'Initial value'**    It's possible to have some attributes to use an initial value for every new record you add, here is where you enter what it should contain. Leave this line blank.
- 'Trigger'**    An attribute could also trigger a program function to be executed. For example, if you enter a name then you can have a program to check if this name already exists. We leave this field empty.

## 4.5 Displaying the Project

After pressing **OK** you now should notice some changes in the display section. Change the choice button on the top of the display section to **Main window**. There you see what the main window holds, currently only table **Person**. If you change the choice button back to **Table mask**, you can see how table **Person** is presented. Currently it's displayed as one panel with one attribute.

Now double-click on **Panel(Person)** at the top of the list in the display section and a window should appear, allowing you to set how this panel should be displayed:

- 'Title'**        The title of a table can be different to it's name. Our table is called **Person** but here we could set it to be **THIS IS THE TABLE PERSON!** if we prefer that better.
  - 'Background'**    The background can be changed to whatever suits your taste.
  - 'Gadgets'**    Here we can define what gadgets we want the panel to have.
- Press **OK** and then double-click on **Name** in the list-view in the display section. This brings up the window that contains the settings for how to display the string attribute **Name**.
- 'Title'**        The same as for the panel. The string you enter here is what is really displayed when in record-editing mode.

- 'Shortcut'**    Here you can set a letter that can be used to jump to this attribute, when in record-editing mode. For jumping to the attribute you need to hold down the **Alt** key (Windows, Mac OS and Linux) respectively the **Amiga** key and then press the letter.

- ‘Home’** Makes the cursor to jump to this attribute whenever a new record is added. In our case we will always or most of the time enter the name first in a new record, so set it.
- ‘Read only’** Set this if the attribute should be read only. Leave it unset.
- ‘Weight’** Decides how much of this attribute should be visible when competing for the space with other attributes. For example, if three 50-character strings resides in a window that only has room for 100 characters, then this number decides how much space the string gets relatively to the other ones. Leave it at 100.
- ‘Background’** The same as for the panel.
- ‘Bubble help’** Here you enter any text you think would be helpful to the user. The bubble help appears after holding the mouse still over an attribute for some seconds. Set this to **‘If you need help, call the author at 112’**.

Leave the structure editor (choose **‘Exit structure editor’** in the **‘Project’** menu) and you are back to record-editing mode where you see how the database really looks like. You’ll see the headline which is the string you may have entered in the display section for the panel. The record counter should say **‘#0/0’** since we haven’t added any records yet. Thereafter is a filter button and two arrow buttons. Below all this you should have the **‘Name’** attribute with the text you may have entered in the display section for this attribute. If you haven’t changed any text at all when in the display section then the panel should be named **‘Person’**, and the string attribute **‘Name’**. Move the mouse above the string attribute **‘Name’** and leave it there for a couple of seconds. If you entered something in the bubble help above then this text should appear in a popup window.

## 4.6 Adding two Reference Attributes

Now we will add two reference attributes. Reference attributes are a bit different than other attributes. As their name might imply they refer to records. This will become more understandable as you try it out for yourself in just a while.

Enter the structure editor again and add two more attributes to table **‘Person’**. Press **‘New’** in the attributes section, name it **‘Father’**, and change it’s type to **‘Reference’**. A reference attribute has only one setting:

- ‘Set reference to’** Tells which table the attribute refers to. It should already be pointing to table **‘Person’**. Leave it that way and press **‘Ok’**.

Add another attribute by pressing **‘New’** in the attribute section and call it **‘Mother’**. The attribute’s type should also be set to reference and point to table **‘Person’**.

As you may have noticed, there are now three attributes in the display section. Click on **‘Father’** and then on the buttons up and down located just to the left. This will change where **‘Father’** is positioned when looking at it while in record-editing mode. Put **‘Father’** at the top, **‘Name’** in the middle, and **‘Mother’** at the bottom.



Next, we specify what contents the reference attributes **'Father'** and **'Mother'** should display from the referenced records. Double-click on **'Father'** in the display section and then click on **'Extras'**. Here we choose to display the string attribute **'Name'**, then we press **'Ok'** and repeat the procedure for **'Mother'**.

## 4.7 Adding Records

Now we should add some records. Leave the structure editor and enter record-editing mode. To add a new record you choose **'New record'** from the **'Table'** menu (on Windows and Linux you find this menu by pressing and holding the right mouse button inside the table mask). The cursor should now automatically jump to the attribute we have set to **'Home'** earlier in the display section in the structure editor. Add two records, one with your fathers name in **'Name'** and another one with your mothers name in **'Name'**. Thereafter you add another record with your own name in **'Name'**.

Now it's time to understand those reference attributes. By pressing on the list-view button on **'Father'** we get a list of all records this reference attribute could refer to. Choose your fathers name and do accordingly down below on the mothers list-view.

Now you should have three records, you, your father and your mother. In your record, your fathers name should be visible in **'Father'** at the top and your mothers name should be in **'Mother'** at the bottom. You can browse through the records by pressing **Alt** together with the cursor keys **Up** and **Down**.

But hey! My parents also has/had parents you say! So let's add another four records, the third generation. Just add the records one by one and write their names in **'Name'**. If you don't remember their names then just enter **'father's father'**, **'mother's father'** or something like that instead. Then you browse through all the records and set **'Father'** and **'Mother'** to what they should contain. When this is done you should have at least seven records, your record, your parents records and your grandparents records.

## 4.8 Filter

Since we now have some records to work with, we could try out the filter function. The filter can sort out records you don't want to display, they will still remain in the database but they are just not displayed anymore.

To edit the filter you choose **'Change filter'** from the **'Table'** menu. A window with loads of operators will appear. This is where you set what conditions a record must fulfill to get displayed.

In this small example we will use the **LIKE** command, which lets you do a joker comparison of an attribute. Press once on the **LIKE** button to the right and then double-click on **'Name'** in the list to the left and (**LIKE Name**) should appear in the string just above the **'Ok'** and **'Cancel'** buttons. Thereafter you type **"\*a\*"** so the whole string shows (**LIKE Name "\*a\*"**). This means that MUIbase should display all records that contain the letter **'a'** anywhere in **'Name'**.

Press **'Ok'** and you may notice that records with no **'a'** in **'Name'**, no longer are visible. Since the letter **'a'** is quite common in most languages and names, all records might still be visible but you can try other letters to make the filter function more clear.



As mentioned earlier there is a button on the panel that says 'F'. This 'F' indicates if the filter is on or off. Finally when you're done testing, turn the filter off so that all records are visible.

## 4.9 Queries

Now that we have played with the filter function a bit, we turn to the query feature of MUIbase. Queries can be used to display information from a database matching certain criteria.

Choose 'Queries' from the 'Program' menu to open the query editor. Now a window with some buttons on the top and two larger areas below appear. The string to the upper left is where you enter the name of what you want to call the query you make.

'Run'            Compiles and runs the query.

'Print'        Prints the result of the query to a file or on your printer.

'Load and Save'  
             Lets you load and save each of the queries.

The first large area is where you enter the query. The second large area is where the result is displayed.

Now let's produce a list of all those persons we tried to filter out previously. Type 'Persons with an a in their name' in the string to the upper left. This is the title for this query. In the upper large area, type:

```
SELECT Name FROM Person WHERE (LIKE Name "*a*")
```

Now when you run this query by pressing the 'Run' button it will produce a list of all persons with the letter 'a' in their name. Try changing the letter to see different results.

At this time we can introduce the AND command. Press the list-view button just to the left of the 'Run' button in the query editor. Then choose 'New' and name it 'Persons with both letter a and s in their names'. Then type

```
SELECT Name FROM Person WHERE  
(AND (LIKE Name "*a*") (LIKE Name "*s*"))
```

Note that we are still using the LIKE command for choosing records containing the letters 'a' or 's' in their names, but the AND command requires that BOTH LIKE criteria are met. Therefore, only records with BOTH the letters 'a' and 's' in their name are displayed when the query is run.

## 4.10 Adding a Table with a Memo and a Button Attribute

These were two ways of selecting and displaying the database. Another way of displaying data can be done by a program. In order to display data we can use an attribute type called memo.

Enter the structure editor and press 'New' in the table section. Name the new table 'Control' and set it's number of records to 'Exactly one'. Click and hold down the mouse button on the new table. Now drag it just a bit above the middle of table 'Person' and release the button. In the table section, 'Control' should now be on top, followed by 'Person' below.

Make sure that table ‘Control’ is selected, then press ‘New’ in the attribute section. Set the type of the new attribute to ‘Memo’ and give it the name ‘Result’. Press ‘Ok’ and then add another attribute to table ‘Control’ by once again pressing ‘New’ in the attribute section. This time, set it’s type to ‘Button’ and name it ‘Pedigree’.

To give the database a better look, click once on ‘Pedigree’ in the display section and push it to the top by pressing the ‘Up’ button once.

## 4.11 Programming MUIbase to do a Pedigree

Now we have a button that can start a program and a memo to display data in. It’s therefore time to enter the program editor. This is done by choosing ‘Edit’ from the ‘Program’ menu. The editor has three buttons:

‘Compile & Close’

Which does just that. It compiles the program and leaves the program editor.

‘Compile’ Compiles the programs but stays in the program editor.

‘Revert’ Reverts all changes back to the state when you entered the program editor.

As all program functions you write will reside in this one window, we will need to separate them from each other. In MUIbase this is done by the DEFUN command. Everything between the two parenthesis will be part of the function `pedigree` in this example:

```
(DEFUN pedigree ()
```

```
  ; This is DEFUN’s end parenthesis
)
```

With this in mind we now type in the first function which will produce a family tree of the current person in the database and place the result in attribute ‘Result’. This `pedigree` function is in fact three functions:

- `pedigree` which sets ‘Control.Result’ by calling another function.
- `getpedigree` which collects the pedigree into a nested list.
- `pedigree2memo` which converts this list to the memo.

```
  ; The program pedigree
```

```
(DEFUN pedigree ()
```

```
  (SETQ Control.Result (pedigree2memo (getpedigree Person NIL) 0 3))
)
```

```
  ; The program getpedigree
```

```
(DEFUN getpedigree (person:Person level:INT)
```

```
  (IF (AND person (OR (NULL level) (> level 0)))
```

```
    (LIST person.Name
```

```
      (getpedigree person.Father (1- level))
```

```
      (getpedigree person.Mother (1- level))
```

```
    )
```

```

    )
  )

; The program pedigree2memo

(DEFUN pedigree2memo (pedigree:LIST indent:INT level:INT)
  (IF (> level 0)
    (+
      (pedigree2memo (NTH 1 pedigree) (+ indent 8) (1- level))
      (IF pedigree (SPRINTF "%s%s\n" indent "" (FIRST pedigree)) "\n")
      (pedigree2memo (NTH 2 pedigree) (+ indent 8) (1- level))
    )
    ""
  )
)

```

When typing this program, make sure that all the parenthesis are where they should be. Too many or too few parenthesis is the most common error you get when MUIbase is compiling your program. The error message from MUIbase probably is ‘Syntax Error’ in this case. Press ‘Compile & close’ and the window closes, which means that there weren’t any mistakes during compiling.

Don’t worry too much if you don’t understand all the functions at first. There is a complete chapter (see [Chapter 15 \[Programming MUIbase\], page 70](#)) containing a reference of all functions including their detailed descriptions.

Now we have a program to run, but first we have to assign the program function to the ‘Pedigree’ button. This is done by entering the structure editor, selecting ‘Control’ in the table section and double-clicking on the ‘Pedigree’ attribute in the attribute section. Then open the list-view ‘Trigger’. In this list, all your program functions will be listed, currently there should be three functions: `pedigree`, `getpedigree` and `pedigree2memo`. Double-click on `pedigree` as the program function the ‘Pedigree’ button will trigger, then press ‘Ok’ and leave the structure editor.

Now if everything is done correctly, pushing the ‘Pedigree’ button will produce a pedigree of the current person. Try changing person to see some different pedigrees.

## 4.12 Programming MUIbase to List a Person’s Children

As the next addition to this database requires some more records, you should add your brothers and sisters. If you don’t have any then write ‘My faked sister 1’, ‘My faked brother 1’ which of course should be set to have the same parents as you have.

Then go to the program editor and type the following for creating another program.

```

; The program children counts the number of children of a person.
; First we define the variables, e.g. "names" holds the children’s names.

(DEFUN children ()
  (LET ( (names "") (count 0) (parent Person) )

```

```

; For all records in table Person do the following:
; If the variable parent appears as father or mother
; in any of the records then:
;     add the name to the variable names
;     increase count by 1.

(FOR ALL Person DO
  (IF (OR (= parent Father) (= parent Mother))
    (
      (SETQ names (+ names Name "\n"))
      (SETQ count (1+ count))
    )
  )
)

; Then we write the result into the Control.Result.
; If the current person doesn't have any children, write one string.
; If he/she has children then write another string.

(SETQ Control.Result
  (+ Person.Name (IF (> count 0)
    (+ " is the proud parent of " (STR count) " children.")
    " does not have any children (yet :-).")
  ))
)

; If the parent has children then add their names.

(IF (<> count 0)
  (SETQ Control.Result
    (+ Control.Result "\n\n"
      (IF (= count 1)
        "The child's name is:"
        "The children's names are:")
      )
    "\n\n"
    names
  )
)

; This is the end parenthesis of the LET-command.
)

; This is the end parenthesis of DEFUN children.
)

```

To create variables, we use the **LET** command. Variables created with the **LET** command are local and only visible within the '**LET**' command's open and closing parentheses. Thus any command that accesses these variables has to be placed within these parentheses.

All we need to execute this program is a new program button. Therefore, enter the structure editor and add a button attribute in table '**Control**'. Call it '**Children**' and choose **children** as the program function it should trigger.

To bring some order in the mask of table '**Control**' it's now time to introduce groups. All objects can be ordered into vertically or horizontally aligned groups.

In the display section, click on '**Pedigree**' and shift-click on '**Children**', thereafter you click on the '**Group**' button to the left. Now the two program buttons will be together in a vertically aligned group. However, we want this one to be horizontally aligned so double-click on the '**VGroup**' that has appeared in the display section. This will open a window that lets you change the settings for this group. Set the title to '**Programs**' and check the '**Horizontal**' button.

At this time we can hide the name of attribute '**Result**' in table '**Control**'. Double-click on '**Result**' in the display section and empty the title field. This will display attribute '**Result**' without any title string.

To make things easier, if we add more programs or attributes in table '**Control**', we should place '**Result**' and the '**Programs**' group into a vertical group. Be sure that you have only marked '**Programs**' and '**Result**' then press '**Group**'. This places '**Programs**' and '**Result**' into a vertical group.

Leave the structure editor and take a look at the result. Then press the '**Children**' button to see the number of children and their names of the current person.

This example could very well be extended into a fully-grown pedigree program. The only real limits are your fantasy and the size of your hard drive.

## 5 Basic Concepts

Before you start setting up your own database projects and entering data for them, you should now about some basic concepts MUIbase is built on.

### 5.1 Projects

A MUIbase project consists of all relevant information you need for managing your data. This includes the project's user interface, the project's data you entered and the programs you wrote for the project.

A project can be loaded from, saved to, and deleted from disk. Any change you make to a project is only done in memory. At any time you can go back to the state of the last saved project by reloading it.

MUIbase is able to handle multiple projects at a time. Therefore it is not necessary to start MUIbase twice just to load another project.

### 5.2 Tables

MUIbase manages data in tables. A table is organized in rows and columns, where rows are called records and columns are called attributes.

See the following table for an example on how to structure a set of addresses in a table.

Name	Street	City
-----	-----	-----
Steffen Gutmann	Wiesentalstr. 30	73312 Geislingen/Eybach
Charles Saltzman	University of Iowa	Iowa City 52242
Nicola Müller	21W. 59th Street	Westmont, Illinois 60559

There exists a special table kind which can hold exactly one record. A table of this kind is sometimes useful for controlling the database project, e.g. you can put buttons into this table for executing various actions, or a read-only attribute for displaying project related information. For an example suppose you have an account database where you store all your income and expense. An exactly-one table now could have a read-only attribute of type real for displaying the total balance.

Each table has two record pointers, a pointer to the record that is currently displayed in the user interface (called *GUI record pointer*) and a pointer to the record that is the current one while executing a MUIbase program (called *program record pointer*).

You can define any number of tables for a MUIbase project. Tables can be added to, renamed, and deleted from a project.

### 5.3 Records

A record is one row of a table. It holds all information about one set, e.g. in a table that manages addresses, one record holds one address.

Each record has a record number that reflects the record's position in the table. This number may change when you add or delete records.

For each table a record called *initial record* exists that holds the default values for initializing new records. The initial record always has a record number of 0.

Records can be added to, changed, and deleted from a table. The records are not necessarily held in memory but are loaded from and stored to disk when needed. There are two restrictions for the total number of records of one table. The record number is a 32bit integer value, which (theoretically) limits the total number of records to 4294967295. Another (and more severe) limitation is that for each record a small record header is kept in memory. These limitations still make MUIbase usable for record numbers of 10,000 and more.

## 5.4 Attributes

An attribute defines one column of a table. It specifies the type and appearance of the corresponding column.

Attributes can be added to, renamed, and deleted from a table. There is no upper limit on the number of attributes per table.

For each attribute you have to specify a type that restricts the contents of this attribute. See the next section for a list of available attribute types.

## 5.5 Attribute Types

Attributes can be of type string, integer, real, Boolean, choice, date, time, memo, reference, virtual, or button. The types are described in more detail below.

Some of the attribute types support a special value called NIL. This special value has the meaning of an undefined value, e.g. for a type of date it means an unknown date. The NIL value is similar to the NULL value in other database systems.

Please note that once you have set the type of an attribute, you cannot change it later.

### 5.5.1 String Attributes

String attributes can store any single line of text. Strings are the most often used attribute type in a database project. For example an address database could store the name, street, and city of a person each in its own string attribute.

For a string attribute you have to specify the maximum number of characters allowed in the string. This number does not directly affect the amount of memory or disk space that is used by this attribute because only the actual string contents are stored (other databases have called this feature compressed strings). If necessary the number can be changed after you have installed a string attribute.

String attributes can also be used to store font- and filenames. For filenames external viewers can be launched to display the file contents. Furthermore an in-line image class allows displaying the image of a file.

String attributes do not support the NIL value.

### 5.5.2 Integer Attributes

Integer attributes store integral values in the range of -2147483648 to 2147483647. They are mostly used for storing quantities of any kind, e.g. the number of children of a person, or the number of song titles on a CD.

Integer attributes support the NIL value representing an undefined integer value.

### 5.5.3 Real Attributes

Real attributes store floating point values in the range of -3.59e308 to +3.59e308. They are used for storing numbers of any kind, e.g. the amount of money in an income/expense project.

For each real attribute you can specify the number of decimal places used for displaying the real value, though internally always the full precision is stored.

Real attributes support the NIL value representing an undefined real value.

### 5.5.4 Boolean Attributes

Boolean attributes store one bit of information. They are used for storing yes/no or true/false values, e.g. in a project managing invoices a Boolean attribute could store the 'has paid?' information.

Boolean attributes use TRUE and NIL as Boolean values. NIL in this case stands for a value of FALSE.

### 5.5.5 Choice Attributes

Choice attributes store one item out of an enumeration of items. For example, in an address project a choice attribute may be used for storing a country name, where the country is one out of 'USA', 'Canada', 'Germany', or 'others'.

A choice attribute does not store the item string but the item number (index) in a record. The number of items and the items itself can be modified after the attribute has been created. However when making changes to a choice attribute, values in existing records are not changed to reflect the new labeling.

Choice attributes do not support the NIL value.

### 5.5.6 Date Attributes

Date attributes store calendar dates. For example, a date attribute can be used for storing birthdays.

The format for entering and displaying date values can be one of 'DD.MM.YYYY', 'MM/DD/YYYY', or 'YYYY-MM-DD', where 'DD', 'MM' and 'YYYY' are standing for two and four digit values representing the day, month and year of the date respectively.

Date attributes support the NIL value representing an undefined date.

### 5.5.7 Time Attributes

Time attributes store the time of day or a period of time. For example, a time attribute can be used for storing the durations of music titles on a CD.

The format for entering and displaying time values can be one of 'HH:MM:SS', 'MM:SS' or 'HH:MM', where 'HH' represents the hours, 'MM' the minutes, and 'SS' the seconds. Internally time values are stored as the number of seconds since 0 AM. Time values larger than 23:59:59 up to the maximum value of 596523:14:07 are possible but negative values are not supported.

Time attributes support the NIL value representing an undefined time.



### 5.5.8 Memo Attributes

Memo attributes store multi-line text of any size. Text size is handled dynamically which means that memory is only allocated for the actual text size. In a project managing movies for example, a memo attribute can be used to store summaries of the movies.

Memo attributes do not support the NIL value.

### 5.5.9 Reference Attributes

Reference attributes are a special type of attribute, normally not found in other database systems. Reference attributes store a pointer to another record. The referenced record may reside in the same or in any other table the reference attribute belongs to.

For example in a pedigree project two reference attributes can be used for storing pointers to the father and mother record. Or in a project managing CDs and music titles, a reference attribute in the table holding the music titles can be used to point to the records of the corresponding CDs.

For displaying a reference attribute, any attributes of the referenced record can be specified. Entering a reference attribute can be done by selecting a record from a list of records.

Reference attributes support the NIL value. Here a value of NIL stands for a pointer to the initial record of the referenced table.

### 5.5.10 Virtual Attributes

Virtual attributes do not store any information in the database itself, but compute them on the fly when needed.

For example, in a project managing invoices where a real attribute holds the amount of money excluding tax, a virtual attribute can be used to "store" the amount of money with tax. Every time the value of the virtual attribute is needed, e.g. for displaying it, it is computed from the corresponding value without tax.

For displaying virtual attributes three kinds exists: Boolean, string and list. These three kinds allow showing the value of the virtual attribute as a TRUE/FALSE value, as a single line of text including numbers, dates, and times, or as a list of several single lines, e.g. for listing all music titles of a CD.

Virtual attributes support the NIL value standing for FALSE (Boolean kind), undefined (string kind), or empty (list kind).

### 5.5.11 Buttons

Actually, buttons are not a real attribute type as they cannot store or display any information. Buttons are just used for triggering MUIbase programs.

## 5.6 Table of Attribute Types

The following table summarizes all available attributes types:

Type	Description	Nil allowed?
String	For strings of lengths 1..999. A string can also be used for storing	No

filenames, font-names or one-string-out-of-n-strings. For filenames you can add a field where the contents of the file are displayed as an image.

Integer	For storing integer values.	Yes
Real	For floating point numbers.	Yes
Boolean	TRUE or NIL.	Yes (NIL = FALSE)
Choice	One number out of n numbers. Numbers are represented by label strings.	No
Date	For storing a date value (1.1.0000 - 31.12.9999).	Yes
Time	For storing time values (00:00:00 - 596523:14:07)	Yes
Memo	Multi-line text of unlimited length.	No
Reference	For storing a reference to a record of another table.	Yes (NIL means initial record)
Virtual	For displaying results from a MUIbase program.	Yes
Button	For triggering a program function	No (N/A)

## 5.7 Memory Consumption

Each attribute type needs a certain amount of memory for storing one value in a record. All types except virtual and button have in common that they need a 2 bytes header holding internal information. Additionally, type dependent space is needed for storing the actual value. The following table lists how much memory including a possible 2 byte header one value of the given type needs in memory and on disk.

Type	Memory space	Disk space
String	$2 + 4 + \text{string-length} + 1$	$2 + \text{string-length} + 1$
Integer	$2 + 4$	$2 + 4$
Real	$2 + 8$	$2 + 8$
Boolean	$2 + 0$	$2 + 0$
Choice	$2 + 2$	$2 + 2$

Date	2 + 4	2 + 4
Time	2 + 4	2 + 4
Memo	2 + 4 + <i>memo-length</i> + 1	2 + <i>memo-length</i> + 1
Reference	2 + 4	2 + 4
Virtual	0	0
Buttons	0	0

Here *string-length* stands for the length of the string to be stored and *memo-length* for the text size of the memo to be stored.

## 5.8 Relationships

Up to now you know how to organize your information into tables with records and attributes. But you may also want to setup relationships between tables.

For example if you want to collect CDs in a database project you would have two tables, one for the CDs themselves and one for the music titles of the CDs. Of course you could also have all music titles within the CD table but then you would have a fixed number of music titles for each CD.

So having these two tables, you now need a link for each music title to the CD containing this title. This is called a *relationship* between the two tables. In MUIbase you use a reference attribute for setting up such a relationship.

By installing a reference attribute into a table you automatically have a relationship between the table the attribute resides in and the table it refers to.

### 5.8.1 One to one Relationships

One to one relationships are very simple relationships where for each record you have one or zero partners in another or in the same table.

For example in a database project that manages your favorite movie actors you could setup a reference attribute called ‘**married with**’ that shows the person the actor is married with. An actor that is currently not married does have a NIL value for this reference field.

Of course, no one prevents the user to set the ‘**married with**’ references of several actors all to the same person. However by programming MUIbase it is possible to detect such cases and handle accordingly.

### 5.8.2 One to many Relationships

One to many relationships are useful for connecting a set of records to one record in another or the same table.

For example in a project managing your bank accounts you could have one table for all bank accounts and one table for all transactions. Now you surely want to know which transaction belongs to which account so you setup a reference attribute in the transaction table referring to the account table.

One to many relationships are the most often used ones. You can use them for managing any hierarchical-like structures, e.g. CDs with music titles, bank accounts with transactions, family trees, etc.

One to many relationships are also the basis for realizing many to many relationships as described in the next section.

### 5.8.3 Many to many Relationships

Many to many relationships are used when you want a set of records to refer to another set of records.

For example in a project that manages movies and actors you would have two tables, one for the movies and the other one for the actors. Now for each movie you want to know the actors that took part in the movie. So you might think to setup a reference attribute in the actor table that refers to the movie table. But when doing this you could only have one movie referenced for each actor because there is only one reference field in the actor table. So what you need is an unlimited number of references from the actor table to the movie table.

This is done by adding a new table that just has two reference attributes, one pointing to the actor table and the other to the movie table. Now you can enter the relationships by adding new records to this table. For each movie-actor constellation you add a new record and specify the movie and actor by setting the corresponding reference fields.

If you want to know in which movies an actor took part then you only have to search for all records in the new table that refer to the actor in question and look at the movie records the found records refer to. Such a search can be done automatically by MUIbase and the result can be displayed in a list-view.

The following tables show an example of how to connect a set of actors to a set of movies.

	Title	Country
-----		
m1:	Batman	USA
m2:	Batman Returns	USA
m3:	Speechless	USA
m4:	Tequila Sunrise	USA
m5:	Mad Max	Australia
m6:	Braveheart	USA

	Name
-----	
a1:	Michael Keaton
a2:	Jack Nicholson
a3:	Kim Basinger
a4:	Danny DeVito
a5:	Michelle Pfeiffer
a6:	Geena Davis
a7:	Christopher Reeve
a8:	Mel Gibson
a9:	Kurt Russell

```

a10:  Sophie Marceau
a11:  Patrick McGoochan
a12:  Catherine McCormack
a13:  Christopher Walken

```

MovieRef	ActorRef
-----	
m1	a1
m1	a2
m1	a3
m2	a1
m2	a4
m2	a5
m2	a13
m3	a1
m3	a6
m3	a7
m4	a8
m4	a5
m4	a9
m5	a8
m6	a8
m6	a10
m6	a11

From these tables you can find out for example that Mel Gibson took part in the movies Tequila Sunrise, Mad Max, and Braveheart, or that in movie Batman the actors Michael Keaton, Jack Nicholson, and Kim Basinger took part.

## 5.9 User Interface

MUIbase uses a graphical user interface (GUI) organized in a hierarchical way for displaying record contents and for entering data. Each project owns its own main window in which further GUI elements (including sub windows) can be placed. The GUI elements are also called *display objects*.

A table is displayed in an own GUI element called *mask*. A mask can display only one record at a time. Its layout and the attributes included in the mask are customizable by the user.

The following sections describe MUIbase' GUI elements for designing a project's layout.

### 5.9.1 Windows

Windows can be used to spread information of a project across several independent areas.

Each project automatically has its own main window. If needed, e.g. if the space of the main window exceeds, additional sub windows can be created. Sub windows can also have further sub windows.

For each sub window a window button can be placed into the parent window allowing to open the sub window. The window button looks like a normal text button but may show a small icon to distinguish itself from other buttons.

Main windows do not have a parent window and therefore have no window button. Closing a main window means closing the whole project.

A window can have other GUI elements as children. If no child has been added to a window then a default image is shown.

### 5.9.2 Masks

A mask is used to display the contents of a table. Only one record of the table can be shown at a time.

The mask may include a panel (see next section) for controlling the table. Other GUI elements like attribute or text objects can be placed into a mask to show the record contents.

Masks cannot be placed inside other masks as this would lead to a hierarchy of masks and therefore to a hierarchy of tables. To setup a hierarchy of tables, use an 1:n relationship between the two tables.

### 5.9.3 Panels

A panel is a small and wide rectangular area placed at the top edge of a mask. A panel can display a title, e.g. the name of the corresponding table, a pair of numbers showing the record number of the current record and the total number of records, and several buttons for controlling the table, e.g. for displaying the next or previous record.

Only one panel can be defined for a mask. If you setup a panel for a mask then an additional border is drawn around the mask, otherwise no border is shown.

### 5.9.4 Attribute Objects

Attribute objects are used to display the contents of one item of a record.

Depending on the type of the attribute the GUI element is either a string field (types string, integer, real, date and time), a check-mark button (type Boolean), a cycle button or a set of radio buttons (type choice), an editor field (type memo), a pop-up list-view (type reference), a text, check-mark, or list-view field (type virtual) or a text or image button (type button). In some cases the GUI element may also be a simple text field if the attribute object is set to read-only.

### 5.9.5 Text Objects

Text objects are used for describing the various field elements of a record mask or just to display some static text.

### 5.9.6 Images

Images can be displayed anywhere in a window. An image can be a pattern, a simple color field, or a pictures in an external file. The image size can be resize-able or fixed.

The image is static. For storing images in a table you use a string attribute (see [Section 5.5.1 \[String type\], page 19](#)).

### 5.9.7 Space Objects

Space objects are used to insert space in the layout of a window or a table mask. A space object can have a vertical (or horizontal) bar for delimiting other GUI elements.

### 5.9.8 Groups

GUI elements can be grouped into horizontal or vertical groups. A group places its children from left to right (horizontal group) or from top to bottom (vertical group).

A group can surround its child objects with a rectangular frame, can display an optional title at the top of the group, and can insert space between its child objects.

### 5.9.9 Balance Objects

Balance objects can be placed between other child objects of a window, mask, or group object. A balance object allows the user to control the weight values of the other child objects and therefore how much space each child gets.

### 5.9.10 Register Groups

A register group can be used to layout GUI elements into several pages where only one page is visible at a time. This is useful if the user interface becomes too large and you don't want to spread it over several windows.

## 6 Managing Projects

In this chapter you find how MUIbase organizes projects, how to open, save, delete, and close them, how to check the data integrity and how swapping of records works.

### 6.1 File Format

A MUIbase project consists of several files stored in a directory owned by the project. This directory is first created when you save the project. Do not make any assumption about the directory structure or filenames inside the directory. In particular, do not remove or place any files or further directories in this directory! They will get lost when reorganizing the project.

The directory contains a file called `Structure.mb` where the descriptions of all tables, attributes, filters, etc. are stored. The record headers are also placed here. For each table you will find a file with the table's name. Here all records of a table are stored. Last but not least there is a file called `.lock`. This file is used for locking a project, that is, MUIbase first locks this file exclusively and then opens other files. If the locking fails, MUIbase knows that there is already a MUIbase application working on this project. Only one MUIbase application is allowed to work on a project at a time since record files are opened in read/write mode and we certainly do not want to have data mixed together by write operations of two or more applications.

### 6.2 Info

MUIbase keeps some information about each project. Select menu item `Project - Info` to get information about the current project. The information you get consists of the project's name, the number of tables, the total number of records in all tables, and a value that shows how many bytes a reorganization of this project would gain. The gain is however only a rough estimate and should not be treated as an exact number. Especially if you have made many changes to the structure of the project (adding or removing attributes) then this value is far from accurate.

### 6.3 New Project

MUIbase can handle any number of projects at a time. You are only limited by the available memory. In order to start another project, choose menu item `Project - New`. This opens a new window with an empty project. You can now define the structure of this project (see [Chapter 14 \[Structure editor\], page 56](#)) or load an existing project from disk (see [Section 6.5 \[Open project\], page 29](#)).

### 6.4 Clear Project

To reset a project select menu item `Project - Clear - Project`. This closes the current project and replaces it by an empty project. After starting MUIbase without projects you arrive in this state automatically.

By selecting menu item `Project - Clear - Records` you start a new project using the structure of the current one. This means that all except the record data of the current project is used for the new project.



If the current project at the time you selected one of above menu items has not been saved to disk then a safety requester appears asking for confirmation of the operation.

## 6.5 Open Project

To load a project select menu item **‘Project - Open - Project’**. This opens a file-requester where you can choose a project from. It's also possible to only load the structure of a project, that is, the whole project is loaded except the record data. To do so, select menu item **‘Project - Open - Structure’**.

If the loaded project has its program source set to external, the external source file is created after opening the project (see [Section 15.2 \[External program source\]](#), page 70).

If you were editing a project at the time of choosing any of the above menu items and the project has not been saved then a safety requester appears asking for confirmation.

## 6.6 Save Project

All changes you make to a project are only done in memory or stored temporarily when swapping records (see [Section 6.9 \[Swap records\]](#), page 30). Thus if you want to make them permanent you have to save the project to disk. This is done by choosing menu item **‘Project - Save’**. If your project doesn't have a name yet then a file-requester asking for a filename appears first.

The reason why MUIbase does not automatically save a project when it is changed, is that, this way, it is you who decides when to save a project! You can always go back to the last saved version of your project by choosing menu item **‘Project - Revert to saved’**. This mechanism is similar to the **‘COMMIT’** and **‘ROLLBACK’** commands in SQL database systems.

If you save a project, all modified records are written to disk and the file **‘Structure.mb’** is recreated. Before creating the new **‘Structure.mb’** file, MUIbase first renames a possibly existing **‘Structure.mb’** file to **‘Structure.old’** to have a safety copy in case the save operation fails.

This mechanism guarantees fast load & save operations but it is not free of reorganization. If you have modified many records then the physical place where the records lie and the resulting fragmentation may become disadvantageous. Therefore a menu item **‘Project - Save & reorg’** exists that does a save & reorganize operation. This operation may take some time depending on the number and size of the records. The save & reorganize operation creates a new directory and rewrites all project related files. The old directory is deleted on success.

Another good time to schedule a reorganization is when you have done changes to the data-structure of a project, e.g. after you have installed a new attribute in a table. These changes are not applied immediately to all records because it would take too much time to load each record, modify it, and save it back to disk. Therefore these changes are put on an internal **‘todo’** list which is applied after loading a record. Applying this list to a record takes only little time. However the longer the list gets the more time it needs. Reorganizing a project causes the **‘todo’** list to be applied to all records, so if you have made many changes to the project structure then reorganizing a project will shorten the time for loading records.

You can also save & reorganize a project to a new filename keeping the old project untouched. To do this select menu item **‘Project – Save & reorg as’** which prompts you to enter a new name for the project.

## 6.7 Admin and User Mode

MUIbase operates either in admin mode (default) or in user mode. You can change between these modes by choosing menu item **‘Project – Change to admin mode’** and **‘Project – Change to user mode’**. When in user mode, several menu items are disabled and structure, program and query editor are not available. Therefore, only basic record editing is possible. In admin mode all operations are permitted.

An admin password can be set for a project by selecting menu item **‘Project – Change admin password’**. Once set, the password has to be entered when changing into admin mode, or permission is denied leaving the project in user mode.

When opening a project that has an admin password set, the project is started in user mode, otherwise (no admin password has been set), it is started in admin mode.

## 6.8 Check Data Integrity

MUIbase can verify if all data in your project is still valid and not corrupted by programs trashing your project files or by system crashes. Choose menu item **‘Project – Check data integrity’** for starting this process.

Usually you should never need this functionality and MUIbase should always report that data integrity is perfect. But if it ever happens to you that your project contains **‘internal errors’**, that is, some records can’t be loaded anymore, then you can repair your project by using this menu item.

MUIbase will write a log file of all affected records and you can save and reorganize the project afterwards. In the log file, records that might be corrupted and records that are not accessible anymore and therefore have been deleted are listed by their record number in the old (corrupted) project and by their record number (in parenthesis) in the reorganized project.

## 6.9 Swap Records

MUIbase doesn’t need to keep all records of a project in memory. Thus loading and saving of projects is much faster. When loading a project, a record header is allocated for each record. The data itself is only loaded when needed, e.g. when it is displayed on the screen. The total number of records is still limited by available memory since each record header needs some few bytes of memory.

You can specify how much memory MUIbase should use for the records of a project. Choose one of the predefined values found in menu item **‘Preferences – Record memory’** (see [Section 7.2.1 \[Record memory\]](#), page 34). MUIbase does not preallocate a block of the specified memory size, it only checks from time to time if the current size of allocated memory is larger than the specified value.

If MUIbase runs out of memory or if the upper limit for the record memory size has been reached then MUIbase tries to free as much record memory as possible. In this case

MUIbase may write modified records to disk to get the maximum available memory possible. You can also force MUIbase to do this by choosing menu item **‘Project – Swap records’**.

MUIbase maintains a free list for each record file. If you delete a record then the record’s file space is added to the free list. Also if you change a record and the record needs to be written to disk then the old file space is added to the free list. However MUIbase makes sure that by reloading you can always go back to the point of the last save operation. MUIbase will not write into areas which are free but where a record still exists that could be reached by reopening the project.

## 6.10 Close Project

When you are done editing a project you can close it by selecting menu item **‘Project – Close’**. This frees the memory and all resources belonging to the project. If the project contains changes that have not been saved yet then a safety requester appears first offering to save, continue or cancel the operation.

For closing a project you can also select menu item **‘Project – Save & close’** which saves the project first if there were any changes and then closes it.

## 7 Preferences

MUIbase offers several preferences items the user can set to his likes. This chapter shows what preferences items are available and gives general information about how the preferences system works.

All preferences items can be divided into user settings and project settings.

### 7.1 User Settings

The user settings contain preferences items that depend on the choice of a user, e.g. the user's language, country or taste. The user settings are shown and can be customized in the top part of menu 'Preferences'.

User settings are stored in the environment of the user. On Windows, Mac OS and Linux they are kept in the home directory of the user as '`.MUIbase.prefs`'. On the Amiga they are stored under '`ENV:MUIbase.prefs`' and '`ENVARC:MUIbase.prefs`'.

The user settings contain the preferences items listed below. Whenever any of these items is changed in the 'Preferences' menu, the settings are written to disk in order to make them permanent.

#### 7.1.1 Formats

By selecting menu item '**Preferences - Formats**' you can specify the formats used when displaying or printing real and date values. After selecting the menu item a new window appears containing the following items:

- a field '**Real format**' for setting the decimal character of real values. You can choose between '**Decimal point**' and '**Decimal comma**'.
- a field '**Date format**' for specifying how date values are output. You can choose between '**Day.Month.Year**', '**Month/Day/Year**' and '**Year-Month-Day**'.
- two buttons '**Ok**' and '**Cancel**' for leaving the window.

The initial values for real and date formats are determined according to the information found in the operating system's '**locale**' environment.

When you are done with all settings, press the '**Ok**' button to leave the window and update the display.

#### 7.1.2 External Editor

Besides providing an internal editor built into MUIbase through the employed GUI toolkit, MUIbase also offers to edit text contents by using an external editor (e.g. the context menu of the built-in editor includes an item that allows to call the external editor for editing the contents). The name of the editor and its parameters are specified by selecting menu item '**Preferences - External editor**'. You should enter a command string that gets executed when calling the external editor. The special string '**%f**' can be used at the place of the filename and is replaced with the actual filename (or a temporary filename MUIbase creates for exchanging the text data) before execution.

For example, on Linux you can specify '**emacs %f**' for using the powerful GNU Emacs editor, or on the Amiga, you can use '**CED %f -keepio**' for switching to the famous CED editor as external editor.

Default is ‘Notepad %f’ on Windows, ‘open -t %f’ on Mac OS, ‘gvim -f %f’ on Linux, and ‘Ed %f’ on the Amiga.

Note that on Mac OS, the default setting always starts the external editor asynchronously, i.e. MUIbase never waits for the editor to quit. If you are running Mac OS 10.5 or higher then you can use ‘open -tWn %f’ for enabling synchronous editing. This is required when using menu item ‘**External editor**’ in the context menu of a Memo attribute (see [Section 8.3 \[Changing records\], page 38](#)) and when using the EDIT command for programming MUIbase (see [Section 15.23.1 \[EDIT\], page 131](#)).

On the Amiga, additionally the sequence ‘%p’ can be used in the command string. When calling the external editor this sequence is replaced by the name of the public screen MUIbase is running on.

### 7.1.3 External Viewer

MUIbase uses an external viewer for displaying the contents of external files, e.g. images, movies or any kind of documents. For example you can use the string data type for storing filenames in a table and then let MUIbase display them using the external viewer. To specify this viewer select menu item ‘**Preferences - External viewer**’. Like for the external editor (see [Section 7.1.2 \[External editor\], page 32](#)) you have to enter a command string here.

Default is ‘%f’ on Windows (using ShellExecute), ‘open %f’ on Mac OS, ‘gnome-open %f’ on Linux, and ‘Multiview %f’ on the Amiga.

Note that on Mac OS, the default setting always starts the external viewer asynchronously, i.e. MUIbase never waits for the viewer to quit. If you are running Mac OS 10.5 or higher then you can use ‘open -Wn %f’ for enabling synchronous viewing. This is required when using the VIEW command for programming MUIbase (see [Section 15.23.3 \[VIEW\], page 131](#)).

On the Amiga, additionally the sequence ‘%p’ can be used in the command string. When calling the external viewer this sequence is replaced by the name of the public screen MUIbase is running on.

### 7.1.4 Extra Buttons in Tab Chain

In the graphical user interface specified in the structured editor there might exist several extra buttons. Under extra buttons fall popup buttons, e.g. file, font or listview popups beneath a string field, auto-show and filter buttons of a reference field, and view buttons next to a string field.

These buttons are usually not included in the cycle chain, that is, you can’t use the *Tab* key to activate them. However if you check menu item ‘**Preferences - Extra buttons in Tab chain**’ then these buttons are also included in the Tab chain. Default is checked.

On the Amiga please note that changing the status of this menu item has only an effect after rebuilding the user interface, e.g. by switching to the structure editor and back to the user interface.

### 7.1.5 Advance on Enter

When the cursor is in an editable single-line text field in the user interface of a project, pressing the *Enter* key can either advance to the next item in the user interface or keep the cursor in the current text field.

If you check menu item ‘**Preferences - Advance on <Enter>**’ then the cursor jumps to the next item in the user interface, otherwise it stays on the current text field. Default is checked.

Note that in any case, pressing **Enter** confirms and stores the entered text.

### 7.1.6 Confirm Quit

If you try to quit MUIbase and there are unsaved projects then a safety requester pops up asking for confirmation. However if all projects have been saved, MUIbase quits silently.

In order to always enforce a requester when quitting MUIbase, menu item ‘**Preferences - Confirm quit**’ has to be checked. In this case you always get a safety requester when selecting menu item ‘**Project - Quit**’ or when closing the last open project. Default is unchecked.

### 7.1.7 MUI

Amiga only. As MUIbase on the Amiga is a MUI application, you can also specify the MUI preferences for this application by choosing menu item ‘**Preferences - MUI**’.

## 7.2 Project Settings

Project settings consists of preferences items that are stored along with a project. These items are shown and changeable in the lower part of menu ‘**Preferences**’ and in menu ‘**Program**’ (the reason to spread these among two menus is to keep menus compact and clear).

The following preferences items belong to the project settings. Whenever any of these settings is changed, the count of changes for the project is incremented.

### 7.2.1 Record Memory

MUIbase does not need to keep all records of a project in memory. Instead it uses a buffer for holding only a small number of records. By choosing a value from menu item ‘**Preferences - Record memory**’ you can set the size of this buffer. Each project has its own buffer, so if you have opened two projects each having a record buffer size of 1MB, MUIbase will use up to 2MB for the records of both projects.

MUIbase does not allocate the memory a priori but uses a dynamic allocation scheme. Furthermore, the specified buffer size is only a soft limit and occasionally MUIbase might exceed it.

Once the buffer gets full, or if MUIbase runs out of memory, all records are flushed from the buffer. This means that unchanged records are simply freed and changed records are first written to disk and then freed (see [Section 6.9 \[Swap records\], page 30](#)).

By giving MUIbase a higher value for the record buffer you can notice a speed increase in accessing the records as more records can be held in memory and fewer disk access is needed. If there is enough memory to hold all records in memory and if you have specified a large enough memory limit (e.g. ‘**unlimited**’) then MUIbase operates with optimal speed.

Default is ‘**unlimited**’.

### 7.2.2 Record Delete Requester

You should check menu item **‘Preferences - Record delete requester’** if you want MUIbase to pop up a safety requester asking for confirmation whenever you try to delete a record. Leave the item unchecked if records should be deleted silently. Default is checked.

### 7.2.3 Paths Relative to Project

If you want to reference external data (e.g. documents or images) then the filename of these data need to be stored in the project. For this purpose you can use absolute path names or path names containing assign names (see [Section 3.8 \[Filename conventions\]](#), page 7). Another way is to store external data relative to the directory of the project (note: this is not the project directory itself but the directory containing the project).

By activating menu item **‘Preferences - Paths relative to directory of project’**, MUIbase changes the current working directory to the directory containing the project. This means that if you have several projects open, MUIbase changes directories according to the currently active project. Once this menu item has been activated for a project, filenames can be specified relative to the directory of the project. This makes a project independent of where it is stored in the file system.

If you leave this menu item unchecked, MUIbase uses the initial working directory when the program was started.

Default is unchecked.

### 7.2.4 Confirm Save & Reorg

Saving and reorganizing a project can take quite some time depending on the size of the project. Therefore, if you select menu item **‘Project - Save & reorg’** or **‘Project - Save & reorg as’**, a safety requester pops up asking for confirmation of this operation. The requester only appears if menu item **‘Preferences - Confirm save & reorg’** is checked, thus you can disable this requester by de-selecting the menu item. Default is checked.

### 7.2.5 Program Source

You can set the program source of a project in menu item **‘Program - Source’** to be **‘Internal’** or **‘External’**. If set to **‘Internal’**, you can use MUIbase’ built-in editor for editing and compiling the project’s program. This is the default. If you wish to use an external editor for programming, you choose **‘External’** and enter the name of a new file to which MUIbase then writes the program source. This allows to load and edit the program source into your favorite editor. For more information about how to use this feature, see [Section 15.2 \[External program source\]](#), page 70.

Note that when setting this menu item from **‘External’** back to **‘Internal’**, the last successfully compiled version of the program is kept.

### 7.2.6 Cleanup External Program Source

Menu item **‘Program - Cleanup external program source’** decides if the external program source file of a project should be deleted when closing the project or when changing menu item **‘Program - Source’** back to **‘Internal’**. See [Section 15.2 \[External program source\]](#), page 70, for more information about external source files. Default is checked.



### 7.2.7 Program Debug Information

For compiling a project's program, you can choose whether debug information should be included in the executable or not. If you compile without debug information and a run-time error occurs then an error description is generated but there is no information about where exactly the error occurred. If you compile with debug information then you also get the exact error location. Compiling with debug information is a bit slower, requires more memory and the resulting program is slightly less efficient.

Use menu item **'Program - Embed debug information'** to turn debug information for compilation on and off. Default is checked. After changing this state, don't forget to recompile the project's program by choosing menu item **'Program - Compile'**.

### 7.2.8 Obsolete Functions

Since MUIbase version 2.7 a few programming functions have been made obsolete (see [Section 15.30 \[List of obsolete functions\], page 144](#)). The functionality of these obsolete functions have been replaced by other mechanisms and the functions do not work as expected any longer. When opening projects containing programming functions which are now obsolete, you can choose how to handle them.

Menu item **'Program - Obsolete functions'** specifies how to react when an obsolete function is called. When choosing **'Ignore silently'** any call of an obsolete function is ignored and program execution continues by skipping over the function call. **'Warn when called'** opens a dialog window informing the user whenever an obsolete function is called and allows to continue execution or to abort with an error message. This is the default. If **'Treat as error'** is selected, any call of an obsolete function generates an error and compilation of programs containing obsolete functions fails with a proper error message.

It is recommended to choose **'Treat as error'** after all calls of obsolete functions have been removed from a project program. In order to find out if your project contains any calls of obsolete functions select **'Treat as error'** and recompile the project program.

### 7.2.9 Sort Trigger Functions

If menu item **'Preferences - Sort trigger functions in pop-ups'** is checked then functions available as triggers are sorted alphabetically for display in requesters for creating and changing tables (see [Section 14.1.1 \[Creating tables\], page 56](#)) and for creating and changing attributes (see [Section 14.2.1 \[Creating attributes\], page 57](#)). Otherwise, functions are listed as they occur in the project's program. Default is unchecked.

### 7.2.10 Program Include Directory

The programming feature of MUIbase allows to include external source files within the project's program (see [Section 15.3.3 \[#include\], page 71](#) for more information). Menu item **'Program - Include directory'** allows to set a directory where MUIbase should search for such include files. Default is **'MUIbase:Include'**.

### 7.2.11 Program Output File

When executing a MUIbase program all output directed to **'stdout'** is actually printed to a file. The name of this file can be entered in a dialog by choosing menu item **'Program - Output file'**. You also specify if the output should be appended to the file or if the file should be erased and re-created on new output.



On Windows, Mac OS and Linux you can direct the output to an external program for processing the data. For this, the first character of the filename must be the pipe symbol ‘|’ followed by the external program and its arguments. The external program has to read the data from its standard input. This allows on Linux for example the following re-directions:

- ‘|lpr’ prints the output on the default printer.
- ‘|mknod /tmp/pipe p; (xterm -e less -f /tmp/pipe &); cat > /tmp/pipe; rm /tmp/pipe’ displays the output using the program ‘less’ in its own ‘xterm’ window.
- ‘/dev/pts/0’ writes the output to a terminal connected to ‘pts/0’. On some desktop environments, e.g. KDE, a daemon listens to this terminal and the output gets displayed in a window.
- ‘/dev/stdout’ passes the output to the standard output channel of MUIbase. This is the default.

On the Amiga there are a few special filenames for directing the output, e.g.:

- ‘PRT:’ prints the output on your printer.
- ‘CON:////MUIbase output/CLOSE/WAIT’ prints the output in a Shell window.
- ‘CONSOLE:’ prints the output in the Shell window where MUIbase has been started from. This is the default.

## 7.3 Save as Default

The project settings are individual for each project, i.e. different projects can have different settings. For new projects a default setting is chosen. This default setting is stored together with the user settings in the user environment (see [Section 7.1 \[User settings\]](#), page 32).

In order to customize the default settings to your needs, you can store the settings of the current project as the default settings for new projects by choosing menu item ‘**Preferences - Save as default**’.

## 8 Record-Editing

This chapter describes how to add, change, delete, and browse records of a database table.

### 8.1 Active Object

MUIbase uses a cursor for displaying which object is the active one. If the active object is a string object, the usual text cursor appears, other objects get a special frame around them. You can cycle through the active objects by pressing the *Tab* or *Shift-Tab* keys. If you press the *Help* or *F1* key, an external viewer is loaded with helpful information about the active object.

The table in which the active object resides is called the *active table*. The panel of a table can be set to the active object. This ensures that you can always set a table to be the active one, although the table may not contain any other activate-able objects.

On Windows, Mac OS and Linux each table has a context menu containing menu items for manipulating the table. This context menu can be opened by pressing the right mouse button somewhere inside the table mask (but outside any other GUI object that has its own context menu).

On Mac OS and Amiga the table menu items are part of the global menu found at the top of the screen.

### 8.2 Adding Records

If you select menu item ‘**Table – New record**’ a new record is allocated in the active table. The record is initialized with the initial values for all attributes. It is also possible to duplicate the current record of the active table by selecting menu item ‘**Table – Duplicate record**’.

If you have installed a trigger function for adding a new record (see [Section 14.1.1 \[Creating tables\]](#), page 56) then this trigger function is called for creating the record. For more information on this mechanism, See [Section 15.29.5 \[New trigger\]](#), page 139.

### 8.3 Changing Records

To change the current record in a table you can activate any attribute object within the table’s mask and enter a new value. For string, integer, real, date, time, and memo attributes you can use the usual editing commands.

An attribute object may have been configured as read-only. In this case you can’t change its value (exception: string attribute with pop-up button).

#### 8.3.1 String Attributes with Pop-up Button

If a string attribute has a pop-up button attached to it then you can press the pop-up button and get a requester to set the string contents, e.g. a file-requester for choosing a filename, or a list of strings to choose one from. The pop-up button can always be used to set the string attribute’s value even if the attribute is set to read-only.

Right to the string field another small button might appear. Pressing this button calls an external viewer to display the file specified in the string field.

### 8.3.2 Entering Integer Values

When entering an integer number, one can use an octal notation (leading 0) or hexadecimal notation (leading 0x) besides the usual decimal notation.

### 8.3.3 Entering Boolean Values

The checked state of Boolean attribute can be toggled with the left mouse button, or with the space bar if the object is the active one.

### 8.3.4 Entering Choice Values

For choice attributes you can choose a value by clicking onto it, or by using the *Up* and *Down* cursor keys to browse through all choice labels.

### 8.3.5 Entering Date Values

Date values can be entered in one of the formats ‘DD.MM.YYYY’, ‘MM/DD/YYYY’ or ‘YYYY-MM-DD’, where ‘DD’, ‘MM’ and ‘YYYY’ are standing for two and four digit values representing the day, month and year of the date respectively. It is possible to omit the year value of a date. In this case the current year is used.

By inserting a single integer value, a date value relatively to the current date can be specified, e.g. when entering ‘0’ the today’s date is used, or when entering ‘-1’ yesterday’s date is used.

### 8.3.6 Entering Time Values

The format for entering time values is specified in the structure editor (see [Section 14.3.3 \[Attribute object editor\]](#), page 62). Possible formats are ‘HH:MM:SS’, ‘MM:SS’ or ‘HH:MM’ where ‘HH’ represent the hours, ‘MM’ the minutes, and ‘SS’ the seconds.

It is possible to omit parts of the format, e.g. entering ‘6:30’ under the format ‘HH:MM:SS’ automatically expands to ‘00:06:30’. When entering a single number, it is regarded as the number of seconds (formats ‘HH:MM:SS’ and ‘MM:SS’) or as the number of minutes (format ‘HH:MM’) respectively, and the corresponding time value is computed.

### 8.3.7 Memo Context Menu

Memo attributes have a context menu that offers further editing possibilities:

- ‘Cut’, ‘Copy’, and ‘Paste’ allow exchanging data with the clipboard.
- ‘Delete’ erases selected text and ‘Select all’ allows to select all text (Windows, Mac OS and Linux).
- ‘Clear’ deletes all text in the memo (Amiga).
- ‘Undo’ and ‘Redo’ allow going back and forth the changes you made to the memo contents (Amiga only).
- ‘Input methods’ and ‘Insert unicode control character’ are GTK-specific menu items (Windows, Mac OS and Linux). Please refer to the GTK documentation.
- With ‘Open text’ and ‘Save text’ you can load and save the memo contents from/into a file.
- ‘External editor’ launches an external editor for editing the memo. See [Section 7.1.2 \[External editor\]](#), page 32, for more information about the external editor.

### 8.3.8 Entering Reference Values

For reference attributes the record reference can be entered through a pop-up list:

- To the right of a reference attribute you find a pop-up button which, if pressed, opens a list of records. Choose a record from the list to set the reference to this record, ‘Initial’ to set the reference to the NIL value, or ‘Current’ to set the reference to the current record of the referenced table.
- You can search for an entry in the referenced table by using the keyboard. After the first key press an input field is opened allowing to enter more characters for the search pattern. After each key press, a search is started immediately (case-insensitive) and the first matching entry is selected. The search method can be specified in the display object of the attribute (see [Section 14.3.3 \[Attribute object editor\]](#), page 62) under the ‘Quick search’ category. You can use the characters ‘\*’ for matching an arbitrary sequence of characters and ‘?’ for matching exactly one arbitrary character. Using the cursor keys *Down* and *Up*, the next respectively previous matching entry is selected. A selected entry is stored when acknowledged by pressing the *Enter* key. If you leave the search window by other means, e.g. pressing *Esc*, then the attribute stays unchanged and keeps its previous value.

### 8.3.9 Entering NIL Value

To enter the NIL value, enter any invalid string for the given attribute type, e.g. if you enter ‘xyz’ in an integer attribute then the value of this attribute is set to NIL. Please note that not all attribute types support the NIL value. See [Section 5.6 \[Table of attribute types\]](#), page 21, for an overview of all attribute types.

## 8.4 Deleting Records

To delete the current record chose menu item ‘Table – Delete record’. Before deleting the record a safety requester may appear asking you for confirmation. You can enable and disable this requester in the preferences settings (see [Section 7.2.2 \[Record delete requester\]](#), page 35).

If you have installed a trigger function for deleting records (see [Section 14.1.1 \[Creating tables\]](#), page 56) then this trigger function is called for deleting the record. For more information on this mechanism, see [Section 15.29.6 \[Delete trigger\]](#), page 140.

It’s also possible to delete all records of a table by choosing menu item ‘Table – Delete all records’. Only records matching the record filter of the corresponding table are deleted. Before deletion a safety requester appears, if enabled. No trigger function is called when deleting all records.

## 8.5 Browsing Records

To view other records than the currently displayed one, select one of the sub menu items in menu item ‘Table – Goto record’. You can go to the previous, next, first, or last record, jump several records backward or forward, or enter the record number of the record you want to view. The record number in this context is the number that is displayed in the corresponding panel for that record (see [Section 5.9.3 \[Panels\]](#), page 26). The panel may also include two arrow buttons for going to the previous and next record.

Record browsing can be easily done using the *Up* and *Down* cursor keys in combination with the *Shift*, *Alt*, and *Ctrl* keys. All possible combinations are listed in menu item ‘Table - Goto record’ and in the following table:

	<i>Alt</i>	<i>Shift-Ctrl</i>	<i>Shift-Alt</i>
<i>Up</i>	Previous record	First record	Jump backward
<i>Down</i>	Next record	Last record	Jump forward

## 9 Filter

Filters can be used to hide records. This chapter describes what types of filters are available and how to use them.

MUIbase knows two types of filters, record filters and reference filters.

### 9.1 Record Filter

A record filter can be installed into a table to filter out records that are not of interest to you. Records that are filtered out are excluded in the table mask and thus the user can't see or browse to them.

#### 9.1.1 Filter Expression

A filter is defined by specifying a Boolean expression that may contain calls to MUIbase programming functions. For each record of the table the filter is specified for, this expression is evaluated. If it returns NIL then the record is filtered out, otherwise it is included in the table mask.

Each table can have its own filter expression.

#### 9.1.2 Changing Filters

To change the filter of the current table select menu item '**Table - Change filter**'. This will open a window containing

- the name of the table you install the filter for in the window title.
- a list of all attributes of the table that can be used in the filter expression. This list is placed in the left part of the window. If you double click a name then the name will be inserted into the filter expression at the current cursor position.
- a table of buttons displaying MUIbase programming functions and operators placed in the right part of the window. Click on one of the buttons to enter the corresponding function in the filter expression. Please note that the presented list of functions and operators is not complete. Other MUIbase functions not shown in one of the buttons must be entered manually. Only those MUIbase functions can be used that don't have side effects, e.g. it is not possible to write data to a file within a filter expression.
- a string input field to enter the filter expression. The attribute and function/operator names are inserted here. You can also directly input your filter expression.
- two buttons '**Ok**' and '**Cancel**' to leave the window.

After you are done with the specification of the filter expression click on the '**Ok**' button to leave the window. The entered expression is now compiled and on successful compilation, the expression is evaluated for all records. Those records for which the Boolean expression doesn't hold are then discarded in the table mask.

In case the expression doesn't compile you will get an error message displayed in the window title bar.

A filter can be turned on and off by clicking on the '**F**' button in the table's panel (if it is installed). After you have specified a filter expression for a table the filter for this table is turned on automatically.

If you (re-)activate a filter then all records of the table are checked whether they match the filter or not.

If a filter is currently active and you change a (filter relevant) attribute in a record of this table then the match-filter state of this record is not recomputed and stays unchanged.

If you allocate a new record in a table with an activated filter then there is no check if the new record matches the filter and the new record has its match-filter state set to `TRUE`.

### 9.1.3 Filter Examples

Here are some examples for valid filter expressions:

- ‘`NIL`’ filters out all records.
- ‘`TRUE`’ doesn’t filter out any record.
- ‘`0`’ has the same effect as ‘`TRUE`’ because for MUIbase all expressions not equal to `NIL` are considered as `TRUE`.
- ‘`(> Amount 100.0)`’ only displays records where the ‘`Amount`’ attribute is greater than 100.0 (we assume here that the table has an attribute ‘`Amount`’ of type real).
- ‘`(NOT (LIKE Name "*x*))`’ filters out all records that have the letter ‘`x`’ in the ‘`Name`’ attribute (a string attribute).

Please note that MUIbase’ programming language uses a lisp-like syntax. For more information about the programming language, see [Chapter 15 \[Programming MUIbase\]](#), page 70.

## 9.2 Reference Filter

Reference attributes can also have a filter behavior. This is useful e.g. for building a hierarchy of tables. As an example see the project ‘`Albums`’.

If the filter of a reference attribute is turned on then the following features are activated:

1. The user can only access records in the attribute’s table that have the reference set to the current record of the referenced table.
2. If the referenced table changes its current record then also a new current record is searched and set for the attribute’s table.
3. When allocating a new record the reference is automatically set to the current record of the referenced table.

Note: Cascading delete has to be implemented manually (by using a delete trigger function).

Do not use the reference filter on cyclic graphs, e.g. self-referencing tables, as it doesn’t make much sense.

## 10 Order

For each table in your database you can specify in which order its records should be displayed. This chapter describes how you can specify an order and what consequences it has.

### 10.1 Empty Order

By default each newly created table has an empty order. This means that if you enter a new record, the created record is inserted at the current position, that is, behind the current record of the table. If you change some fields of a record, the position of the record within the table stays unchanged.

### 10.2 Order by Attributes

Sometimes it is useful to have the records sorted by a certain attribute, e.g. by the ‘Name’ attribute if the table has one.

In MUIbase you can specify for each table a list of attributes by which its records should be sorted. All records are first sorted by the first attribute in this list. If two records are equal in an attribute then the next attribute in the list determines the order. For each attribute you can specify if the records should be sorted ascending or descending.

For determining the order of attributes the rules of the following table are used:

Type	Order relation
Integer Choice	$NIL < MIN\_INT < \dots < -1 < 0 < 1 < \dots < MAX\_INT$ (Choice values are treated as integers)
Real	$NIL < -HUGE\_VAL < \dots < -1.0 < 0.0 < 1.0 < \dots < HUGE\_VAL$
String Memo	$NIL < "" < \dots < "a" < "AA" < "b" < \dots$ (String comparison is done case insensitive)
Date	$NIL < 1.1.0000 < \dots < 31.12.9999$
Time	$NIL < 00:00:00 < \dots < 596523:14:07$
Boolean	$NIL < TRUE$
Reference	$NIL < any\_record$ (Records itself are not comparable for ordering)

If you have specified an order for a table then the records are automatically rearranged whenever you add a new record or change an order-relevant field of a record.



### 10.3 Order by a Function

Sometimes you want a more complex ordering scheme than a simple attribute list as described in the last section. For example, an attribute list can't hold reference attributes, so it is not possible to sort your records by a reference field. The reason for this is that with reference attributes MUIbase is not able to keep your records ordered at all times (see [Section 15.29.7 \[Comparison function\]](#), page 140 in the MUIbase programming chapter for more details).

However, MUIbase also offers the possibility to specify a comparison function for sorting your records. You can specify any function that you have written using the MUIbase program editor. The function will be called with two record pointers and the returned value should reflect the order of the two records. The function can use any operations for comparing the records, thus it can e.g. compare records using reference attributes. For more information on this mechanism, see [Section 15.29.7 \[Comparison function\]](#), page 140.

If you decide to use a comparison function for ordering a table then please note that MUIbase cannot always detect when records of the table have to be reordered since the dependencies are unknown. Use menu item 'Table - Reorder all records' when records are becoming unsorted.

### 10.4 Changing Orders

To specify an order for the current table select menu item 'Table - Change order'. This will open a window containing

- the name of the table in the window title.
- a choice field 'Type' where you specify whether the table should be ordered by using an 'Attribute list' or by using a 'Comparison function'. Depending on the state of 'Type' you can enter data into the following items.

For an order using an attribute list, the following items exist:

- a list of all attributes of the table that can be used in the order list. This list is placed in the left part of the window. If you double click a name then the name will be inserted into the order list at the current cursor position.
- the current list of attributes used for ordering. This list is placed in the right part of the window. The top item in this list is the first attribute of the order list. You can rearrange items by dragging them to other positions in the list. To add further attributes drag them from the attribute list to the order list. You can remove an attribute from the order list by dragging the item out of the order list and dropping it onto the attribute list.

Each entry in the order list has an arrow symbol pointing up or down on the left side. You can toggle the state of the arrow by double-clicking it. If an arrow is pointing up then the sorting order for this attribute is done ascending, if it is pointing down, it is done descending.

For an order using a comparison function, there is the following item:

- a field 'Function for comparing records' where you can enter the name of a function that should be called for comparing two records of the table. You can use the pop-up button to the right of the string field for choosing a name from a list of all function

names. If you leave the field empty then an empty order is used. For more information on how to use a comparison function, including the arguments that are passed to it, see [Section 15.29.7 \[Comparison function\]](#), page 140.

Furthermore, the following buttons exist:

- a **‘Clear’** button that clears all fields for an empty order.
- two buttons **‘Ok’** and **‘Cancel’** for leaving the window.

To enter a new attribute order list, choose **‘Attribute list’** in the **‘Type’** field and press the **‘Clear’** button. Then use drag & drop as described above to build up a new list of attributes. If you want to have an empty order then just don’t add any attributes to the order list.

When you are done specifying the order, press the **‘Ok’** button. MUIbase will then reorder all records of the table.

## 10.5 Reorder all Records

If you ever think that some records of a table became unsorted, e.g. when using a comparison function for ordering, then you can choose menu item **‘Table – Reorder all records’** for sorting all records.

## 11 Search For

For record browsing you can use a search requester to search for a specific record. The search feature uses a search pattern that you provide and checks all records for a successful match with this pattern. If it finds one then this record will be displayed in the table mask.

### 11.1 Search Requester

To open the search requester select menu item **‘Table - Search for’**. This will open a window containing

- a string field for entering the search pattern. Use the characters **‘\*’** and **‘?’** as jokers. The character **‘\*’** matches any number of characters (including zero characters) whereas **‘?’** matches exactly one single character.
- a field **‘Case sensitive’**. If you check it, searching uses case sensitive string comparison, otherwise case insensitive string comparison is used.
- a field **‘All fields’**. If checked then all fields of a record are tested for a successful match with the search pattern. Otherwise only the attribute which has been the active one when the search requester has been opened is tested. In case the active object at the time when opening the search requester has not been an attribute object, this field is checked and disabled automatically.
- two radio buttons for the search direction, **‘Forward’** and **‘Backward’**.
- two radio buttons for specifying from which record the search should be started. **‘First/last record’** will start the search from the first or last record depending on the search direction. **‘Current record’** starts the search from the current record.
- two buttons **‘Search’** and **‘Cancel’** for leaving the window.

After you have entered a search pattern and left the requester with the **‘Search’** button MUIbase starts searching for a matching record. The comparison of an attribute field with the search pattern is always done string based, attribute fields of non-string types are therefore converted to strings first.

If a matching record is found, it is displayed as the current record in the table mask. Otherwise a **‘Search pattern not found’** message appears.

If you search on an attribute that is used as the first attribute for ordering and your search pattern doesn’t start with a joker (**‘\*’** or **‘?’**) then an improved search algorithm (binary search) is used that makes use of the record order. This can speed up searching significantly.

### 11.2 Forward/Backward Search

Two further menu items allow to search for the next and the previous occurrence of the search pattern. Select menu item **‘Table - Search forward’** for browsing to the next record that matches the search pattern, and **‘Table - Search backward’** to go to the previous matching record.

## 11.3 Search Pattern Examples

Here are some search pattern examples:

- ‘Lassie’ searches for records that have the string ‘Lassie’ in one of the search attributes.
- ‘\*x\*’ searches for records that have the letter ‘x’ in one of the search attributes.
- ‘???’ searches for records that have exactly three characters in one of the search attributes, e.g. a record with an entry ‘UFO’.

## 12 Import and Export

For sharing your records with other databases, MUIbase offers a way to import and export records from and to other databases. Import and export is done by reading and writing text files. The text files must follow a special format described in the next section.

### 12.1 File Format

For importing records into MUIbase, all records of a table should be in a single text file. If records of several tables are to be imported, they should be in separate import files, one for each table.

An import file consists of lines and columns. Lines are separated by a record delimiter, columns by a field delimiter. The delimiters can be specified in the import and export dialog. Since a record field itself may contain such a delimiter, double quotes can be used around all fields for escaping delimiters.

The import file should be of the following structure:

- The first line contains the field names. For each name the table where the records are imported to must contain an attribute with the exact same name. In case no matching attribute is found for a given name, the import fails with an error message.
- The following lines contain one record each. Since all fields are given as strings, they are converted automatically to the type of the destination attribute. For attributes of type Boolean, the field must be either NIL or TRUE (case insensitive), otherwise an error message is generated. For attributes of type choice the exact label string must be specified (case sensitive). For reference attributes the record number starting with 1 must be specified. For all other types a value of NIL is used if the field cannot be converted to the required type.
- If you decide to use double quotes then all record fields including the line containing the field names must be surrounded by double quotes.

### 12.2 Sample Import File

The following sample import file uses `\n` and `\t` for record and field delimiters and double quotes around all fields. The file can be imported into a table with the following attributes:

- Name (string)
- NumChildrens (integer)
- Feminine (Boolean)
- Job (choice)
- Notes (memo)

```
"Name" "NumChildrens" "Feminine" "Job" "Notes"
"Janet Jackson" "???" "TRUE" "Musician" "Latest CD: The velvet rope"
"Bernt Schiele" "???" "NIL" "Scientist" "Research interests:
Robotics, Autonomy and Computer Vision"
"Gerhard" "0" "NIL" "Precision mechanic" ""
```

## 12.3 Importing Records

To import records into the active table select menu item **‘Table – Import records’**. This will open a window containing

- a string field for entering the import filename. To the right of this field you find three buttons. Click on the first one to open a file dialog for choosing the filename. The second button starts an external viewer with the entered filename while the third button starts an editor for editing the file contents.
- two string fields for entering the record and field delimiters. You can enter a single character or an escape code by typing `\n`, `\t`, `\f`, `\??? (octal code)`, or `\x?? (hex code)`. Delimiter characters must be 7bit ASCII characters.
- a field **‘Double quotes’** that can be checked to specify that the fields are surrounded by double quotes.
- a field **‘Overwrite records’** that if selected will overwrite existing records with the imported data. This can be useful in case you want to keep the existing records (e.g. because there are references to them) but update them with new information.
- two buttons **‘Import’** and **‘Cancel’** for leaving the window.

If you press the **‘Import’** button, MUIbase will load the specified file and import all found records. If everything went fine and new records were created in the table, MUIbase asks after the import process if you really want to add the newly imported records to the table. At this point you can still cancel the operation. Overwritten records, however, can only be recovered by reverting the project.

If an error occurs while reading the import file then an error message is shown.

If you need a more sophisticated import routine it is recommended to write your own import routine as a MUIbase program.

## 12.4 Exporting Records

To export records from the active table select menu item **‘Table – Export records’**. This will open a window containing

- a string field for entering the export filename. Right to this field you find a pop-up button to open a file dialog for choosing the filename.
- two string fields for entering the record and field delimiters. You can enter a single character or an escape code by typing `\n`, `\t`, `\f`, `\??? (octal code)`, or `\x?? (hex code)`.
- a field **‘Double quotes’** that can be checked to specify that the fields should be surrounded by double quotes.
- a field **‘Filter’**. If checked only the records that match the currently installed record filter are written to the export file.
- two buttons **‘Export’** and **‘Cancel’** for leaving the window.

After you pressed the **‘Export’** button, MUIbase will open the specified file and write out the records including a header line containing the attribute names. The export feature always writes all attribute fields of a table to the export file.

For a more customized export routine, you can use MUIbase’ query editor (see [Chapter 13 \[Data retrieval\]](#), page 51) or you write your own export function as a MUIbase program.

## 13 Data Retrieval

For data retrieval MUIbase offers two ways: the programming feature and the query editor.

The programming feature allows you to install buttons in table masks which, when pressed, call program functions. The usage of this feature is described in the structure editor chapter (see [Chapter 14 \[Structure editor\]](#), page 56) and in the chapter about programming MUIbase (see [Chapter 15 \[Programming MUIbase\]](#), page 70).

This chapter describes the usage of the query editor, a requester where you can enter queries and view the output in a scrolling list-view.

### 13.1 Select-From-Where Queries

MUIbase offers a select-from-where query similar to the one in SQL database systems. The query allows you to list the record contents from one or more tables. Only records matching certain criteria are included in the output. The (incomplete) syntax of an select-from-where query is

```
SELECT exprlist FROM tablelist [WHERE test-expr]
[ORDER BY orderlist]
```

where *exprlist* is a comma separated list of expressions to be printed (usually the attribute names) or a simple star \* matching all attributes of the specified tables, *tablelist* is a comma separated list of tables whose records are examined, *test-expr* is the expression that is tested for each set of records to be included in the output, and *orderlist* is a comma separated list of attributes that defines the order for listing the output. Please note that the WHERE and ORDER BY fields are optional, denoted by the brackets [].

For example, the query

```
SELECT * FROM table
```

lists the attribute contents of all records in the given table.

```
SELECT attr1 FROM table WHERE (LIKE attr2 "*Madonna*")
```

lists the value of the *attr1* field in all records of *table* where the contents of field *attr2* contain the word 'Madonna'.

For more information about the select-from-where query including its full syntax, see [Chapter 15 \[Programming MUIbase\]](#), page 70, for more example see [Section 13.4 \[Query examples\]](#), page 53.

### 13.2 Query Editor

For entering and running queries, open the query editor by choosing menu item 'Program - Queries'. The query editor is able to manage several queries, however only one query is displayed at a time. The query editor window contains the following items:

- a string input field with an attached pop-up button. The edit-able string field displays the name of the current query. By pressing the pop-up button, a list with further query names together with several buttons appear. You can select one of the listed queries to make it the current one, press the 'New' button to create a new query, press the 'Duplicate' button to get a copy of the activated query, click on the 'Sort' button to sort the list of queries, or press the 'Delete' button to delete the selected query.

For leaving the pop-up window without changing anything, click on the pop-up button again.

- a ‘Run’ button that compiles and runs the query program and displays the output in the output list-view.
- a ‘Print’ button that opens a requester (see [Section 13.3 \[Printing queries\], page 52](#)) for printing the results.
- an editor field for entering the query program. Here you usually enter a select-from-where query. However, it is also possible to enter any expression of MUIbase’ programming language. This can be useful if you want to do simple computations or update some fields of a table by using a simple program. Please note that MUIbase automatically surrounds your program expression with a pair of parenthesis, thus you can omit the outermost ones.
- a list-view that displays the output after running the current query. The output is formatted into rows and columns. The title row holds the field names of the select-from-where query (usually the attribute names). The other rows hold the contents of the query result, one set of records per row. Each field is displayed in its own column. By clicking on a column title you can sort the list with respect to this column. A second click onto the same column title reverses the order. On the Amiga you can set a secondary sort column by holding down the *Shift* key when clicking on a title. If you double click on entry in the list and the entry was generated from a record then this record is displayed in the corresponding table mask. This offers an easy way to jump to a certain record in its table mask.

The query editor is a non-modal requester. This means that you can leave the query editor open and still work with the rest of the application. You can close the query editor at any time by clicking on the close button in the window title bar.

### 13.3 Printing Queries

After you have run a query you can print the result to a file or printer by clicking on the ‘Print’ button in the query editor. This opens a print requester containing the following items:

- a field ‘Delimiter’ where you specify how the columns should be separated. ‘Spaces’ pads the fields with space characters. Padding is done on the left or on the right side depending on the type of the field (numbers are padded on the left, text on the right side). ‘Tabs’ inserts exactly one tab character between the columns. This can be useful if you want to use the print requester for exporting records (see below). ‘Custom’ allows to specify a custom delimiter string to be printed between fields.
- a field ‘Font’ where you specify which font should be used for printing the output. ‘NLQ’ stands for near letter quality which should print the output in better quality than ‘Draft’.
- a field ‘Size’ where you define the character size. ‘Pica’ prints in a large font (10 CPI), ‘Elite’ in a medium font (12 CPI) and ‘Condensed’ in a small font (17 CPI).
- a string field ‘Init sequence’ where you can enter a string for initializing your printer. The contents of this field are written directly after opening the printer. For example you can use ‘\33c’ as init sequence which resets your printer.



- a field ‘**Indent**’ where you can enter a number of spaces that are used for indenting each output line.
- a field ‘**Headline**’ that, if checked, prints the field names in the first line.
- a field ‘**Escape codes**’. If not checked the output of all escape codes is suppressed which means that the settings of the fields ‘**Font**’ and ‘**Size**’ are ignored and the contents of ‘**Init sequence**’ are not printed. Suppressing the output of all escape codes is useful if you want to generate an ASCII file, e.g. for exporting records.
- a field ‘**Quotes**’ that, if checked, surrounds all fields with double quotes.
- a field ‘**After printing**’ where you can specify how the output should be finished. ‘**Form feed**’ prints a form feed character `\f`. ‘**Line feeds**’ prints a number of line feed characters `\n`. The number of line feeds can be entered in the string field to the right of the ‘**Line feeds**’ button. ‘**Nothing**’ doesn’t write anything to the printer.
- a string field ‘**Output**’ with an attached pop-up button. You can use the pop-up button to open a file requester for choosing a filename or directly enter the filename into the string field. For writing the output to your printer enter ‘`|lpr`’ (Linux) respectively ‘`PRT:`’ (Amiga). For other special filenames, see [Section 7.2.11 \[Program output file\]](#), [page 36](#).
- two buttons ‘**Ok**’ and ‘**Cancel**’ for leaving the print requester.

After you are done with all settings, click on the ‘**Ok**’ button to start the print job.

The print requester can also be used for exporting records to an ASCII file. To do this, specify ‘**Tabs**’ (or ‘**Custom**’) in the ‘**Delimiter**’ field, set the number of spaces in the ‘**Indent**’ field to 0, check ‘**Headline**’, un-check ‘**Escape codes**’ to suppress the font, size and init sequence settings, optionally check ‘**Quotes**’ if you want the field contents to be surrounded by double quotes, check ‘**Nothing**’ in the ‘**After printing**’ field, and enter the output filename in the ‘**Output**’ field. Using the query editor together with the print requester for exporting records can be more powerful than when using MUIbase’ import/export feature (see [Chapter 12 \[Import and Export\]](#), [page 49](#)) since in the query editor any query can be entered whereas the export requester only uses a fixed query.

## 13.4 Query Examples

To give you an impression of the power of the select-from-where queries this section gives you some sample queries.

Suppose we have two tables ‘**Person**’ and ‘**Dog**’. ‘**Person**’ has a string attribute ‘**Name**’, an integer attribute ‘**Age**’, and two reference attributes ‘**Father**’ and ‘**Mother**’ that refer to the father and mother records in table ‘**Person**’. The table contains the following records:

	Name	Age	Father	Mother
-----				
p1:	Steffen	26	p2	p3
p2:	Dieter	58	NIL	NIL
p3:	Marlies	56	NIL	NIL
p4:	Henning	57	NIL	NIL

'Dog' has a string attribute 'Name', a choice attribute 'Color' and a reference attribute 'Owner' that refers to the owner in the 'Person' table. The table contains the following records:

	Name	Color	Owner
d1:	Boy	white	p3
d2:	Streuner	grey	NIL

Given these data the following sample select-from-where queries can be run:

```
SELECT * FROM Person
```

results to:

Name	Age	Father	Mother
Steffen	26	Dieter	Marlies
Dieter	58		
Marlies	56		
Henning	57		

(For the reference attributes the 'Name' field of the referenced record is printed.)

```
SELECT Name "Child", Age,
       Father.Name "Father", Father.Age "Age",
       Mother.Name "Mother", Mother.Age "Age"
FROM Person WHERE (AND Father Mother)
```

results to:

Child	Age	Father	Age	Mother	Age
Steffen	26	Dieter	58	Marlies	56

```
SELECT Name, Color,
       (IF Owner Owner.Name "No owner") "Owner"
FROM Dogs
```

results to:

Name	Color	Owner
Boy	white	Marlies
Streuner	grey	No owner

```
SELECT a.Name, a.Age, b.Name, b.Age FROM Person a, Person b
WHERE (> a.Age b.Age)
```

results to:

a.Name	a.Age	b.Name	b.Age
Dieter	58	Steffen	26
Marlies	56	Steffen	26
Henning	57	Steffen	26
Dieter	58	Marlies	56
Henning	57	Marlies	56
Dieter	58	Henning	57

## 14 Structure Editor

MUIbase has two different operating modes, the record-editing mode, where you enter and browse records, and the structure-editing mode where you define the structure, that is, the tables, attributes and appearance of a project. This chapter describes the structure editor and explains how to manage the structure of a project.

To switch from record-editing to structure-editing mode select menu item **‘Structure editor’** in the **‘Project’** menu. This closes all windows and opens the structure editor window. To switch back to record-editing mode select menu item **‘Project - Exit structure editor’** or simply close the structure editor by clicking on the close button in the window title bar.

The structure editor window is divided into three parts: in the upper left part there is a field **‘Tables’** for managing the tables of the project, the lower left part is occupied by a field **‘Attributes’** for managing the attributes of a table, and the right part is used by a field **‘Display’** for managing the project’s GUI elements.

### 14.1 Table Management

In the **‘Tables’** field of the structure editor you can create, change, delete and sort tables.

#### 14.1.1 Creating Tables

To create a new table press the **‘New’** button in the **‘Tables’** field. This will open the **‘New table’** requester containing

- a string field for entering the name of the table. Each table must have a unique name that starts with an uppercase letter followed by further letters, digits or underscore characters. Non-ASCII characters like German umlauts are not allowed. Please note that in the user interface for the table it is still possible to display any strings including strings with non-ASCII characters.
- a field **‘Number of records’** where you specify how many records the table is going to hold. **‘Unlimited’** means that the table can hold any number of records, **‘Exactly one’** means that the table can have only one record. The latter one is sometimes useful for controlling the project (see [Section 5.2 \[Tables\]](#), page 18).
- a field **‘Trigger functions’** where you can enter the names of two functions. In the **‘New’** string field you enter the name of the function that should be called whenever the user wants to create a new record, the **‘Delete’** string field holds the name of the function that should be called whenever the user wants to delete a record. You can use the pop-up buttons to the right of the string fields for choosing a function name from a list of all names. If you leave a field empty then default actions are executed (records are created automatically and records are deleted after an optional safety requester). For more information on how to use these trigger functions, including the arguments that are passed to them, see [Section 15.29.5 \[New trigger\]](#), page 139 and [Section 15.29.6 \[Delete trigger\]](#), page 140.
- two buttons **‘Ok’** and **‘Cancel’** for leaving the requester.

When you are done with all settings, press the **‘Ok’** button to create the new table. If you made an error somewhere, e.g. you entered an invalid name, a message window pops

up giving you more information about the error you did. If everything goes fine, the ‘**New table**’ requester will be closed and the new table is displayed in the structure editor’s table list.

### 14.1.2 Changing Tables

After you have created a new table you can still change it. Just double-click on the table’s name and the ‘**Change table**’ requester pops up. This requester is similar to the one when creating a table (see [Section 14.1.1 \[Creating tables\], page 56](#)) and allows you to make changes in any field by entering a new value.

When you are done with all changes, press the ‘**Ok**’ button to leave the requester.

Please note that you can’t change the number of records from ‘**Unlimited**’ to ‘**Exactly one**’ if the table already contains more than one record.

### 14.1.3 Deleting Tables

To delete a table, click on the table’s name in the structure editor’s table list, then press the ‘**Del**’ button below the list. Before the table is actually deleted a safety requester pops up asking for confirmation. If you confirm this requester by pressing the ‘**Delete**’ button, the table is deleted.

A problem occurs if the table is used somewhere in the project’s program. In this case the table can’t simply be deleted but all references to the table must be removed from the program. If the table you want to delete is used in the project’s program then the program editor pops up and displays the first occurrence to the table. You should now modify the program such that no references to this table remain in the program. After you removed a reference you can jump to the next one by pressing the ‘**Compile**’ button. At any point you can still cancel the whole operation by pressing the ‘**Revert**’ button and closing the program editor.

### 14.1.4 Sorting Tables

For sorting the tables in the ‘**Tables**’ field of the structure editor you have several choices. You can order them manually, that is, you use drag & drop to rearrange a table, or you use the ‘**Sort**’ button below the list-view which orders the tables alphabetically (‘**Sort1**’), or according to the order in the list display (‘**Sort2**’).

## 14.2 Attribute Management

In the ‘**Attributes**’ field of the structure editor you can create, copy, change, delete and sort the attributes of the active table in the ‘**Tables**’ field.

### 14.2.1 Creating Attributes

To create a new attribute for the active table press the ‘**New**’ button in the ‘**Attribute**’ field. This will open the ‘**New attribute**’ requester containing

- a string field for entering the name of the attribute. Each attribute in a table must have an unique name that starts with an uppercase letter followed by further letters, digits or underscore characters. Non-ASCII characters like German umlauts are not allowed. Please note that in the user interface it is still possible to display any strings including strings with non-ASCII characters for the attribute.

- a choice field ‘**Type**’ where you specify the type of the attribute. For more information on attribute types, see [Section 5.5 \[Attribute types\]](#), page 19.
- a section below the ‘**Type**’ field for specifying type specific settings. For more information about this section, see [Section 14.2.2 \[Type specific settings\]](#), page 58.
- a field ‘**Trigger**’ where you can enter the name of a function that should be called whenever the user wants to change the contents of the attribute field in a record. You can use the pop-up button to the right of the string field for choosing a name from a list of all function names. If you leave the field empty then a default action is executed, that is, the entered value is simply stored in the attribute field. For more information on how to use the trigger function, including the arguments that are passed to it, see [Section 15.29.8 \[Attribute trigger\]](#), page 141.
- a field ‘**Count changes**’. If checked, every change to the attribute in a record counts as a change of the project. Uncheck this item, if you want to ignore changes to an attribute.
- two buttons ‘**Ok**’ and ‘**Cancel**’ for leaving the requester.

When you are done with all settings, press the ‘**Ok**’ button to create the new attribute. If you made an error somewhere, e.g. you entered an invalid name, a message window pops up giving you more information about the error you did. If everything goes fine, the ‘**New attribute**’ requester will be closed and the new attribute is displayed in the structure editor’s attribute list.

### 14.2.2 Type Specific Settings

In the type specific section the following settings can be specified:

- For attributes of type string you have
  - an integer field ‘**Max length**’ for entering the maximum string length for this attribute field.
  - a string field ‘**Initial value**’ for specifying the value that is used for initializing the attribute field. Any string up to the specified maximum length can be entered here.
- For attributes of type integer, real, date, and time the type specific section offers
  - a field ‘**Initial value**’ where you specify the value for initializing the attribute field. You can choose between ‘**NIL**’ and ‘**other**’. If you select ‘**other**’ then you should enter the initial value into the string field on the right.
  - a string field ‘**NIL string**’ where you enter the string that should be displayed when the attribute field holds the NIL value.
- For Boolean attributes the type specific section contains a field ‘**Initial value**’ where you can choose between ‘**NIL**’ and ‘**TRUE**’ for the initial value.
- The type specific section for choice attributes offers
  - a button ‘**Edit labels**’ for opening the ‘**Edit labels**’ window where you can enter the label strings for the choice field (see [Section 14.2.3 \[Label editor\]](#), page 59).
  - a choice field ‘**Initial value**’ for specifying the value for initializing the attribute field.
- For reference attributes the type specific section contains

- a list-view displaying all tables for specifying to which table the reference should be made. Click on the table the attribute is going to reference.
- a field ‘Auto show’. If checked, the referenced table gets updated automatically with the referenced record whenever the user switches to another record.
- a field ‘Filter’. If checked, the reference filter of this attribute is turned on. See [Section 9.2 \[Reference filter\]](#), page 43, for more information about this feature.

Reference attributes always have the NIL value as initial value.

- The type specific section for virtual attributes contains a string field ‘Compute’ where you enter the name of a function that should be called for calculating the value of the attribute. You can use the attached pop-up button for selecting a name from a list of all function names. For more information on how to use this function, see [Section 15.29.9 \[Programming virtual attributes\]](#), page 142.
- Memo and button attributes do not have any type specific settings. The initial value for memo attributes is an empty string.

### 14.2.3 Label Editor

Whenever you have to define a list of labels, e.g. the list of labels for a choice attribute, the label editor comes into place. The label editor is a window containing:

- a list-view displaying the current list of labels. You can click on a label to make it the active one. The active label is also displayed in the string field below the list-view. You can use drag & drop to rearrange the labels.
- a string field ‘Label’ that displays the active label and allows changing it. The changes do only have effect after you pressed the **Enter** key. If there is currently no active label then pressing **Enter** inserts new labels into the list.
- a button ‘New’ that deactivates the current label which allows entering new labels into the ‘Label’ string field.
- a button ‘Remove’ that removes the active label from the list.
- a button ‘Sort’ for ordering the list of labels alphabetically.
- two buttons ‘Ok’ and ‘Cancel’ for leaving the label editor.

After you entered all labels or made changes to them, press the ‘Ok’ button to leave the window.

### 14.2.4 Copying Attributes

In case you need a lot of similar attributes, it is possible to copy an attribute. Just select the desired attribute and press the ‘Copy’ button below the attribute list. This opens the ‘Copy attribute’ requester where the settings of the selected attribute are displayed. Change some of the fields, e.g. the name, and then press ‘Ok’ for creating a copy of the attribute.

### 14.2.5 Changing Attributes

After you have created a new attribute it is still possible to change some settings of it. Just double-click on the attribute’s name and the ‘Change attribute’ requester pops up. This requester is similar to the one when creating an attribute (see [Section 14.2.1 \[Creating attributes\]](#), page 57) and allows you to make changes in some fields. The fields that cannot be changed, e.g. the attribute type, are displayed ghosted.

The following notes should be taken into account when changing an attribute.

- The type of an attribute cannot be changed. If you ever want to change the type of an attribute, it is best to create a new one of the desired type and copy the record contents from the old attribute to the new one by entering a simple MUIbase program in the query editor (see [Section 13.2 \[Query editor\]](#), page 51).
- If you change the initial value of an attribute then only new records will get the new value for initializing the record.
- For choice attributes you should be careful when changing its labels. The labels are only used for displaying the choice field contents, internally, numbers are stored that are used as an index into the list of labels. Thus, if you change the order of labels, you actually don't change the internal number but the label which is displayed for it! Therefore you should not change the order of labels after you created a choice attribute. Appending new labels to the end of the label list, however, doesn't make any problems. For a more flexible way of having a choice-like field where you can also change the order of labels, use a string attribute together with the 'List-view pop-up' feature (see [Section 14.3.3 \[Attribute object editor\]](#), page 62).
- The referenced table of a reference attribute cannot be changed.

When you are done with all changes, press the 'Ok' button to leave the requester.

### 14.2.6 Deleting Attributes

To delete an attribute, click on it's name in the structure editor's attribute list and press the 'Del' button below the list. Before the attribute is actually deleted a safety requester pops up asking for confirmation. If you confirm this requester by pressing the 'Delete' button, the attribute is deleted.

A problem occurs if the attribute is used somewhere in the project's program. In this case the attribute can't simply be deleted but all references to it must be removed from the program. If the attribute you want to delete is used in the project's program then the program editor pops up and displays the first occurrence to this attribute. You should now modify the program such that no references to this attribute remain in the program. After you removed a reference you can jump to the next one by pressing the 'Compile' button. At any point you can still cancel the whole operation by pressing the 'Revert' button and closing the program editor.

### 14.2.7 Sorting Attributes

For sorting the attributes in the 'Attributes' field of the structure editor you have several choices. You can order them manually, that is, you use drag & drop to rearrange an attribute, or you use the 'Sort' button below the list-view which orders the attributes alphabetically ('Sort1') or according to the order in the list display ('Sort2').

## 14.3 Display Management

In the 'Display' field of the structure editor you specify how the database elements should be arranged in the user interface. The field consists of a choice field, a list-view and several buttons.



### 14.3.1 Display Field

The display field contains the following items:

- a choice item with two settings, ‘Table mask’ and ‘Main window’. In ‘Table mask’ you specify how the attributes of the active table are arranged in the user interface. In ‘Main window’ you specify how the tables are arranged.
- a list-view that displays the current specification of the user interface. The list is organized as a tree. Items that have an arrow to its left are composite GUI objects and can be opened and closed by (double-) clicking on the arrow symbol. A double-click on the item itself opens a window for editing its settings. All GUI objects that have the same parent object are layed out in the same way (either vertically or horizontally). How the layout is done is determined by the parent GUI object: tables, panels and windows layout their elements vertically, groups layout their elements according to the settings in the group editor (see [Section 14.3.8 \[Group editor\]](#), page 68).
- a button ‘+’ (‘Add’) for adding the active table or the active attribute (depending on the state of the display choice field) to the display list-view. Usually tables and attributes are added to the display list-view automatically when you create them.
- a button ‘-’ (‘Rem’) for removing the active item from the display list-view. If you remove a table then the whole table form is removed from the user interface, thus you can’t see the table in the project’s GUI. This way you can hide entire tables. If you remove an attribute from the display list-view then the attribute doesn’t appear in the project’s GUI. This is useful for hiding attributes.
- two buttons ‘Up’ and ‘Down’ for moving the active item one field up, respectively down, in the display list-view.
- two buttons ‘In’ and ‘Out’ for moving the active item one hierarchical level down or up in the display list-view.
- a button ‘Panel’ for adding a panel to the table. See [Section 14.3.2 \[Panel editor\]](#), page 62, for more information about setting up a panel.
- a button ‘Text’ for adding a text object to the display list-view (see [Section 14.3.5 \[Text editor\]](#), page 67).
- a button ‘Image’ for adding an image object (see [Section 14.3.6 \[Image editor\]](#), page 67).
- a button ‘Space’ for putting space between the other objects (see [Section 14.3.7 \[Space editor\]](#), page 67).
- a button ‘Balance’ for adding a balance object to the display list-view. The balance object is useful for controlling the size of the other GUI objects.
- a button ‘Group’ for adding a group object to the display list-view. Before you press the ‘Group’ button you can multi-select the items in the list-view that should be moved into the new group. See [Section 14.3.8 \[Group editor\]](#), page 68, for more information about setting up a group object.
- a button ‘Register’ for adding a register group to the display list-view. As for group objects, you can multi-select the GUI objects that should be moved into the new register group. For more information about setting up a register group see [Section 14.3.9 \[Register group editor\]](#), page 68.

- a button ‘**Window**’ for adding a new window to the display list-view. As before, you can multi-select the GUI objects that should be moved into the new window. For more information about setting up a window, see [Section 14.3.10 \[Window editor\]](#), page 68.

For more information about the GUI elements, including their usage, see [Section 5.9 \[User interface\]](#), page 25.

### 14.3.2 Panel Editor

When you add a panel to a table’s mask, or when you double click an existing panel object in the display list-view, the ‘**Panel**’ window pops up. This window contains the following items:

- a string field ‘**Title**’ for entering a title that should be displayed in the panel header.
- a string field ‘**Font**’ with a pop-up button for selecting a font for the title. If you leave the field empty, a default font is used.
- a field ‘**Background**’ with a check-mark field ‘**Default**’ for specifying the background of the panel header. If you check the ‘**Default**’ field then a default background is chosen. Otherwise you can click on the ‘**Background**’ button to open a window for specifying a custom background.
- a field ‘**Num/All**’. If checked the number of the current record and the number of all records are displayed in the right part of the panel header.
- a field ‘**Filter**’ that, if checked, adds a filter button to the panel header. With the filter button you can turn on and off the record filter of the table. If you don’t check this field then the menu item ‘**Table – Change filter**’ will also be disabled for this table, thus you can’t enter a filter expression for the table. For more information about record filters, see [Section 9.1 \[Record filter\]](#), page 42.
- a field ‘**Arrows**’ for adding two arrow buttons to the table mask. The arrow buttons allows you to browse through the records of the table. If you don’t check this field then you can’t browse the records of this table and all sub menu items of menu item ‘**Goto record**’ and the menu items ‘**Search for**’, ‘**Search forward**’, and ‘**Search backward**’ in menu ‘**Table**’ are disabled.
- two buttons ‘**Ok**’ and ‘**Cancel**’ for leaving the window.

After you are done with all settings, click on the ‘**Ok**’ button to close the window.

### 14.3.3 Attribute Object Editor

When you add an attribute to the display list-view, a default GUI object is created for it. To change the settings of the attribute object, a double click on it opens the ‘**Display attribute**’ window. This window contains several items depending on the type of the attribute. The following items are included for most attribute types:

- a string field ‘**Title**’ for entering a title that is displayed near the attribute object (or, for buttons, inside the object). If you leave this field empty then no title is displayed.
- a choice field ‘**Position**’ for specifying where the title (if any) is placed relative to the attribute object. You can choose between ‘**Left**’, ‘**Right**’, ‘**Top**’, and ‘**Bottom**’.
- a string field ‘**Shortcut**’ for entering a letter than can be used together with the **Alt** key (Windows, Mac OS and Linux) or the *Amiga* key to activate the object.

- a field **‘Home’**. If checked, this attribute object becomes the home object. The home object is used as the object where the cursor is set on whenever a new record is allocated. This is quite useful if you always want to start entering data into the same attribute whenever you create a new record. If you mark an attribute object as the home object then all other attribute objects in the same table lose this property.
- a field **‘Tab chain’**. If checked the object is part of the focus chain that can be cycled through by pressing the **Tab** key. Uncheck this item if you want to skip over the element.
- a field **‘Read only’** that, if checked, gives the object a read only status. This means that you can only read its contents but can’t edit them. Note however, that if you add a pop-up button to the object, the contents can still be changed by the selections offered in the pop-up.
- a choice field **‘Alignment’** for specifying how the attribute contents should be presented in the object. You can choose between **‘Center’**, **‘Left’** and **‘Right’** for displaying the contents centered, flushed left, or flushed right.
- an area **‘Enabled/disabled’**. If **‘Always enabled’** is selected then the object is always enabled independent of the record shown. **‘Disabled in initial’** disables the object in the initial record but enables it otherwise. If **‘Compute enabled’** is chosen then a function for computing the enabled state of the object can be entered to the right. The function does not take any arguments. If it returns NIL then the object is disabled, otherwise enabled. In case the compute function is left empty or cannot be found, the object gets disabled. For more information on how to use this trigger function, see [Section 15.29.10 \[Compute enabled function\], page 142](#).
- a numerical field **‘Weight’** for specifying the weight of the object. The value of this field determines how much space, relatively to the other objects, the object gets in the final window layout. For most attribute types this field only affects the horizontal size of the object as most objects have a fixed height. For a Memo attribute, for example, it also affects the vertical size.
- a field **‘Font’** for selecting the font used for displaying the attribute contents. If you leave the field empty, a default font is used.
- a field **‘Background’** with a check-mark field **‘Default’** for specifying how the background of the attribute should look like. If you check the **‘Default’** field then a default background is used, otherwise you can click on the **‘Background’** field to open a window for specifying a custom background.
- an editor field **‘Bubble help’** where you can enter text that should be displayed as the bubble help information for this attribute object.
- two buttons **‘Ok’** and **‘Cancel’** for leaving the window.

If you are done with all settings, press the **‘Ok’** button to leave the window.

### 14.3.4 Type Specific Settings

Besides the above items, the following type-specific items are present:

- for attributes of type string there is an **‘Extras’** page which contains:
  - a field **‘Display picture’** that, if checked, attaches an image field to the string attribute for displaying the image whose filename is taken from the attribute contents. The image field is put above the string field. If you don’t check this item

then the settings of the fields ‘**Title at string field**’, ‘**Hide string field**’, and ‘**Size**’ are meaningless.

- a field ‘**Title at string field**’. If checked, the title of the attribute object is placed to the left of the string field, thus the image field gets more space in the window. If you don’t check this field then the title is placed next to the image field.
  - a field ‘**Hide string field**’ for removing the string field from the user interface. If checked, only the image field is displayed.
  - a field ‘**Size**’ for specifying how size-handling is done for the image area. If ‘**Resizable**’ is active, the object can be resized and might become larger than the size of its image. ‘**Fixed**’ sets the object’s size to the size of the image. If the image sizes vary from record to record then the object is resized accordingly every time. ‘**Scrollable**’ adds two scrollers to the object allowing to view images that are larger than the visible region. If ‘**Scaled**’ is activated then the image is scaled to the size of the display object. ‘**Aspect-scaled**’ also scales the image but preserves its aspect ratio.
  - a field ‘**File pop-up**’ that, if checked, adds a pop-up button to the right of the string field. This buttons serves to open a file requester for choosing a filename.
  - a field ‘**Font pop-up**’ for adding a pop-up button that opens a font requester.
  - a field ‘**List-view pop-up**’. If checked, a pop-up button is attached to the right of the string field for opening a list-view pop-up where you can choose a string from a list. The label strings for the list-view pop-up are defined to the right of the ‘**List-view pop-up**’ field. The label items can be either ‘**Static**’ or ‘**Computed**’. If ‘**Static**’ is chosen then the list of strings can be entered in the label editor which is opened after pressing the ‘**Edit labels**’ button. For more information about the label editor, see [Section 14.2.3 \[Label editor\]](#), page 59. If ‘**Computed**’ is chosen then a trigger function can be entered in the ‘**Compute**’ field which is called whenever the pop-up button is pressed. This function should return a memo text with one label string per line, or NIL for an empty list (see [Section 15.29.12 \[Compute list-view labels\]](#), page 143). Only one of the fields ‘**File pop-up**’, ‘**Font pop-up**’ and ‘**List-view pop-up**’ can be made active.
  - a field ‘**View**’ that, if checked, adds a button to the right of the string field for launching an external viewer with the attribute contents as argument. This can be useful if you store filenames in the attribute and want to display the contents of a file by starting an external viewer. The external viewer can be specified in menu item ‘**Preferences – External viewer**’ (see [Section 7.1.3 \[External viewer\]](#), page 33).
- for attributes of type choice there is a field ‘**Kind**’ where you choose whether the attribute contents should be displayed by a ‘**Cycle button**’ or by a set of ‘**Radio buttons**’. If you select ‘**Cycle button**’ then you can set the title position to one of ‘**Left**’, ‘**Right**’, ‘**Top**’, or ‘**Bottom**’. If you select ‘**Radio buttons**’ then two check mark items ‘**Frame**’ and ‘**Horizontal**’ allow drawing a border around the radio buttons and specifying a horizontal layout.
  - for attributes of type real there is an integer field ‘**Num decimals**’ where you can enter the number of decimals for displaying the floating point values.

- for attributes of type time there is a choice field **Format** for choosing how time values are displayed and entered. You can select between **HH:MM:SS**, **MM:SS** and **HH:MM**. If **HH:MM** is chosen, the number of seconds are discarded for display and single numbers on input are regarded as the number of minutes.
- for reference attributes there is an **Extras** page that contains the following items:
  - a list-view field **Display** where you specify which contents of a referenced record should be displayed. You can multi-select several items in this list. If you select **Record number** then the record number of a referenced record is included in the display. The other items are the names of attributes in the referenced table. You can rearrange the order of items by drag and drop.
  - an area **Pop-up** where you specify which records of the referenced table should be made available for selection in the pop-up of the reference field and how they should be displayed. If **Records** is set to **All** then all records are available. If **Records** is set to **Match filter** then only the records of the referenced table that match the currently installed filter are available. If **Records** is set to **Computed** then a function for computing the set of records can be entered in the **Compute** field. This trigger function should return a list that is searched for the occurrence of records of the referenced table. Any such record found is shown in the pop-up. Non-record items are ignored. See [Section 15.29.13 \[Compute reference records\], page 144](#), for more information about this function. In addition to the records specified by the **Records** setting, the initial record and the currently shown record in the referenced table can be added to the pop-up list by activating the **Initial record** and **Current record** items respectively. The item **Use multi-column list** decides whether the fields selected in the **Display** list are shown in multiple columns, or if the items are concatenated by a separation character **-**.
  - an area **Quick search** where you specify how the keyboard type-ahead search for the reference item should be performed. Quick search allows to find entries in the referenced table when the reference field or the associated popup list view is the active object (see [Section 8.3 \[Changing records\], page 38](#)). If **Disabled** is selected then no searching is performed and any key presses are simply ignored. Choosing **In first order field** searches only in the attribute that has been specified as the first one for ordering the referenced table (see [Section 10.2 \[Order by attributes\], page 44](#)). If **In all fields** is active then searching is performed in all fields of the referenced table. Finally, in **Prefix search only** you specify whether a found entry has to start with the search pattern, or if the search pattern can appear anywhere in a field. Default is prefix search in the first order field as this kind of search can be performed quite efficiently.
  - a field **Show**. If checked then the GUI object for displaying the reference is created as a button. Clicking on this button will show you the referenced record in the table mask of the referenced table. For this the window the referenced table resides in is opened and brought to front if necessary.
  - a field **Auto show** that, if checked, adds a button to the right of the reference field for turning auto show mode on and off for this attribute. When turned on, the referenced table gets updated automatically with the referenced record whenever the user switches to another record.

- a field **Filter** that, if checked, adds a button to the right of the reference field for turning on and off the reference filter for this attribute. See [Section 9.2 \[Reference filter\]](#), page 43, for more information about reference filters.
- for virtual attributes the attribute object editor contains an **Extras** page with the following items:
  - a choice field **Kind** where you specify how the contents of the virtual attribute should be displayed. You can choose between **Boolean** that uses a check-mark field for displaying Boolean values, **Text** that uses a text field for displaying one line of text (including date, time, and numerical values), and **List** which uses a list-view for displaying a list of lines (e.g. the result of a select-from-where query).
  - if you set the **Kind** field to **Text** then two further fields are available: **Alignment** for specifying how the attribute contents should be presented and **Num decimals** for entering the number of decimals that should be used if the attribute contents are of type real.
  - if the **Kind** field has been set to **List** then further fields **Show titles**, **Tab chain** and **On double click** are available. If **Show titles** is checked, the first line of the attribute contents is displayed as a title row in the list-view. Otherwise no title row is displayed and the first line is ignored. If **Tab chain** is checked, the object is part of the focus chain that can be cycled through by pressing the **Tab** key. Uncheck this item if you want to skip over the element. In **On double click** you can specify which action should take place when the user double-clicks a list item. **Do nothing** ignores the double click. **Show record** displays the record corresponding to the clicked item in the corresponding table view. If you select **Call trigger** then the name of a trigger function can be entered to the right. This trigger function is called on each double click. For more information about the trigger function, including the arguments that are passed to it, see [Section 15.29.11 \[Double click trigger\]](#), page 143.
  - a field **Auto update**. If checked, the virtual attribute is recomputed automatically whenever a dependent item changes. This includes switching to a different record in a dependent table, changing a dependent attribute, adding or deleting a record in a dependent table, changing the user/admin mode of the project, or recompiling the project program. The dependencies for the virtual attribute are obtained automatically from the virtual attribute's compute function. Use menu item **Project - Export structure** for viewing all obtained dependencies (you might need to recompile the project's program in order to update the dependencies when you turn this feature on). Note that the virtual attribute is recomputed irrespective of the record a dependent attribute is changed in and irrespective of whether the dependent attribute's value was changed in the user interface or during a program execution. However, the auto-updating of virtual attributes usually happens only after all other program executions are completed. If **Auto update** is unchecked, the value of the virtual attribute is only computed when it is explicitly queried or set in a program function.
- For buttons there are the following additional fields:
  - a choice field **Kind** where you choose between **Text button** and **Image button**.
  - if you set the button kind to **Text button** then further fields **Title**, **Font**,



- ‘Background’ and ‘Default’ appear for entering the text to be displayed in the button, the font used for displaying the text, and the background settings.
- if the button kind has been set to ‘Image button’ then a field ‘Image’ for specifying the image to be displayed and a field ‘Size’ for specifying the size-handling of the image button are offered.

### 14.3.5 Text Editor

When you add a text object to the display list-view, or when you change one by double clicking on it, a window ‘Text’ is opened. This window contains the following items:

- a string field ‘Title’ for entering the text that should be displayed.
- a numerical field ‘Weight’ for specifying the horizontal weight of the text object.
- a choice field ‘Font’ for specifying the font of the text. If you leave this field empty, a default font is used.
- two fields ‘Background’ and ‘Default’ for specifying the background settings of the text object.
- two buttons ‘Ok’ and ‘Cancel’ for leaving the window.

When you are done with all settings, press the ‘Ok’ button to leave the window.

### 14.3.6 Image Editor

The image editor appears when you add a new image object or double click on an existing one. It contains the following items:

- a field ‘Weight’ for specifying the weight of the image object in the final window layout.
- a field ‘Image’ for specifying the image that should be displayed.
- a field ‘Size’ where you specify how resize-handling should be done. If you select ‘Resizable’ then the image object can be resized. ‘Fixed’ sets the object size to the size of the displayed image.
- two buttons ‘Ok’ and ‘Cancel’ for leaving the window.

When you are done with all settings, click on the ‘Ok’ button to close the window.

### 14.3.7 Space Editor

After you have added a space object to the display list-view, you can change its default settings by double clicking on it. This will open the ‘Space’ window containing the following items:

- a field ‘Delimiter’ that, if checked, displays a vertical or horizontal bar (depending on the layout of the parent object) in the center of the space object. This is useful for separating parts in the window layout.
- a numerical field ‘Weight’ for specifying the weight of the object.
- two fields ‘Background’ and ‘Default’ for specifying the background settings.
- two buttons ‘Ok’ and ‘Cancel’ for leaving the window.

After you are done with all settings, press the ‘Ok’ button to close the window.

### 14.3.8 Group Editor

After you have added a group object to the display list-view, you can change its settings by double clicking on it. This will open the ‘Group’ window offering the following items:

- a string field ‘Title’ for entering a title string that should be displayed centered above the group. If you leave this field empty then no title is displayed.
- a numerical field ‘Weight’ for specifying the weight of this object.
- two fields ‘Background’ and ‘Default’ for specifying the background settings.
- a field ‘Border’ that, if activated, draws a border frame around the group.
- a field ‘Horizontal’. If checked, the layout of the group is done horizontally and the group is listed in the display list-view as ‘HGroup’. Otherwise the group is organized vertically and the display list-view will show a ‘VGroup’ for this group.
- a field ‘Spacing’ that, if checked, adds some space between the group’s child objects. Otherwise no space is put between the objects.
- two buttons ‘Ok’ and ‘Cancel’ for leaving the window.

When you are done with all settings, press the ‘Ok’ button to leave the window.

### 14.3.9 Register Group Editor

Double click on a register group object in order to change its settings. This will open the ‘Register group’ window offering the following items:

- a numerical field ‘Weight’ for specifying the weight of this object.
- an area ‘Labels’ for specifying the label of each register page. You should specify exactly the same number of labels as there are elements in the register group. For more information on how to enter and edit the labels, see [Section 14.2.3 \[Label editor\]](#), page 59.
- two buttons ‘Ok’ and ‘Cancel’ for leaving the window.

When you are done with all settings, press the ‘Ok’ button to leave the window.

### 14.3.10 Window Editor

To change the settings of a window object, double click the window object. This will open the window editor containing the following items:

- a string field ‘Title’ where you enter a string that should be displayed in the window title bar and in the optional window button.
- a string field ‘Id’ (Amiga only) which can hold up to four characters defining the MUI window id. If an id is entered, the window position and size can be saved using MUI’s snapshot window facility. Note that all ids defined within MUIbase should be unique. If you accidentally reuse an already taken id then the windows will share the same settings for their window dimensions. Ids beginning with an underscore ‘\_’ are reserved for internal use. If you leave the field empty then no MUI window id is set and the window dimensions are stored in the project file. The main window always has the first four characters of the project name as its window id. Specifying an window id is useful if a project is edited under different system configurations as the window positions are then taken from the individual configuration.



- a field **‘Button’** that, if checked, places a button for opening the window into the parent window. If this item is not checked then the window can only be opened from a MUIbase program using the SETWINDOWOPEN function (see [Section 15.21.3 \[SETWINDOWOPEN\]](#), page 129). The following items specify the appearance of the window button.
- a string field **‘Shortcut’** where you enter the shortcut for activating the window button.
- an area **‘Enabled/disabled’**. If **‘Always enabled’** is selected then the window button is always enabled. **‘Disabled in initial’** disables the button if it is inside a table and the table shows its initial record, otherwise it is enabled. If **‘Compute enabled’** is chosen then a function for computing the enabled state of the window button can be entered to the right. The function does not take any arguments. If it returns NIL then the button is disabled, otherwise enabled. In case the compute function is left empty or cannot be found, the window button gets disabled. For more information on how to use this trigger function, see [Section 15.29.10 \[Compute enabled function\]](#), page 142. Additionally if the field **‘Close window when disabled’** is checked then the window is closed automatically when the button gets disabled.
- a numerical field **‘Weight’** for specifying the weight of the window button.
- a field **‘Font’** for selecting the font of the window button. If you leave the field empty, a default font is used.
- two fields **‘Background’** and **‘Default’** for specifying the background settings of the window button.
- two buttons **‘Ok’** and **‘Cancel’** for leaving the window.

When you are done with all settings, press the **‘Ok’** button to close the window.

## 14.4 Export Structure

Sometimes it is useful to get an overview of all tables and attributes of a project, e.g. when you want to write a MUIbase program. You can do this by selecting menu item **‘Project – Export structure’**. This will ask you for a filename where the list of tables and attributes should be written to.

The output will first list the project name, followed by all tables in this project. For each table all attributes including their types and their optional trigger functions are listed. For virtual attributes also the dependencies for automatically updating the value of the virtual field are listed.

## 15 Programming MUIbase

This chapter (the largest in this manual) describes the programming language of MUIbase, including all available functions. This chapter, however, is not intended as a general guide about programming. You should be familiar with the basics about programming and should already have written some small (and correctly working :-)) programs.

### 15.1 Program Editor

To enter a program for a project, open the program editor by selecting menu item **‘Program - Edit’**. If you are using the setting program source internal (see [Section 7.2.5 \[Program source\]](#), [page 35](#)), this will open the **‘Edit program’** window containing:

- a text editor field where you edit the project program.
- a button **‘Compile & close’** for compiling the program and, on successful compilation, leaving the program editor.
- a button **‘Compile’** to compile the program. If your program contains an error somewhere then an error description is put into the window title and the cursor is set on the faulty location.
- a button **‘Revert’** that reverts all changes since the last successful compilation.

The program editor is a non-modal window. You can leave the window open and still work with the rest of the application. You can close the editor at any time by clicking into its window close button. If you have done changes since the last successful compilation then a safety requester pops-up asking for confirmation to close the window.

In case you have set menu item **‘Program - Source’** to **‘External’** the external editor (see [Section 7.1.2 \[External editor\]](#), [page 32](#)) is launched with the filename of the external source file when choosing **‘Program - Edit’**. This allows you to edit the program source using your favorite editor (see [Section 15.2 \[External program source\]](#), [page 70](#)).

You can also compile a project’s program without opening the program editor by choosing menu item **‘Program - Compile’**. This can be useful if you e.g. do changes to an external include file and want to incorporate these changes in the project’s program.

### 15.2 External program source

By choosing menu item **‘Program - Source - External’** and entering a filename you can make the program source of a project externally available. This allows you to load the program source into your favorite editor for programming.

If compilation is successful, the compiled program is set as the project’s program and used when executing trigger functions. When you save a project, the last successfully compiled program is store with the project inside the project file. Thus, after saving and closing a project, the external source is no longer needed actually. You can specify if unneeded external source files should be deleted automatically by checking menu item **‘Preferences - Cleanup external program source’**.

The status of menu item **‘Program - source’** is stored with the project. If you re-open a project that uses the external source feature, the external source file is created after opening the project. If the external source file already exists and is different from the version stored inside the project, a safety requester asks for confirmation before overwriting the file.

On the Amiga you can send the `compile` command to MUIbase' ARexx port from your editor. MUIbase then reads the external program source, compiles it, and returns the compile status with an optional error message containing filename, line and column, and an error description. This allows you to place the cursor at the exact location of where a compile error occurred. See [Section 16.9 \[ARexx compile\]](#), [page 147](#), for details on return values and error format.

## 15.3 Preprocessing

MUIbase programs are pre-processed similar to a C compiler preprocessing a C source file. This section describes how to use the preprocessing directives.

All directives start with a hash symbol `#` which should be the first character on a line. Space or tab characters can appear after the initial `#`.

### 15.3.1 `#define`

```
#define name string
```

Defines a new symbol with the given name and contents. The symbol *string* can be any text including spaces and ends at the end of line. If *string* does not fit on one line you can use further lines by using a backslash character `\` at the end of each line (except for the last one). If the symbol *name* does occur in the remaining source code then it will be replaced by the contents of *string*.

Example: `'(PRINTF "X is %i" X)'` prints `'X is 1'` (Occurrences of *name* in strings are not altered.)

The replacement of defined symbols is done syntactically which means that you can replace symbols with any text, e.g. you can define your own syntax like in the following example:

```
#define BEGIN (
#define END )

BEGIN defun test ()
    ...
END
```

The replacement string of a definition may contain another symbol that has been defined with the `#define` directive to allow nested definitions. However there is an upper limit of 16 nested definitions.

See also `#undef`, `#ifdef`, `#ifndef`.

### 15.3.2 `#undef`

```
#undef name
```

Removes the definition of symbol *name*. If *name* has not been defined, nothing happens.

See also `#define`, `#ifdef`, `#ifndef`.

### 15.3.3 `#include`

```
#include filename
```

Reads in the contents of *filename* (a string surrounded with double quotes) at this location. MUIbase searches in the current directory and in the directory specified in the preferences (see [Section 7.2.10 \[Program include directory\]](#), page 36) for loading the file. The file contents are processed by the compiler as if they were part of the current source code.

An external file may include one or more other external files. However there is a limit of 16 nested `#include` directives. To protect files from including them more than once, you can use conditional compilation.

Be careful when moving source code to external files! Debugging and tracking down errors is much harder for external files. Move only well tested and project independent code to external files.

### 15.3.4 `#if`

`#if const-expr`

If the given constant expression results to non-NIL then the text up to the matching `#else`, `#elif`, or `#endif` is used for compilation, otherwise (the expression results to NIL) the text up to the matching `#else`, `#elif`, or `#endif` is discarded for compilation.

Currently you can only use TRUE and NIL as constant expressions.

See also `#ifdef`, `#ifndef`, `#elif`, `#else`, `#endif`.

### 15.3.5 `#ifdef`

`#ifdef name`

If the given symbol has been defined with a `#define` directive then the text up to the matching `#else`, `#elif`, or `#endif` is used for compilation, otherwise it is discarded.

See also `#if`, `#ifndef`, `#elif`, `#else`, `#endif`.

### 15.3.6 `#ifndef`

`#ifndef name`

If the given symbol has not been defined with a `#define` directive then the text up to the matching `#else`, `#elif`, or `#endif` is used for compilation, otherwise it is discarded.

See also `#if`, `#ifdef`, `#elif`, `#else`, `#endif`.

### 15.3.7 `#elif`

`#elif const-expr`

Any number of `#elif` directives may appear between an `#if`, `#ifdef`, or `#ifndef` directive and a matching `#else` or `#endif` directive. The lines following the `#elif` directive are used for compilation only if all of the following conditions hold:

- The constant expression in the preceding `#if` directive evaluated to NIL, the symbol name in the preceding `#ifdef` is not defined, or the symbol name in the preceding `#ifndef` directive was defined.
- The constant expression in all intervening `#elif` directives evaluated to NIL.
- The current constant expression evaluates to non-NIL.

If the above conditions hold then subsequent `#elif` and `#else` directives are ignored up to the matching `#endif`.

See also `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`.

### 15.3.8 `#else`

`#else`

This inverts the sense of the conditional directive otherwise in effect. If the preceding conditional would indicate that lines are to be included, then lines between the `#else` and the matching `#endif` are ignored. If the preceding conditional indicates that lines would be ignored, subsequent lines are included for compilation.

Conditional directives and corresponding `#else` directives can be nested. However there is a maximum nesting count limit of 16 nested conditional directives

See also `#if`, `#ifdef`, `#ifndef`, `#elif`, `#endif`.

### 15.3.9 `#endif`

`#endif`

Ends a section of lines begun by one of the conditional directives `#if`, `#ifdef`, or `#ifndef`. Each such directive must have a matching `#endif`.

See also `#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`.

## 15.4 Programming Language

MUIbase uses a programming language with a lisp-like syntax. Indeed several constructs and functions have been adopted from standard lisp. However, MUIbase is not fully compatible to standard lisp. Many functions are missing (e.g. destructive commands) and the meaning of some commands is different (e.g. the `return` command).

### 15.4.1 Why Lisp?

The advantage of a lisp-like language is that you can program in both, a functional and an imperative way. Functional languages are getting quite popular in mathematical applications. The basic concept in functional languages is the use of expressions. Functions are defined in a mathematical way and recursion is used heavily.

Imperative programming languages (e.g. C, Pascal, Modula) use an imperative description on how to compute things. Here, the state is the basic concept (e.g. variables) and a program computes its output by going from one state to another (e.g. by assigning values to variables).

Lisp combines both techniques and therefore you can choose in which way you want to implement things. Use the one which is more appropriate for the specific problem or which you like more.

### 15.4.2 Lisp Syntax

A lisp expression is either a constant, a variable, or a function application. For calling functions, lisp uses a prefix notation. The function and its arguments are surrounded by parenthesis. For example, to add two values `a` and `b`, you write

(+ a b)

All expressions return a value, e.g. in the above example the sum of **a** and **b** returned. Expressions can be nested, that is, you can place an expression as a sub-expression into another one.

Function evaluation is done by using a call-by-value scheme, this means that the arguments are evaluated first before a function is called.

If not stated otherwise, all functions are strict, that is, *all* arguments of a function are evaluated before the function is called. Some functions, however, are non-strict, e.g. **IF**, **AND** and **OR**. These functions may not evaluate all arguments.

### 15.4.3 Kinds of Programs

MUIbase knows three kinds of programs. The first one is the project program kind. In a program of this kind you can define functions and global variables. The functions can be used as trigger functions for attributes. You define a project program in the program editor (see [Section 15.1 \[Program editor\]](#), page 70).

The second kind is the query program kind. For this kind you can enter expressions only. An expression is allowed to contain global variables and calls to functions that are defined in the project program. However, you cannot define new global variables or functions in a query program. Query programs are entered in the query editor (see [Section 13.2 \[Query editor\]](#), page 51).

The third kind of programs are filter expressions. Here you can only enter expressions that contain calls to pre-defined MUIbase functions. Not all pre-defined functions are available, only those that don't have a side effect, e.g. you cannot use a function that writes data to a file. Filter expressions are entered in the change filter requester (see [Section 9.1.2 \[Changing filters\]](#), page 42).

### 15.4.4 Name Conventions

In a MUIbase program you can define symbols like functions and local or global variables. The names of these symbols must follow the following conventions:

- the first character of a name must be a lowercase letter. This distinguishes program symbols from table and attribute names.
- The following characters can be any letters, digits or underscore characters. Other characters like German umlauts are not allowed.

### 15.4.5 Accessing Record Contents

To access tables and attributes in a MUIbase program, you have to specify a path to them. A path is a dot separated list of components where each component is the name of a table or an attribute.

Paths can either be relative or absolute. Absolute paths are specified with a table name as the first component, followed by a list of attributes that lead to the final attribute you want to access. E.g. the absolute path '**Person.Name**' accesses the '**Name**' attribute in the current record of table '**Person**', or the absolute path '**Person.Father.Name**' accesses the '**Name**' attribute in the record referenced by the '**Father**' field (a reference attribute to table '**Person**').

Relative paths already have a current table to which they are relative. For example in a filter expression the current table is the table for which you write the filter expression. The relative path for an attribute in the current table is simply the attribute name itself. For attributes that are not directly accessible from the current table but indirectly via a reference attribute, the same rules as for absolute paths apply.

It is not always clear if a specified path is a relative or an absolute one, e.g. suppose you are writing a filter expression for a table 'Foo' that has an attribute 'Bar' and there also exists a table 'Bar'. Now entering 'Bar' would be ambiguous, what is meant, the table or the attribute? Therefore all paths are first treated as relative ones. If no attribute is found for the specified path than the path is treated as global. In our example the attribute would be preferred.

But now, what if you want to access the table in the above example? Therefore the path must be given absolute. To indicate that a path is global you have to insert two colons in front of the path. In our example you would have to type '::Bar' for accessing the table.

To give you a better understanding of paths and their semantics, consider in our example that the 'Bar' attribute in the 'Foo' table is a reference to the 'Bar' table, and the 'Bar' table has an attribute 'Name'. Now you can access the 'Name' attribute by typing 'Bar.Name' or '::Bar.Name'. Both expressions have a different meaning. '::Bar.Name' means to take the current record of table 'Bar' and return the value of the 'Name' attribute of this record, whereas 'Bar.Name' takes the current record of table 'Foo', extracts the record reference of the 'Bar' field and uses this record for getting the value of the 'Name' attribute.

To make a more complete example, consider that table 'Bar' has two records. One that contains 'Ralph' and one that contains 'Steffen' in the 'Name' field. The first record should be the current one. Furthermore table 'Foo' has one record (the current one) whose 'Bar' field refers to the second record of table 'Bar'. Now '::Bar.Name' results to 'Ralph' and 'Bar.Name' to 'Steffen'.

### 15.4.6 Data Types for Programming

The programming language of MUIbase knows of the following data types:

Type	Description
Boolean	all expressions. Non-NIL expressions are treated as TRUE.
Integer	long integer, 32 bit, choice values are automatically converted to integers
Real	double, 64 bit
String	strings of arbitrary length
Memo	like strings but line oriented format
Date	date values
Time	time values

Record	pointer to a record
File	file descriptor for reading/writing
List	list of items, NIL is empty list.

All programming types support the NIL value.

### 15.4.7 Constants

The programming language of MUIbase can handle constant expressions which can be entered depending on the type of the expression:

Type	Description																						
Integer	Integer constants in the range of -2147483648 to 2147483647 can be specified as usual. Values starting with 0 are interpreted as octal numbers, values starting with 0x as hexadecimal numbers.																						
Real	Floating point constants in the range of -3.59e308 to 3.59e308 can be specified as usual, in scientific or non-scientific format. If you omit the decimal point then the number may not be treated as a real number but as an integer instead.																						
String	String constants are any character strings surrounded by double quotes, e.g. "sample string". Within the double quotes you can enter any characters except control characters or new lines. However there are special escape codes for entering such characters: <table data-bbox="479 1375 1177 1753"> <tr><td><code>\n</code></td><td>new line (nl)</td></tr> <tr><td><code>\t</code></td><td>horizontal tabulator (ht)</td></tr> <tr><td><code>\v</code></td><td>vertical tabulator (vt)</td></tr> <tr><td><code>\b</code></td><td>backspace (bs)</td></tr> <tr><td><code>\r</code></td><td>carriage return (cr)</td></tr> <tr><td><code>\f</code></td><td>form feed (ff)</td></tr> <tr><td><code>\\</code></td><td>backslash character itself</td></tr> <tr><td><code>\"</code></td><td>double quote</td></tr> <tr><td><code>\e</code></td><td>escape code 033</td></tr> <tr><td><code>\nnn</code></td><td>character with octal code <i>nnn</i></td></tr> <tr><td><code>\xnn</code></td><td>character with hex code <i>nn</i></td></tr> </table>	<code>\n</code>	new line (nl)	<code>\t</code>	horizontal tabulator (ht)	<code>\v</code>	vertical tabulator (vt)	<code>\b</code>	backspace (bs)	<code>\r</code>	carriage return (cr)	<code>\f</code>	form feed (ff)	<code>\\</code>	backslash character itself	<code>\"</code>	double quote	<code>\e</code>	escape code 033	<code>\nnn</code>	character with octal code <i>nnn</i>	<code>\xnn</code>	character with hex code <i>nn</i>
<code>\n</code>	new line (nl)																						
<code>\t</code>	horizontal tabulator (ht)																						
<code>\v</code>	vertical tabulator (vt)																						
<code>\b</code>	backspace (bs)																						
<code>\r</code>	carriage return (cr)																						
<code>\f</code>	form feed (ff)																						
<code>\\</code>	backslash character itself																						
<code>\"</code>	double quote																						
<code>\e</code>	escape code 033																						
<code>\nnn</code>	character with octal code <i>nnn</i>																						
<code>\xnn</code>	character with hex code <i>nn</i>																						
Memo	Same as string constants.																						
Date	Constant date values can be specified in one of the																						



formats ‘DD.MM.YYYY’, ‘MM/DD/YYYY’, or ‘YYYY-MM-DD’, where ‘DD’, ‘MM’ and ‘YYYY’ are standing for two and four digit values representing the day, month and year of the date respectively.

**Time** Constant time values can be entered in the format ‘HH:MM:SS’, where ‘HH’ represent the hours, ‘MM’ the minutes, and ‘SS’ the seconds.

For some other pre-defined constant values, see [Section 15.25 \[Pre-defined constants\]](#), page 135.

### 15.4.8 Command Syntax

In the remainder of this chapter, you will find the description of all commands and functions available for programming MUIbase. The following syntax is used for describing the commands:

- text written in brackets `[]` is optional. If you omit the text inside the brackets then a default value is assumed.
- text separated by a vertical bar `|` denotes several options, e.g. ‘`a | b`’ means that you can specify either ‘`a`’ or ‘`b`’.
- text written in a font like `var` indicates a place-holder that can be filled with other expressions.
- dots `...` indicate that further expressions can follow.
- all other text is obligatory.

## 15.5 Defining Commands

This section lists commands for defining functions and global variables. The commands are only available for project programs.

### 15.5.1 DEFUN

DEFUN defines a function with the specified name, a list of arguments that are passed to the function, and a list of expressions to evaluate.

```
(DEFUN name (varlist) expr ...)
```

The name of a function must start with a lowercase character, followed by further characters, digits or underscore characters (see [Section 15.4.4 \[Name conventions\]](#), page 74).

The parameters *varlist* specifies the arguments of the function:

```
varlist: var1 ...
```

where *var1* ... are the names for the arguments. The names must follow the same rules as the function name.

It is also possible to add type specifiers to the arguments (see [Section 15.27 \[Type specifiers\]](#), page 136).

The function executes the expressions *expr*, ... one by one and returns the value of the last expression. The function may call further functions including itself. A self defined function can be called like calling a pre-defined function.

For example to count the number of elements of a list, you can define the following function:

```
(DEFUN len (l)
  (IF (= l NIL)
    0
    (+ 1 (len (REST l)))
  )
)
```

Functions defined with DEFUN are listed in the pop-up list-views of table and attribute requesters (see [Section 14.1.1 \[Creating tables\]](#), page 56 and [Section 14.2.1 \[Creating attributes\]](#), page 57).

This command is only available for project programs.

See also DEFUN\*, DEFVAR.

### 15.5.2 DEFUN\*

DEFUN\* is the star version of DEFUN and has the same effect as DEFUN (see [Section 15.5.1 \[DEFUN\]](#), page 77). The only difference is that functions defined with DEFUN\* are not listed in the pop-up list-views when creating or changing tables and attributes. However, it is still possible to enter the function name in the corresponding string fields.

This command is only available for project programs.

See also DEFUN, DEFVAR.

### 15.5.3 DEFVAR

```
(DEFVAR var [expr])
```

Defines a global variable with initial value of *expr* or NIL if *expr* is missing. The name of the variables must start with a lowercase letter followed by further letters, digits or underscore characters (see [Section 15.4.4 \[Name conventions\]](#), page 74).

You can also add a type specifier to the variable name (see [Section 15.27 \[Type specifiers\]](#), page 136).

DEFVAR is only available for project programs. All DEFVAR commands should be placed on top of all function definitions.

After execution of a trigger function (when MUIbase returns to the user interface), all global variables loose their contents. They are re-initialized with their initial value *expr* on the next invocation of a trigger function. If this is not desired, use the DEFVAR\* (see [Section 15.5.4 \[DEFVAR\\*\]](#), page 79) command which allows to store the value of global variables between program calls.

Please use global variables rarely (if at all). All global variables have to be initialized (and *expr* be evaluated if given) whenever a trigger function is called from outside.

Example: '(DEFVAR x 42)' defines a global variable 'x' with value 42.

There are some pre-defined global variables in MUIbase (see [Section 15.24 \[Pre-defined variables\]](#), page 135).

See also DEFVAR\*, DEFUN, DEFUN\*, LET.

#### 15.5.4 DEFVAR\*

(DEFVAR\* *var* [*expr*])

DEFVAR\* has the same effect as the DEFVAR command (see [Section 15.5.3 \[DEFVAR\]](#), [page 78](#)) except that a variable defined with DEFVAR\* does not lose its value after the program completes.

On the first invocation of the program, *var* is initialized with *expr* or NIL if *expr* is omitted. Subsequent calls of the program do not evaluate *expr* again but use the value of *var* from the previous call. This way it is possible to transfer information from one program call to another without storing data in an external file or a database table. Note, however, that all global variables defined with DEFVAR\* lose their contents when the project program is re-compiled. If you wish to permanently store information, use a (possibly hidden) attribute in a table.

See also DEFVAR, DEFUN, DEFUN\*, LET.

### 15.6 Program Control Functions

This section lists functions for program control, e.g. functions for defining local variables, loop functions, conditional program execution, loop control functions and more.

#### 15.6.1 PROGN

To evaluate several expressions one by another the PROGN construction can be used.

(*expr* ...)

executes *expr* ... one by one. Returns the result of the last expression (or NIL if no expression has been specified). In Lisp this construct is known as (PROGN [*expr* ...]).

Example: '(1 2 3 4)' results to 4.

See also PROG1.

#### 15.6.2 PROG1

Another way, besides the PROGN function, to evaluate several expressions one by another is the PROG1 expression.

(PROG1 [*expr* ...])

executes *expr* ... and returns the value of the first expression (or NIL if no expression has been specified).

Example: '(PROG1 1 2 3 4)' results to 1.

See also PROGN.

#### 15.6.3 LET

LET defines a new block of local variables. This is useful, e.g., for defining local variables of a function. The syntax is

(LET (*varlist*) *expr* ...)

where *varlist* is a list of local variables.

*varlist*: *varspec* ...

**varspec**: (*var expr*) | *var*

Here *var* is the name of the variable and must start with a lowercase character, followed by further characters, digits or underscore characters (see [Section 15.4.4 \[Name conventions\]](#), [page 74](#)).

In the case of a (*var expr*) specification, the new variable is initialized with the given expression. In the other case the new variable is set to NIL.

It is also possible to add type specifiers to the variables. (see [Section 15.27 \[Type specifiers\]](#), [page 136](#)).

After initializing all variables the list of expressions *expr* . . . are evaluated and the value of the last one is returned.

For example, the following LET expression

```
(LET ((x 0) y (z (+ x 1)))
      (+ x z)
    )
```

results to 1.

See also DOTIMES, DOLIST, DO, DEFVAR.

#### 15.6.4 SETQ

The SETQ function sets values to variables, attributes and tables.

```
(SETQ lvalue1 expr . . .)
```

Sets *lvalue1* to the value of *expr*. The dots indicate assignments for further lvalues. An lvalue is either a variable, an attribute in a table, or a table. In case of a variable, the variable must have been previously defined (e.g. by a LET expression).

Setting the value of a table means setting its program or GUI record pointer: ‘(SETQ *Table expr*)’ sets the program record pointer of *Table* to the value of *expr*, ‘(SETQ *Table\* expr*)’ sets the GUI record pointer of it and updates the display. For more information about program and GUI record pointers, see [Section 5.2 \[Tables\]](#), [page 18](#).

SETQ returns the value of the last expression.

Example: ‘(SETQ a 1 b 2)’ assigns 1 to the variable ‘a’, 2 to the variable ‘b’ and returns 2.

See also SETQ\*, LET, DEFVAR, Tables, Semantics of expressions.

#### 15.6.5 SETQ\*

SETQ\* is the star version of SETQ (see [Section 15.6.4 \[SETQ\]](#), [page 80](#)) and has similar effects. The difference is that when assigning a value to an attribute, SETQ\* calls the trigger function of that attribute (see [Section 15.29.8 \[Attribute trigger\]](#), [page 141](#)) instead of directly assigning the value. In case no trigger function has been specified for an attribute, SETQ\* behaves like SETQ and simply assigns the value to the attribute.

Example: ‘(SETQ\* Table.Attr 0)’ calls the trigger function of ‘Table.Attr’ with an argument of 0.

Warning: With this function it is possible to write endless loops, e.g. if you have defined a trigger function for an attribute and this function calls SETQ\* to set a value to itself.

See also SETQ\*, LET, DEFVAR.

### 15.6.6 FUNCALL

FUNCALL is used to call a function with arguments.

```
(FUNCALL fun-expr [expr ...])
```

Calls the function *fun-expr* with the given arguments. The expression *fun-expr* can be any expression whose value is a pre-defined or user-defined function, e.g. a variable holding the function to call. If the number of arguments is not correct, an error message is generated.

FUNCALL returns the return value of the function call or NIL if *fun-expr* is NIL.

For more information about functional expressions, see [Section 15.26 \[Functional parameters\]](#), page 136.

See also APPLY.

### 15.6.7 APPLY

APPLY is used to apply a function to a list of arguments.

```
(APPLY fun-expr [expr ...] list-expr)
```

Applies the function *fun-expr* to a list created by **cons**'ing the arguments *expr* ... to *list-expr*. In other words: calls the function *fun-expr* with the arguments *expr* ... and *list-expr* expanded to its list elements.

The expression *fun-expr* can be any expression whose value is a pre-defined or user-defined function, e.g. a variable holding the function to call. The last argument *list-expr* must be a valid list or NIL, otherwise an error message is generated. If the number of arguments is not correct, an error occurs.

APPLY returns the return value of the function call or NIL if *fun-expr* is NIL.

For more information about functional expressions, see [Section 15.26 \[Functional parameters\]](#), page 136.

Example: '(APPLY + 4 (LIST 1 2 3))' returns 10.

See also FUNCALL.

### 15.6.8 IF

IF is a conditional operator.

```
(IF expr1 expr2 [expr3])
```

The expression *expr1* is tested. If it results to non-NIL then the value of *expr2* is returned else the value of *expr3* (or NIL if not present) is returned.

This function is not strict, that is, only one expression of *expr2* or *expr3* will be evaluated.

See also CASE, COND.

### 15.6.9 CASE

CASE is similar to the **switch** statement in the C language.

```
(CASE expr [case ...])
```

Here *expr* is the selection expression and *case* ... are pairs consisting of:

```
case: (value [expr ...])
```

where *value* is a single expression or a list of expressions and *expr ...* are the expressions to be executed if one of the case expression matches.

The CASE expression first evaluates *expr*. Then each case pair is checked whether it (or one of the expressions in the list) matches the evaluated expression. If a matching case expression is found then the corresponding expressions are executed and the value of the last expression is returned. If no case matches, NIL is returned.

Example: '(CASE 1 ((2 3 4) 1) (1 2))' returns 2.

See also IF, COND.

### 15.6.10 COND

COND is, like IF, a conditional operator.

```
(COND [(test-expr [expr ...]) ...])
```

COND test the first expression of each list one by one. For the first one that does not result to NIL, the corresponding expressions *expr ...* are evaluated and the value of the last expression is returned.

If all tested expressions result to NIL then NIL is returned.

#### Example

```
(COND ((> 1 2) "1 > 2")
      ((= 1 2) "1 = 2")
      ((< 1 2) "1 < 2")
      )
```

results to "1 < 2".

See also IF, CASE.

### 15.6.11 DOTIMES

For simple loops the DOTIMES command can be used.

```
(DOTIMES (name int-expr [result-expr ...]) [loop-expr ...])
```

Here *name* is the name of a new variable which will be used in the loop. The name must start with a lowercase character, followed by further characters, digits or underscore characters (see [Section 15.4.4 \[Name conventions\]](#), page 74).

The number of times the loop is executed is given in *int-expr*. In *result-expr ...* expressions can be specified that are executed after terminating the loop. In *loop-expr* you specify the body of the loop, that is, the expressions that are evaluated in each loop run.

Before executing the loop, DOTIMES computes the value of *int-expr* to determine the number of times the loop gets executed. Here *int-expr* is evaluated only once at the start of the loop and must result to an integer value. Then DOTIMES sets the loop variable to the values of 0 to *int-expr*-1 one by one for each loop run. First, the variable is initialized with zero and checked if it is already greater or equal to the value of *expr*. If *int-expr* is negative or NIL or if the variable is greater or equal to the value of *expr* then the loop is terminated and the result expressions are evaluated. Otherwise the loop expressions are evaluated and the variable is incremented by one. Then the execution returns to the termination test and, possibly, does further loop runs.

The **DOTIMES** expression returns the value of the last result expression or **NIL** if no result expression has been given.

### Example

```
(DOTIMES (i 50 i) (PRINT i))
```

Prints the numbers from 0 to 49 and returns the value 50.

See also **DOLIST**, **DO**, **FOR ALL**, **LET**.

## 15.6.12 DOLIST

For loops through lists the **DOLIST** expression can be used.

```
(DOLIST (name list-expr [result-expr ...]) [loop-expr ...])
```

Here *name* is the name of a new variable which will be used in the loop. The name must start with a lowercase character, followed by further characters, digits or underscore characters (see [Section 15.4.4 \[Name conventions\]](#), page 74).

In *list-expr* you specify the list over which the loop should be executed, *result-expr ...* are expressions which are evaluated after terminating the loop, and *loop-expr ...* build the body of the loop.

Before executing the loop, **DOLIST** computes the value of *list-expr*. This expression is evaluated only once at the start of the loop and must result to a list value. Then **DOTIMES** sets the loop variable to the nodes of the list one by one for each loop run. First the loop variable is initialized to the first node of the list. If the list is already empty (**NIL**) then the loop is terminated and the result expressions are evaluated. Otherwise the loop expressions are evaluated and the variable is set to the next node of the list. Then the execution returns to the termination test and, possibly, does further loop runs.

The **DOLIST** expression returns the value of the last result expression or **NIL** if no result expression has been given.

### Example

```
(DOLIST (i (SELECT * FROM Accounts)) (PRINT i))
```

Prints all records of table 'Accounts' and returns **NIL**.

See also **DOTIMES**, **DO**, **FOR ALL**, **LET**.

## 15.6.13 DO

With the **DO** expression arbitrary loops can be programmed.

```
(DO ([binding ...]) (term-expr [result-expr ...]) [loop-expr ...])
```

where *binding ...* are the variable bindings, each of which is either:

- a new name for a variable (which is initialized to **NIL**)
- a list of the form: (*name init [step]*) where *name* is a name for the new variable, *init* is the initial value of the variable, and *step* is a step expression.

Furthermore, *term-expr* is the termination test expression, *result-expr ...* are the result expressions (the default is **nil**) and *loop-expr ...* build the body of the loop.

The **DO** loop first initializes all local variables with the *init* expressions, then tests the termination expression. If it results to **TRUE**, the loop is terminated and the result expressions are evaluated. The value of the last result expression is returned. Otherwise the

body of the loop (*loop-expr* ...) is executed and each variable is updated by the value of its step expression. Then the execution returns to test the terminating expression and so on.

### Example

```
(DO ((i 0 (+ i 1))) (>= i 5) i) (PRINT i))
```

Prints the values 0, 1, 2, 3 and 4 and returns the value 5. Of course this is a quite complicated way to build a simple FOR loop. Therefore a simpler version exists, the DOTIMES expression.

See also DOTIMES, DOLIST, FOR ALL, LET.

### 15.6.14 FOR ALL

The FOR ALL expression is used to loop over a list of records.

```
(FOR ALL table-list [WHERE where-expr] [ORDER BY order-list] DO expr ...)
```

Here *table-list* is a comma-separated list of tables, *where-expr* an expression to test each set of records, *order-list* a comma-separated list of expressions by which the record sets are ordered, and *expr* ... are the expressions that are executed for each record set.

FOR ALL first generates a list of all record sets for which the loop body should be executed. This is done like in the SELECT expression. Please see [Section 15.20.12 \[SELECT\], page 128](#), for more information about how this list is generated. For each element of this list the loop body *expr* ... is executed.

For example, summing up an attribute of a table can be done in the following way:

```
(SETQ sum 0)
(FOR ALL Accounts DO
  (SETQ sum (+ sum Accounts.Amount))
)
```

The FOR ALL expression returns NIL.

See also SELECT, DOTIMES, DOLIST, DO.

### 15.6.15 NEXT

NEXT can be used for controlling DOTIMES, DOLIST, DO and FOR ALL loops.

Calling NEXT in the body of a loop will jump to the next loop iteration. This can be used for skipping non-interesting loop runs, like e.g. in the following example:

```
(FOR ALL Table DO
  (IF not-interested-in-the-current-record (NEXT))
  ...
)
```

See also EXIT, DOTIMES, DOLIST, DO, FOR ALL.

### 15.6.16 EXIT

EXIT can be used to terminate a loop.

```
(EXIT [expr ...])
```



EXIT within a loop body will terminate the loop, execute the optional expressions *expr* ..., and return the value of the last expression (or NIL in case of no expression) as the return value of the loop. Possible return expressions of the loop as for example in

```
(DOTIMES (x 10 ret-expr ...) ...)
```

are not executed.

You may use the EXIT function for example to end a FOR ALL loop when you found the record you are interested in:

```
(FOR ALL Table DO
  (IF interested-in-the-current-record (EXIT Table))
  ...
)
```

See also NEXT, RETURN, HALT, DOTIMES, DOLIST, DO, FOR ALL.

### 15.6.17 RETURN

Within a function definition you can return to the caller by using the RETURN command.

```
(RETURN [expr ...])
```

terminates the function, executes the optional expressions *expr* ..., and returns the value of the last expression (or NIL in case of no expression).

#### Example

```
(DEFUN find-record (name)
  (FOR ALL Table DO
    (IF (= Name name) (RETURN Table))
  )
)
```

The example searches a record whose Name attribute matches the given name. The function returns the first record found or NIL in case of no record found.

See also HALT, EXIT.

### 15.6.18 HALT

HALT can be used to terminate program execution.

```
(HALT)
```

stops program execution silently.

See also ERROR, EXIT, RETURN.

### 15.6.19 ERROR

For aborting program execution with an error message the ERROR function can be used.

```
(ERROR fmt [arg ...])
```

stops program execution and pops up a requester with an error message. The error message is generated from *fmt* and the optional arguments *arg* ... like in the SPRINTF function (see [Section 15.12.32 \[SPRINTF\]](#), [page 102](#)).

See also HALT, SPRINTF.

## 15.7 Type Predicates

For each type a predicate is defined that returns TRUE if the supplied expression is of the specified type and NIL otherwise. The predicates are:

Predicate	Description
(STRP <i>expr</i> )	TRUE if <i>expr</i> is of type string, NIL otherwise.
(MEMOP <i>expr</i> )	TRUE if <i>expr</i> is of type memo, NIL otherwise.
(INTP <i>expr</i> )	TRUE if <i>expr</i> is of type integer, NIL otherwise.
(REALP <i>expr</i> )	TRUE if <i>expr</i> is of type real, NIL otherwise.
(DATEP <i>expr</i> )	TRUE if <i>expr</i> is of type date, NIL otherwise.
(TIMEP <i>expr</i> )	TRUE if <i>expr</i> is of type time, NIL otherwise.
(NULL <i>expr</i> )	TRUE if <i>expr</i> is NIL (an empty list), NIL otherwise.
(CONSP <i>expr</i> )	TRUE if <i>expr</i> is a non-empty list, NIL otherwise.
(LISTP <i>expr</i> )	TRUE if <i>expr</i> is a list (may be NIL), NIL otherwise.
(RECP <i>table expr</i> )	TRUE if <i>expr</i> is a record pointer of the given table. If <i>expr</i> is NIL then TRUE is returned (initial record). If <i>table</i> is NIL then a check is done whether <i>expr</i> is a record pointer of any table.

## 15.8 Type Conversion Functions

This section lists functions for converting values from one type to another one.

### 15.8.1 STR

STR can be used to convert an expression into a string representation.

(STR *expr*)

converts *expr* into a string representation. The type of *expr* determines the conversion:

Type	Return string
String	The string itself.
Memo	The whole memo text in one string.
Integer	String representation of integer value.

Real	String representation of real value. If <i>expr</i> is an attribute then the number decimals specified for this attribute are used, otherwise 2 decimals are used.
Choice	Label string of the choice attribute.
Date	String representation of date value.
Time	String representation of time value.
Boolean	The string "TRUE"
NIL	User defined nil string if <i>expr</i> is an attribute, the string "NIL" otherwise.
Record	String representation of record number.
Others	String representation of internal address pointer.

See also MEMO, SPRINTF.

### 15.8.2 MEMO

MEMO can be used to convert an expression into a memo.

(MEMO *expr*)

converts *expr* into a memo representation. It treats the expression like the STR function (see [Section 15.8.1 \[STR\]](#), page 86) but returns a memo text instead of a string.

See also STR.

### 15.8.3 INT

INT is used to convert an expression into an integer.

(INT *expr*)

converts *expr* into an integer value. Possible conversions are:

Type	Return value
String	If the complete string represents a valid integer value, it is converted into an integer. A string starting with 0 is interpreted as an octal number, one starting with 0x as hexadecimal number. Leading and following spaces are ignored. If the string does not represent an integer value, NIL is returned.
Memo	Same as for string.

Integer	The value itself.
Real	If the value lies within the integer range than the real value is rounded and returned, otherwise NIL is returned.
Choice	The internal number (starting with 0) of the current label.
Date	Number of days since 01.01.0000.
Time	Number of seconds since 00:00:00.
Record	Record number.
NIL	NIL
Others	An error message is generated and program execution is aborted.

See also REAL, ASC.

#### 15.8.4 REAL

REAL is used to convert an expression into a value of type real.

`(REAL expr)`

converts *expr* into a real. It treats the expression like the INT function (see [Section 15.8.3 \[INT\]](#), page 87) but returns a value of type real instead of an integer.

See also INT.

#### 15.8.5 DATE

DATE is used to convert an expression into a date (with your favorite friend :-).

`(DATE expr)`

converts the given expression into a date value. Possible conversions are:

Type	Return value
String	If the whole string represents a date value then the string is converted into a date value. Leading and following spaces are ignored. If it does not represent a date value, NIL is returned.
Memo	Same as for string.
Integer	A date value is generated where the given integer represents the number of days since 01.01.0000. If the integer value is too great (date value would be greater than 31.12.9999) or negative then NIL is returned.

Real	Same as for integer.
Date	The value itself.
NIL	NIL
others	An error message is generated and program execution is aborted.

See also DATEDMY.

### 15.8.6 TIME

TIME is used to convert an expression into a time value.

`(TIME expr)`

converts the given expression into a time value. Possible conversions are:

Type	Return value
String	If the whole string represents a time value then the string is converted into a time value. Leading and following spaces are ignored. If it does not represent a time value, NIL is returned.
Memo	Same as for string.
Integer	A time value is generated where the given integer represents the number of seconds since 00:00:00.
Real	Same as for integer.
Time	The value itself.
NIL	NIL
others	An error message is generated and program execution is aborted.

## 15.9 Boolean Functions

This section lists Boolean operators.

### 15.9.1 AND

AND checks if all of its arguments are TRUE.

`(AND [expr ...])`

checks *expr* ... one by another until one expression evaluates to NIL. If all expressions evaluate to non-NIL then the value of the last expression is returned. Otherwise NIL is returned.

This function is non-strict which means that arguments of AND may not be evaluated, e.g. in '(AND NIL (+ 1 2))' the expression '(+ 1 2)' is not evaluated since a NIL value has already been processed, however in '(AND (+ 1 2) NIL)' the expression '(+ 1 2)' gets evaluated.

See also OR, NOT.

### 15.9.2 OR

OR checks if all of its arguments are NIL.

(OR [*expr* ...])

checks *expr* ... one by another until one expression evaluates to non-NIL. Returns the value of the first non-NIL expression or NIL if all expressions evaluate to NIL.

This function is non-strict which means that arguments of OR may not be evaluated, e.g. in '(OR TRUE (+ 1 2))' the expression '(+ 1 2)' is not evaluated since a non-NIL value (here 'TRUE') has already been processed, however in '(OR (+ 1 2) TRUE)' the expression '(+ 1 2)' gets evaluated.

See also AND, NOT.

### 15.9.3 NOT

NOT is used to invert the value of a Boolean expression.

(NOT *expr*)

results to TRUE if *expr* is NIL, NIL otherwise.

See also AND, OR.

## 15.10 Comparison Functions

In this section you will find functions for comparing values.

### 15.10.1 Relational Operators

For comparing two values in a MUIbase program, use

(*op* *expr1* *expr2*)

where *op* is in {=, <, <=, >, >=, <\*, <=\*, >\*, >=\*}. The star is used for special comparison (strings are compared case insensitive, records are compared by using the order defined by the user).

The following table shows all rules for comparing two values in a MUIbase program.

Type	Order relation
Integer	NIL < MIN_INT < ... < -1 < 0 < 1 < ... < MAX_INT
Real	NIL < -HUGE_VAL < ... < -1.0 < 0.0 < 1.0 < ... < HUGE_VAL
String:	NIL < "" < "Z" < "a" < "aa" < "b" < ... (case sensitive)

	<code>NIL &lt;* "" &lt;* "a" &lt;* "AA" &lt;* "b" &lt; ...</code>	(case insensitive)
Memo:	same as string	
Date	<code>NIL &lt; 1.1.0000 &lt; ... &lt; 31.12.9999</code>	
Time	<code>NIL &lt; 00:00:00 &lt; ... &lt; 596523:14:07</code>	
Boolean	<code>NIL &lt; TRUE</code>	
Record:	<code>NIL &lt; any_record</code>	(records itself are not comparable with <)
	<code>NIL &lt;* rec1 &lt;* rec2</code>	(order specified by user)

See also `CMP`, `CMP*`, `LIKE`.

### 15.10.2 `CMP`

`CMP` returns an integer representing the order of its arguments.

`(CMP expr1 expr2)`

returns a value less than 0 if `expr1` is smaller than `expr2`, 0 if `expr1` is equal to `expr2`, and a value greater than 0 if `expr1` is greater than `expr2`. For determining the order the simple (non-star) order relation as for relational operators (see [Section 15.10.1 \[Relational operators\]](#), page 90) is used.

Do not assume that the returned value will always be -1, 0, or 1!

Example: `'(CMP "Bike" "bIKE)'` results to -1.

See also `CMP*`, Relational operators.

### 15.10.3 `CMP*`

`CMP*` is the star version of `CMP`. The difference is that `CMP*` uses the extended ordering as defined for relational operators (see [Section 15.10.1 \[Relational operators\]](#), page 90) where strings are compared case insensitive and records are compared using the user defined record order.

Example: `'(CMP* "Bike" "bIKE)'` results to 0.

See also `CMP`, Relational operators.

## 15.11 Mathematical Functions

Here, some mathematical functions are listed.

### 15.11.1 Adding Values

For adding values, use

`(+ expr ...)`

Returns the sum of the arguments `expr ...`. If any argument value is `NIL` then the result is `NIL`. If the values are of type real or integer then a real (or integer) value is the result.

You may also add strings or memos. In this case the result is the concatenation of the strings/memos.

If *expr* is of type date and the rest of the arguments are integers/real then the sum of integers/real are interpreted as a number of days and added to *expr*. If the resulting date is out of range (smaller than 1.1.0000 or greater than 31.12.9999) then NIL is the result.

If *expr* is of type time and the rest of the arguments are integers/real or other time values then the sum of integers/real (interpreted as a number of seconds) and time values are added to *expr*. If the resulting time is out of range (smaller than 00:00:00 or greater than 596523:14:07) then NIL is the result.

## Examples

Expression	Value
(+ 1 2 3)	6
(+ 5 1.0)	6.0
(+ "Hello" " " "world!")	"Hello world!"
(+ 28.11.1968 +365 -28 -9)	22.10.1969
(+ 07:30:00 3600)	08:30:00
(+ 03:00:00 23:59:59)	26:59:59

See also -, 1+, \*, CONCAT, CONCAT2, ADDMONTH, ADDYEAR.

### 15.11.2 Subtracting Values

For subtracting values, use

```
(- expr1 expr2 ...)
```

Subtracts the sum of *expr2* ... from *expr1*. Here the same rules apply as for adding values (see [Section 15.11.1 \[add\], page 91](#)), except that strings and memos can't be subtracted.

(- *expr*) has a special meaning, it returns the negative value of *expr* (integer or real), e.g. '(- (+ 1 2))' results to -3.

See also +, 1-.

### 15.11.3 1+

1+ increases an integer or real expression by one.

```
(1+ expr)
```

Returns the value of *expr* (integer or real) plus one. If *expr* is NIL then NIL is returned.

See also +, 1-.

### 15.11.4 1-

1- decreases an integer or real expression by one.

```
(1- expr)
```

Returns the value of *expr* (integer or real) minus one. If *expr* is NIL then NIL is returned.

See also -, 1+.



### 15.11.5 Multiplying Values

For multiplying integer/real values, use

```
(* expr ...)
```

Returns the multiplication of the integer/real values *expr ...*. If all arguments are integers then an integer is returned, otherwise the result is a value of type real.

See also +, /.

### 15.11.6 Dividing Values

For dividing integer/real values, use

```
(/ expr1 [expr2 ...])
```

Divides *expr1* by the multiplication of the rest of the arguments. Returns a real value. On division by zero, NIL is returned.

See also \*, DIV, MOD.

### 15.11.7 DIV

DIV is used for integer division.

```
(DIV int1 int2)
```

Returns the integer division of *int1* with *int2*. For example, '(DIV 5 3)' results to 1.

See also /, MOD.

### 15.11.8 MOD

MOD is used for modulo calculation.

```
(MOD int1 int2)
```

Returns *int1* modulo *int2*. For example, '(MOD 5 3)' results to 2.

See also DIV.

### 15.11.9 MAX

MAX returns the argument that has the largest value.

```
(MAX expr ...)
```

Returns the maximum value of the arguments *expr ...* (all integers or reals). If one of the expressions is NIL then NIL is returned.

See also MIN.

### 15.11.10 MIN

MIN returns the argument that has the smallest value.

```
(MIN expr ...)
```

Returns the minimum value of the arguments *expr ...* (all integers or reals). If one of the expressions is NIL then NIL is returned.

See also MAX.

### 15.11.11 ABS

ABS computes the absolute value of an expression.

`(ABS expr)`

Returns the absolute value of *expr* (integer or real). If *expr* is NIL then NIL is returned.

### 15.11.12 TRUNC

TRUNC truncates decimals of a real value.

`(TRUNC real)`

Returns the largest integer (as a real number) not greater than the specified real number. If *real* is NIL then NIL is returned.

Examples: `'(TRUNC 26.1)'` results to 26, `'(TRUNC -1.2)'` results to -2.

See also ROUND.

### 15.11.13 ROUND

ROUND rounds a real value.

`(ROUND real digits)`

Returns the specified real number rounded to *digits* decimal digits. If *real* or *digits* are NIL then NIL is returned.

Examples: `'(ROUND 70.70859 2)'` results to 70.71, `'(ROUND 392.36 -1)'` results to 390.0.

See also TRUNC.

### 15.11.14 RANDOM

RANDOM can be used to generate random numbers.

`(RANDOM expr)`

Returns a random number. On the first call the random number generator is initialized with a value generated from the current time. RANDOM generates a random number in the range of 0 . . . *expr*, excluding the value of *expr* itself. The type of *expr* (integer or real) is the return type. If *expr* is NIL then NIL is returned.

#### Examples:

Example	Meaning
<code>(RANDOM 10)</code>	returns a value from 0 to 9,
<code>(RANDOM 10.0)</code>	returns a value from 0.0 to 9.99999...

### 15.11.15 POW

POW computes the power of values.

`(POW x y)`

Returns the value of real value *x* raised to the power of real value *y*. If either *x* or *y* are NIL, or if *x* is negative and *y* not an integral number, then NIL is returned.

Example: `'(POW 2 3)'` results to 8.

See also SQRT, EXP.

### 15.11.16 Sqrt

Sqrt computes the square root of a number.

(Sqrt *x*)

Returns the square root of real value *x*. If *x* is NIL or a negative number then NIL is returned.

See also POW.

### 15.11.17 EXP

EXP computes the exponential function.

(EXP *x*)

Returns the value of the base of the natural logarithm raised to the power of real value *x*. If *x* is NIL then NIL is returned.

See also POW, LOG.

### 15.11.18 LOG

LOG computes the natural logarithm of a number.

(LOG *x*)

Returns the natural logarithm of real value *x*. If *x* is NIL or not a positive number then NIL is returned.

See also EXP.

## 15.12 String Functions

This section deals with functions useful for strings.

### 15.12.1 LEN

LEN computes the length of a string.

(LEN *str*)

Returns the length of the given string or NIL if *str* is NIL.

See also WORDS, LINES, MAXLEN.

### 15.12.2 LEFTSTR

LEFTSTR extracts a sub string out of a string.

(LEFTSTR *str len*)

Returns the left part of the given string with at most *len* characters. If *str* or *len* are NIL or if *len* is negative then NIL is returned.

Example: '(LEFTSTR "Hello world!" 5)' results to "Hello".

See also RIGHTSTR, MIDSTR, WORD, LINE.

### 15.12.3 RIGHTSTR

RIGHTSTR extracts a sub string out of a string.

(RIGHTSTR *str len*)

Returns the right part of the given string with at most *len* characters. If *str* or *len* are NIL or if *len* is negative then NIL is returned.

Example: '(RIGHTSTR "Hello world!" 6)' results to "world!".

See also LEFTSTR, MIDSTR, WORD, LINE.

### 15.12.4 MIDSTR

MIDSTR extracts a sub string out of a string.

(MIDSTR *str pos len*)

Returns a part of the given string with at most *len* characters. The sub string starts at the *pos*-th position (starting with zero). If *len* is NIL then the full sub string starting at *pos* is returned. If *str* is NIL or if *len* is negative then NIL is returned. If *pos* is out of range, that is, negative or greater than the string length, NIL is returned.

Example: '(MIDSTR "Hello world!" 3 5)' results to "lo wo".

See also LEFTSTR, RIGHTSTR, WORD, LINE, SETMIDSTR, INSMIDSTR.

### 15.12.5 SETMIDSTR

SETMIDSTR replaces a sub string in a string.

(SETMIDSTR *str index set*)

Returns a copy of string *str* where the sub string starting at *index* is overwritten with string *set*. The length of the returned string is greater or equal to the length of *str*. If one of the arguments is NIL or if *index* is out of range then NIL is returned.

Example: '(SETMIDSTR "Hello world!" 6 "Melanie!")' results to "Hello Melanie!".

See also INSMIDSTR, REPLACESTR.

### 15.12.6 INSMIDSTR

INSMIDSTR is used to insert a sub string into a string.

(INSMIDSTR *str index insert*)

Returns a copy of string *str* where the string *insert* has been inserted at the given index. If one of the arguments is NIL or if *index* is out of range then NIL is returned.

Example: '(INSMIDSTR "Hello world!" 6 "MUIbase-")' results to "Hello MUIbase-world!".

See also SETMIDSTR, REPLACESTR.

### 15.12.7 INDEXSTR

INDEXSTR searches a string for the first occurrence of a sub string.

(INDEXSTR *str substr*)

Searches for the first occurrence of *substr* in *str*. String comparison is done case-sensitive. Returns the index (starting with 0) of the sub string in *str* or NIL if the sub string is not present. If one of the arguments is NIL then NIL is returned.

Example: `'(INDEXSTR "Hello world!" "world")'` returns 6.

See also INDEXSTR\*, RINDEXSTR, RINDEXSTR\*, INDEXBRK, INDEXBRK\*.

### 15.12.8 INDEXSTR\*

INDEXSTR\* has the same effect as INDEXSTR (see [Section 15.12.7 \[INDEXSTR\], page 96](#)) except that string comparison is done case-insensitive.

See also INDEXSTR, RINDEXSTR, RINDEXSTR\*, INDEXBRK, INDEXBRK\*.

### 15.12.9 INDEXBRK

INDEXBRK searches for the first occurrence of a character in a string.

`(INDEXBRK str brkstr)`

Searches for the first occurrence of a character from *brkstr* in *str*. String comparison is done case-sensitive. Returns the index (starting with 0) of the first character found in *str* or NIL if no character is found. If one of the arguments is NIL then NIL is returned.

Example: `'(INDEXBRK "Hello world!" "aeiou")'` returns 1.

See also INDEXBRK\*, RINDEXBRK, RINDEXBRK\*, INDEXSTR, INDEXSTR\*.

### 15.12.10 INDEXBRK\*

INDEXBRK\* has the same effect as INDEXBRK (see [Section 15.12.9 \[INDEXBRK\], page 97](#)) except that string comparison is done case-insensitive.

See also INDEXBRK, RINDEXBRK, RINDEXBRK\*, INDEXSTR, INDEXSTR\*.

### 15.12.11 RINDEXSTR

RINDEXSTR searches a string for the last occurrence of a sub string.

`(RINDEXSTR str substr)`

Searches for the last occurrence of *substr* in *str*. String comparison is done case-sensitive. Returns the index (starting with 0) of the sub string in *str* or NIL if the sub string is not present. If one of the arguments is NIL then NIL is returned.

Example: `'(RINDEXSTR "Do itashimashite." "shi")'` returns 11.

See also RINDEXSTR\*, INDEXSTR, INDEXSTR\*, RINDEXBRK, RINDEXBRK\*.

### 15.12.12 RINDEXSTR\*

RINDEXSTR\* has the same effect as RINDEXSTR (see [Section 15.12.11 \[RINDEXSTR\], page 97](#)) except that string comparison is done case-insensitive.

See also RINDEXSTR, INDEXSTR, INDEXSTR\*, RINDEXBRK, RINDEXBRK\*.

### 15.12.13 RINDEXBRK

RINDEXBRK searches for the last occurrence of a character in a string.

`(RINDEXBRK str brkstr)`

Searches for the last occurrence of a character from *brkstr* in *str*. String comparison is done case-sensitive. Returns the index (starting with 0) of the last character found in *str* or NIL if no character is found. If one of the arguments is NIL then NIL is returned.

Example: `'(RINDEXBRK "Konnichiwa" "chk")'` returns 6.

See also RINDEXBRK\*, INDEXBRK, INDEXBRK\*, RINDEXSTR, RINDEXSTR\*.

### 15.12.14 RINDEXBRK\*

RINDEXBRK\* has the same effect as RINDEXBRK (see [Section 15.12.13 \[RINDEXBRK\]](#), [page 97](#)) except that string comparison is done case-insensitive.

See also RINDEXBRK, INDEXBRK, INDEXBRK\*, RINDEXSTR, RINDEXSTR\*.

### 15.12.15 REPLACESTR

REPLACESTR replaces sub strings by other strings.

```
(REPLACESTR str [substr1 replacestr1 ...])
```

Replaces all occurrences of *substr1* in *str* by *replacestr1*. Continues replacing further sub strings in the new string using the next pair of search and replacement string until all arguments have been processed. Note that the number of arguments must be odd with arguments at even positions specifying the search strings each followed by the replacement string. Due to the fact that the result of a replacement is used in the next following replacement, multiple replacements can take place. This should be considered when using this function. A different order of the arguments might help resolving conflicts since replacements are done from left to right.

If any of the strings are NIL or any search string is empty then NIL is returned.

Example: ‘(REPLACESTR "black is white" "black" "white" "white" "black")’ results to "black is black".

See also REPLACESTR\*, SETMIDSTR, INSMIDSTR, REMCHARS.

### 15.12.16 REPLACESTR\*

REPLACESTR\* has the same effect as REPLACESTR (see [Section 15.12.15 \[REPLACESTR\]](#), [page 98](#)) except that string comparison is done case-insensitive for searching sub strings.

See also REPLACESTR, SETMIDSTR, INSMIDSTR, REMCHARS.

### 15.12.17 REMCHARS

REMCHARS removes characters from a string.

```
(REMCHARS str chars-to-remove)
```

Returns a copy of *str* where all characters of *chars-to-remove* are removed from. If *str* or *chars-to-remove* are NIL then NIL is returned.

Example: ‘(REMCHARS *your-string* " \t\n")’ removes all spaces, tabs and newline characters from *your-string*.

See also REPLACESTR, TRIMSTR.

### 15.12.18 TRIMSTR

TRIMSTR removes leading and trailing characters from a string.

```
(TRIMSTR str [front back])
```

Returns a copy of *str* where leading and trailing characters have been removed. If called with one argument only, spaces, form-feeds, newlines, carriage returns, and horizontal and vertical tabs are removed. When called with three arguments, *front* specifies the leading and *back* specifies the trailing characters to be removed. Note that TRIMSTR cannot be called with two arguments.

If any of *str*, *front*, or *back* is NIL then NIL is returned.

Example: (TRIMSTR " I wrecked Selma's bike. ") results to "I wrecked Selma's bike.", (TRIMSTR "007 " "0" " \f\n\r\t\v") results to "7".

See also REMCHARS.

### 15.12.19 WORD

WORD returns a word of a string.

(WORD *str num*)

Returns the *num*-th word (starting with zero) of the given string. Words in a string are non-empty sub strings separated by space-like characters (e.g. space, tab or newline characters).

If *str* or *num* are NIL, or if *num* is out of range, that is, less than zero or greater or equal to the number of words, then NIL is returned.

Example: '(WORD "Therefore, I lend Selma my bike." 3)' results to "Selma".

See also WORDS, LINE, LEFTSTR, RIGHTSTR, MIDSTR.

### 15.12.20 WORDS

WORDS counts the number of words in a string.

(WORDS *str*)

Returns the number of words of the given string or NIL if *str* is NIL. Words are sub strings separated by space-like characters (e.g. space, tab, or newline characters).

Example: '(WORDS "Actually, it wasn't really my bike.")' results to 6.

See also WORD, LINES, LEN.

### 15.12.21 STRTOLIST

STRTOLIST converts a string to a list of sub-strings.

(STRTOLIST *str [sep]*)

Creates a list of sub-strings by breaking the string *str* at the occurrences of the separation sequence *sep*. If *sep* is not specified then the tab character "\t" is used. If *sep* is the empty string "" then a list of all characters in the string is returned.

If *str* or *sep* are NIL then NIL is returned.

#### Examples

'(STRTOLIST "I\tlike\tJapan.")' results to ( "I" "like" "Japan." ).

'(STRTOLIST "Name|Street|City" "|")' results to ( "Name" "Street" "City" ).

'(STRTOLIST "abc" "")' results to ( "a" "b" "c" ).

See also MEMOTOLIST, LISTTOSTR.

### 15.12.22 LISTTOSTR

LISTTOSTR converts a list of items into a string.

(LISTTOSTR *list [sep]*)

Converts the given list of items into a string by concatenation of the string representations of each list element separated by the sequence *sep*. If *sep* is not specified then the tab character "\t" is used. If *list* or *sep* are NIL then NIL is returned.

## Examples

‘(LISTTOSTR (LIST "Peter is" 18 "years old"))’ results to: "Peter is\t18\tyears old".

‘(LISTTOSTR (LIST "Name" "Street" "City") "|")’ results to: "Name|Street|City".

See also LISTTOMEMO, CONCAT, CONCAT2, STRTOLIST.

### 15.12.23 CONCAT

CONCAT concatenates strings.

```
(CONCAT [str ...])
```

Returns the concatenation of the given list of strings where space characters have been inserted between the strings. If one of the strings is NIL, or the list is empty, then NIL is returned.

Example: ‘(CONCAT "I" "thought" "it" "was" "an" "abandoned" "bike.")’ results to "I thought it was an abandoned bike."

See also CONCAT2, +, LISTTOSTR, COPYSTR, SPRINTF.

### 15.12.24 CONCAT2

CONCAT2 concatenates strings.

```
(CONCAT2 insert [str ...])
```

Returns the concatenation of the given list of strings. The strings will be separated with the given *insert* string. If *insert* is NIL, one of the strings is NIL, or the list is empty, then NIL is returned.

Example: ‘(CONCAT2 "! " "But" "it" "wasn't!")’ results to "But! it! wasn't!".

See also CONCAT, +, LISTTOSTR, COPYSTR, SPRINTF.

### 15.12.25 COPYSTR

COPYSTR creates copies of a string.

```
(COPYSTR str num)
```

Returns a string consisting of *num* times the string *str*. If *str* is NIL, *num* is NIL or less than zero, NIL is returned.

Example: ‘(COPYSTR "+-" 5)’ results to "+-+-+--+".

See also CONCAT, CONCAT2, +, SPRINTF.

### 15.12.26 SHA1SUM

SHA1SUM computes the SHA1 hash of a string.

```
(SHA1SUM str)
```

Returns a string containing the SHA1 hash of the given string. If *str* is NIL then NIL is returned.

Example: ‘(SHA1SUM "flower, sun and beach")’ results to "47b6c496493c512b40e042337c128d85ecf15ba4".

See also ADMINPASSWORD, demo ‘Users.mb’.



### 15.12.27 UPPER

UPPER converts a string to upper case.

(UPPER *str*)

Returns a copy of the given string where all characters are converted to upper case. If *str* is NIL then NIL is returned.

Example: '(UPPER "Selma found a letter attached to my bike.")' results to "SELMA FOUND A LETTER ATTACHED TO MY BIKE."

See also LOWER.

### 15.12.28 LOWER

LOWER converts a string to lower case.

(LOWER *str*)

Returns a copy of the given string where all characters are converted to lower case. If *str* is NIL then NIL is returned.

Example: '(LOWER "The letter was from Silke.")' results to "the letter was from silke."

See also UPPER.

### 15.12.29 ASC

ASC converts a character to its internal integer representation.

(ASC *str*)

Returns the internal integer code of the first character of *str*. On Windows, Mac OS and Linux this is the unicode representation. On Amiga, it is the 8-bit integer code in the default character encoding. If *str* is empty, 0 is returned. If *str* is NIL, NIL is returned.

Example: (ASC "A") results to 65.

See also CHR, INT.

### 15.12.30 CHR

CHR converts an integer code to a character.

(CHR *int*)

Returns a string containing the character with integer code *int*. On Windows, Mac OS and Linux, *int* is interpreted as a unicode character. On Amiga, *int* is the 8-bit integer in the default character encoding. If *int* is 0, an empty string is returned. If *int* is NIL or not in the range of valid character values, NIL is returned.

Example: '(CHR 67)' results to "C".

See also ASC, STR.

### 15.12.31 LIKE

LIKE compares strings.

(LIKE *str1 str2*)

Returns TRUE if *str1* matches *str2*, NIL otherwise. The string *str2* may contain the joker characters '?' and '\*' where '?' matches exactly one character and '\*' matches a string of any length. String comparison is done case insensitive.

Example: ‘(LIKE "Silke has been in France for one year." "\*France\*")’ results to TRUE.

See also Comparison functions.

### 15.12.32 SPRINTF

SPRINTF formats a string with various data.

(SPRINTF *fmt* [*expr* ...])

SPRINTF takes a series of arguments, converts them to strings, and returns the formatted information as one string. The string *fmt* determines exactly what gets written to the return string and may contain two types of items: ordinary characters which are always copied verbatim and conversion specifiers which direct **SPRINTF** to take arguments from its argument list and format them. Conversion specifiers always begin with a ‘%’ character.

Conversion specifiers always take the following form:

%[*flags*][*width*][.*precision*]*type*

where

- The optional *flags* field controls output justification, sign character on numerical values, decimal points, and trailing blanks.
- The optional *width* field specifies the minimum number of characters to print (the field width), with padding done with blanks or zeros.
- The optional *precision* field specifies either the maximum number of characters to be printed for types string, Boolean, date and time, or the number of digits after the decimal point to be printed for values of type real.
- The *type* field specifies the actual type of the argument that **SPRINTF** will be converting, such as string, integer, real, etc.

Note that all of the above fields are optional except for *type*. The following tables list the valid options for these fields.

#### Flags field

-:	The result is left justified, with padding done on the right using blanks. By default when ‘-’ is not specified, the result is right justified with padding on the left with ‘0’'s or blanks.
+:	The result will always have a ‘-’ or ‘+’ character prepended to it if it is a numeric conversion.
0:	For numbers padding on the left is done using leading zeros instead of spaces.
space:	Positive numbers begin with a space instead of a ‘+’ character, but negative values still have a prepended ‘-’.

#### Width field

<i>n</i> :	A minimum of <i>n</i> characters are output. If the conversion has less than <i>n</i> characters, the field is padded with blanks or leading zeros.
*:	The width specifier is supplied in the argument list as an integer or real value, before the actual conversion argument. This value is limited to the range of 0 to 999.

## Precision field

- .n:** For string, Boolean, date and time values, *n* is the maximum number of characters written from the converted item. For conversions of real values, *n* specifies the number of digits after the decimal point (conversions ‘f’ and ‘e’) or the number of significant digits (conversion ‘g’). For integer conversions this field is ignored.
- .\*:** The precision is supplied in the argument list as an integer or real value, before the actual conversion argument. This value is limited to the range of 0 to 999.

## Type field

- b:** Converts a Boolean parameter to "TRUE" or "NIL".
- i:** Converts an integer value to a signed decimal notation.
- o:** Converts an integer value to an unsigned octal notation.
- x:** Converts an integer value to an unsigned hexadecimal notation using lowercase letters ‘abcdef’.
- X:** Converts an integer value to an unsigned hexadecimal notation using uppercase letters ‘ABCDEF’.
- e:** Converts a real number using the format [-]d.ddde+dd. Exactly one digit is before the decimal point, followed by an ‘e’, followed by an exponent. The number of digits after the decimal point is determined by the precision field, or is 2 if precision is not specified. The decimal point will not appear if precision is 0.
- f:** Converts a real value using the format [-]ddd.ddd. The number of digits after the decimal point is determined by the precision field, or is 2 if precision is not specified. The decimal point will not appear if precision is 0.
- g:** Converts a real value using style ‘e’ or ‘f’ depending on the number of digits for representing the value. If the precision is not specified then 15 significant digits are used. Trailing zeros other than a single zero after the decimal point are removed.
- s:** Writes a string value until either the end of string is reached or the number of characters written equals the precision field.
- d:** Converts a date value.
- t:** Converts a time value.
- %:** The character ‘%’ is written, and no argument is converted.

SPRINTF returns the formatted string or NIL in case *fmt* is NIL.

## Examples

Call	Result
(SPRINTF "Hello")	"Hello"

```

(SPRINTF "%s" "Hello")           "Hello"
(SPRINTF "%10s" "Hello")         "      Hello"
(SPRINTF "%-10.10s" "Hello")     "Hello      "
(SPRINTF "%010.3s" "Hello")      "      Hel"
(SPRINTF "%-5.3b" TRUE)          "TRU  "
(SPRINTF "%i" 3)                  "3"
(SPRINTF "%03i" 3)                "003"
(SPRINTF "%0- 5.3i" 3)            " 3  "
(SPRINTF "%f" 12)                 "12.00"
(SPRINTF "%10e" 12.0)             "  1.20e+01"
(SPRINTF "%+-10.4f" 12.0)         "+12.0000 "
(SPRINTF "%10.5t" 12:30:00)       "      12:30"
(SPRINTF "%d" 28.11.1968)         "28.11.1968"
(SPRINTF "He%s %5.5s!"
  "llo"
  "world champion ship")          "Hello world!"

```

See also PRINTF, FPRINTF, STR, +, CONCAT, CONCAT2, COPYSTR.

## 15.13 Memo Functions

This section deals with functions useful for memos.

### 15.13.1 LINE

LINE extracts a line in a memo.

```
(LINE memo num)
```

Returns the *num*-th line (starting with zero) of the given memo. The line string will not have a terminating newline character. If *memo* or *num* are NIL or if *num* is out of range, that is, less than zero or greater or equal to the number of lines, then NIL is returned.

See also LINES, WORD.

### 15.13.2 LINES

LINES returns the number of lines in a memo.

```
(LINES memo)
```

Returns the number of lines of the given memo or NIL if *memo* is NIL.

See also LINE, WORDS, LEN.

### 15.13.3 MEMOTOLIST

MEMOTOLIST converts a memo into a list of strings.

```
(MEMOTOLIST memo [expandstr])
```

Converts the given memo to a list. If *memo* is NIL then NIL is returned, otherwise a list is generated where each element contains one line of the memo.

If *expandstr* is given and is not NIL then resulting list of strings is further processed by applying STRTOLIST to each list element. This results into a list of lists of strings.

## Examples

‘(MEMOTOLIST "My insurance\npays for\nthe wrecked bike.")’ results to ( "My insurance" "pays for" "the wrecked bike." ).

‘(MEMOTOLIST "Here is\ta multi-columned\nexample." TRUE)’ results to ( ( "Here is" "a multi-columned" ) ( "example" ) ).

See also STRTOLIST, LISTTOMEMO.

### 15.13.4 LISTTOMEMO

LISTTOMEMO converts a list into a memo.

```
(LISTTOMEMO list)
```

Converts the given list into a memo. If *list* is NIL then NIL is returned, otherwise a memo is generated where each line consists of the string representation of the corresponding list element. If a list element is a sub-list then LISTTOSTR (see [Section 15.12.22 \[LISTTOSTR\]](#), [page 99](#)) is applied on it before integrating the resulting string into the memo.

## Examples

‘(LISTTOMEMO (LIST "Silke" "lends me" "'my' bike" "till" 01.09.1998))’ results to: "Silke\nlends me\n'my' bike\ntill\n01.09.1998".

‘(LISTTOMEMO (LIST (LIST "Name" "Birthday") (LIST "Steffen" 28.11.1968))’ results to: "Name\tBirthday\nSteffen\t28.11.1968".

See also LISTTOSTR, MEMOTOLIST.

### 15.13.5 FILLMEMO

FILLMEMO fills in a memo with the results of expressions.

```
(FILLMEMO memo)
```

Creates a copy of the given memo where all substrings of the form ‘\$(*expr*)’ are replaced by their results after evaluation.

Example: ‘(FILLMEMO "(+ 1 1) is \$(+ 1 1).")’ results to "(+ 1 1) is 2."

Please use only small expressions in the memo as debugging and tracking down errors is not easy here.

See also FORMATMEMO, INDENTMEMO.

### 15.13.6 FORMATMEMO

FORMATMEMO formats a memo.

```
(FORMATMEMO memo width [fill [singlelinesec]])
```

Formats *memo* to a memo with lines not longer than *width* characters. If *fill* is given and non-NIL then spaces are used to pad the lines to exactly *width* characters. The memo is processed section-wise. A section starts at the first non-white-space character. If *singlelinesec* is specified and non-NIL then all characters until the end of this line are taken as the section. Otherwise, all characters on this and the following lines are counted as the section until a line starting with a white-space character is reached. The whole section is formatted word-wise, that is, as many words are put on one line as there are space for. Remaining words are then put on the next line and so on.

See also FILLMEMO, INDENTMEMO.

### 15.13.7 INDENTMEMO

INDENTMEMO indents a memo by putting spaces to the left.

`(INDENTMEMO memo indent)`

Returns a copy of the given memo where each line is indented by *indent* space characters. If *memo* or *indent* is NIL then NIL is returned. If *indent* is negative, a value of 0 is used.

See also FILLMEMO, FORMATMEMO.

## 15.14 Date and Time Functions

This section deals with functions useful for date and time values.

### 15.14.1 DAY

DAY extracts the day field of a date.

`(DAY date)`

Returns an integer representing the day of the given date value. If *date* is NIL then NIL is returned.

See also MONTH, YEAR, DATEDMY.

### 15.14.2 MONTH

MONTH extracts the month field of a date.

`(MONTH date)`

Returns an integer representing the month of the given date value. If *date* is NIL then NIL is returned.

See also DAY, YEAR, DATEDMY, MONTHDAYS.

### 15.14.3 YEAR

YEAR extracts the year field of a date.

`(YEAR date)`

Returns an integer representing the year of the given date value. If *date* is NIL then NIL is returned.

See also DAY, MONTH, DATEDMY, YEARDAYS.

### 15.14.4 DATEDMY

DATEDMY creates a date value from day, month, and year.

`(DATEDMY day month year)`

Creates a date value from the given day, month, and year. If any of *day*, *month*, or *year* is NIL or out of the valid range, or if the resulting date is invalid, then NIL is returned.

Example: ‘(DATEDMY 28 11 1968)’ results to the 28th of November, 1968.

See also DATE, TODAY, DAY, MONTH, YEAR.

### 15.14.5 MONTHDAYS

MONTHDAYS gets the number of days of a month.

(MONTHDAYS *month year*)

Returns the number of days of the given month and year as an integer. If *month* is NIL or out of the valid range (less than 1 or greater than 12) then NIL is returned. If *year* is NIL then a non-leap year is considered for computing the number of days. If *year* is invalid (less than 0 or greater than 9999) then NIL is returned.

Examples: '(MONTHDAYS 2 2004)' results to 29, '(MONTHDAYS 2 NIL)' results to 28.

See also YEARDAYS, MONTH.

### 15.14.6 YEARDAYS

YEARDAYS gets the number of days of a year.

(YEARDAYS *year*)

Returns the number of days of the given year as an integer. If *year* is NIL or out of the valid range (less than 0 or greater than 9999) then NIL is returned.

Examples: '(YEARDAYS 2004)' results to 366, '(YEARDAYS 2005)' results to 365.

See also MONTHDAYS, YEAR.

### 15.14.7 ADDMONTH

ADDMONTH adds a number of months to a date.

(ADDMONTH *date months*)

Returns a date value where the given number of months has been added to the given date value. Negative values for *months* subtract months. If *date* or *months* is NIL, or the resulting date is invalid, then NIL is returned.

ADDMONTH handles over- and underflow of the month field by adjusting the year field accordingly. In case the day field exceeds the maximum number of days of the resulting month, it is decremented to the maximum allowed day.

Examples: '(ADDMONTH 30.01.2004 1)' results to 29.02.2004, '(ADDMONTH 30.01.2004 -1)' results to 30.12.2003.

See also ADDYEAR, +.

### 15.14.8 ADDYEAR

ADDEAR adds a number of years to a date.

(ADDEAR *date years*)

Returns a date value where the given number of years has been added to the given date value. Negative values for *years* subtract years. If *date* or *years* is NIL, or the resulting date is invalid, then NIL is returned.

ADDEAR decrements the day field by 1 in case *date* represents February 29th and the resulting year is not a leap year.

Examples: '(ADDEAR 29.02.2004 1)' results to 28.02.2005, '(ADDEAR 04.02.2004 -1962)' results to 04.02.0042.

See also ADDMONTH, +.

### 15.14.9 TODAY

TODAY returns the current date.

(TODAY)

Returns the current date as a date value.

See also NOW, DATEDMY.

### 15.14.10 NOW

NOW returns the current time.

(NOW)

Returns the current time as a time value.

See also TODAY.

## 15.15 List Functions

This section lists functions for processing lists.

### 15.15.1 CONS

CONS builds a pair of expressions.

(CONS *elem list*)

Constructs a new list. The first element of the new list is *elem*, the rest are the elements of *list* (which should be a list or NIL). The list *list* is not copied, only a pointer is used to reference it!

Example: ‘(CONS 1 (CONS 2 NIL))’ results to ( 1 2 ).

The elements of a list can be of any type, e.g. it’s also possible to have a list of lists (e.g. see [Section 15.20.12 \[SELECT\], page 128](#)). The CONS constructor can also be used to build pairs of elements, e.g. ‘(CONS 1 2)’ is the pair with the two integers 1 and 2.

See also LIST, FIRST, REST.

### 15.15.2 LIST

LIST generates a list out of its arguments.

(LIST [*elem* ...])

takes the arguments *elem* ... and generates a list of it. This is equivalent to calling (CONS *elem* (CONS ... NIL)). Note that NIL alone stands for an empty list.

See also CONS, LENGTH.

### 15.15.3 LENGTH

LENGTH determines the length of a list.

(LENGTH *list*)

returns the length of the given list.

Example: ‘(LENGTH (LIST "a" 2 42 3))’ results to 4.

See also LIST.



#### 15.15.4 FIRST

FIRST extracts the first element in a list.

`(FIRST list)`

returns the first element of the given list. If *list* is empty (NIL) then NIL is returned.

See also REST, LAST, NTH, CONS.

#### 15.15.5 REST

REST returns the sub-list after the first element of a list.

`(REST list)`

returns the rest of the given list (the list without the first element). If *list* is empty (NIL) then NIL is returned.

Example: ‘`(REST (LIST 1 2 3))`’ results to `( 2 3 )`.

See also FIRST, CONS.

#### 15.15.6 LAST

LAST extracts the last element in a list.

`(LAST list)`

Returns the last element of the given list or NIL if *list* is NIL.

See also FIRST, NTH.

#### 15.15.7 NTH

NTH extracts the n-th element of a list.

`(NTH n list)`

Returns the *n*-th element of the given list (starting with 0) or NIL if the element doesn’t exist.

See also FIRST, LAST.

#### 15.15.8 APPEND

APPEND concatenates lists.

`(APPEND [list ...])`

returns the concatenation of *list* ....

Example: ‘`(APPEND (list 1 2) (list 3 4) (list 5))`’ results to `( 1 2 3 4 5 )`.

See also LIST.

#### 15.15.9 REVERSE

REVERSE reverses a list.

`(REVERSE list)`

returns the reverse list.

Example: ‘`(REVERSE (list 1 2 3))`’ results to `( 3 2 1 )`.

### 15.15.10 MAPFIRST

MAPFIRST applies a function to all list elements.

```
(MAPFIRST func list [...])
```

Builds a list whose elements are the result of the specified function called with the arguments of the given list elements one by one. The length of the returned list is as long as the length of the longest specified list. If one of the specified lists is too short then the list is padded with NIL elements.

#### Examples

Expression	Value
(MAPFIRST 1+ (LIST 1 2 3))	( 2 3 4 )
(MAPFIRST + (LIST 1 2 3) (LIST 2 3))	( 3 5 NIL )

### 15.15.11 SORTLIST

SORTLIST sorts the elements of a list.

```
(SORTLIST func list)
```

Returns a copy of the specified list that has been sorted using the function *func* for ordering. The order function must take two arguments one for each element and return an integer value less than zero if the first element is smaller than the second one, a value greater than zero if the first element is greater than the second one, and a value of zero if the two elements are equal.

Example for a string comparing function usable for sorting:

```
(DEFUN cmp_str (x y)
  (COND
    ((< x y) -1)
    ((> x y) 1)
    (TRUE 0)
  )
)
```

Now you can sort a list by calling:

```
(SORTLIST cmp_str (LIST "hi" "fine" "great" "ok"))
```

which results to ( "fine" "great" "hi" "ok" ).

See also SORTLISTGT, MAPFIRST.

### 15.15.12 SORTLISTGT

SORTLISTGT sorts the elements of a list.

```
(SORTLISTGT gtfunc list)
```

Like SORTLIST but here you specify an order function that returns a value not equal to NIL if the first element is greater than the second one, and NIL otherwise.

Example: ‘(SORTLISTGT > (LIST "hi" "fine" "great" "ok"))’ result to ( "fine" "great" "hi" "ok" ).

See also SORTLIST, MAPFIRST.

## 15.16 Input Requesting Functions

For requesting information from the user, the following functions can be used.

### 15.16.1 ASKFILE

ASKFILE prompts the user for entering a filename.

```
(ASKFILE title oktext default savemode)
```

Pops up a file-requester for entering a filename. The window title can be set in *title*, the text of the ‘Ok’ button in *oktext*, and the initial filename in *default*. You may specify NIL for one of them to use default values. The last argument *savemode* (Boolean) allows setting the file-requester to a save mode. This mode should be used when asking for a filename to write something to.

ASKFILE returns the entered filename as string or NIL in case the user canceled the requester.

See also ASKDIR, ASKSTR.

### 15.16.2 ASKDIR

ASKDIR prompts the user for entering a directory name.

```
(ASKDIR title oktext default savemode)
```

Pops up a file-requester for entering a directory name. The arguments are used in the same way as in ASKFILE (see [Section 15.16.1 \[ASKFILE\]](#), page 111).

ASKDIR returns the entered directory name as string or NIL in case the user canceled the requester.

See also ASKFILE, ASKSTR.

### 15.16.3 ASKSTR

ASKSTR prompts the user for entering a string.

```
(ASKSTR title oktext default maxlen [secret])
```

Pops up a requester asking for a string to enter. The window title, the text of the ‘Ok’ button, and the initial value can be set in *title*, *oktext*, and *default* respectively (strings or NIL for default values), *maxlen* determines the maximum characters the user can enter. If *secret* is given and is not NIL then the entered string is made invisible by displaying a bullet sign for each string character.

ASKSTR returns the entered string or NIL in case the user canceled.

See also ASKFILE, ASKDIR, ASKCHOICESTR, ASKINT.

### 15.16.4 ASKINT

ASKINT prompts the user for entering an integer value.

```
(ASKINT title oktext default min max)
```

Pops up a requester asking for an integer to enter. The window title and the text of the ‘Ok’ button can be specified in *title* and *oktext* (strings or NIL for default values). In *default* you pass the initial integer value or NIL to start with an empty editing field. In *min*

and *max* you can set the integer range. Entered values outside this range are not accepted by the requester. Use *NIL* for default min and max values.

ASKINT returns the entered integer or *NIL* if the user canceled the requester.

See also ASKSTR.

### 15.16.5 ASKCHOICE

ASKCHOICE prompts the user to select one item out of many items.

```
(ASKCHOICE title oktext choices default [titles])
```

Pops up a requester allowing the user to choose one item out of a list of items. You can set the window title and the text of the ‘Ok’ button in *title* and *oktext* (strings or *NIL* for default values). In *choices* you specify a list of choices. You can use a multi-column format by providing each choice item as a list of sub-items. The initial choice value can be set in *default* and is the index in the list of choices (starting with index 0 for the first item). Use *NIL* for no initial choice. If the optional argument *titles* is specified and is non-*NIL* then a list header containing *titles* is shown. For using a multi-column format, specify *titles* as a list of column titles.

Both, *choices* and *titles*, can also be given as a memo and a string (instead of lists). If so, they are converted to lists automatically by calling (MEMOTOLIST *choices* TRUE) (see [Section 15.13.3 \[MEMOTOLIST\], page 104](#)) and (STRTOLIST *titles*) (see [Section 15.12.21 \[STRTOLIST\], page 99](#)) respectively.

ASKCHOICE returns the index of the chosen item or *NIL* if the user canceled the requester.

#### Example

```
(LET ((items (LIST "First Entry" 2 3.14 "Last entry"))) index)
  (SETQ index (ASKCHOICE "Choose one item" "Ok" items NIL))
  (IF index
    (PRINTF "User chose item num %i with contents <%s>\n"
      index (STR (NTH index items)))
    )
  )
```

Consider the case where you would like to ask the user to select a certain record in a table. The table shall be called ‘Article’ with attributes ‘Name’, ‘Number’, and ‘Prize’. The following code fragment shows how to use ASKCHOICE for asking to select a record with Prize larger than 10 and ordered by Name:

```
(LET ((query (SELECT Article, Name, Number, Prize from Article
  WHERE (> Prize 10) ORDER BY Name))
  (recs (MAPFIRST FIRST (REST query))) ; record pointers
  (items (MAPFIRST REST (REST query))) ; choices
  (titles (REST (FIRST query))) ; titles
  (index (ASKCHOICE "Choose" "Ok" items NIL titles))
  (rec (NTH index recs)))
  ; now rec holds the selected record (or NIL on cancel)
)
```

See also ASKCHOICESTR, ASKOPTIONS.

### 15.16.6 ASKCHOICESTR

ASKCHOICESTR prompts the user for entering a string value offering several pre-defined ones.

```
(ASKCHOICESTR title oktext strings default [titles])
```

Pops up a requester allowing the user to choose one string out of many strings or to enter any other string in a separate string field. You can set the window title and the text of the ‘Ok’ button in *title* and *oktext* (strings or NIL for default values). In *strings* you specify a list of choices. You can use a multi-column format by providing each choice item as a list of sub-items. The initial value of the string field can be set in *default* (string or NIL for an empty string field). If the optional argument *titles* is specified and is non-NIL then a list header containing *titles* is shown. For using a multi-column format, specify *titles* as a list of column titles.

Both, *strings* and *titles*, can also be given as a memo and a string (instead of lists). If so, they are converted to lists automatically by calling (MEMOTOLIST *strings* TRUE) (see [Section 15.13.3 \[MEMOTOLIST\], page 104](#)) and (STRTOLIST *titles*) (see [Section 15.12.21 \[STRTOLIST\], page 99](#)) respectively.

ASKCHOICESTR returns the chosen string or NIL if the user canceled the requester.

#### Example

```
(LET ((strings (LIST "Claudia" "Mats" "Ralphie"))) likebest)
  (SETQ likebest
    (ASKCHOICESTR "Who do you like the best?" "Ok" strings
      "My collies!"
    )
  )
  (IF likebest (PRINTF "User chose <%s>\n" likebest))
)
```

See also ASKCHOICE, ASKOPTIONS.

### 15.16.7 ASKOPTIONS

ASKOPTIONS prompts the user for selecting several items out of a list of items.

```
(ASKOPTIONS title oktext options selected [titles])
```

Pops up a requester allowing the user to select several options out of a list of options. You can set the window title and the text of the ‘Ok’ button in *title* and *oktext* (strings or NIL for default values). In *options* you specify a list of options. You can use a multi-column format by providing each option item as a list of sub-items. The initial selection state can be set in *selected* as a list of integers each specifying an index whose corresponding item in *options* should initially be selected. Use NIL for all items being unselected. If the optional argument *titles* is specified and is non-NIL then a list header containing *titles* is shown. For using a multi-column format, specify *titles* as a list of column titles.

Both, *options* and *titles*, can also be given as a memo and a string (instead of lists). If so, they are converted to lists automatically by calling (MEMOTOLIST *options* TRUE) (see [Section 15.13.3 \[MEMOTOLIST\], page 104](#)) and (STRTOLIST *titles*) (see [Section 15.12.21 \[STRTOLIST\], page 99](#)) respectively.

ASKOPTIONS returns a list of integers each specifying the index of a selected item, or NIL in case the user canceled or did not choose any item.

## Example

```
(LET ((options (LIST "Salva Mea" "Insomnia" "Don't leave" "7 days & 1 week"))
      (selected (LIST 0 1 3)))
  )
  (SETQ selected (ASKOPTIONS "Select music titles" "Ok" options selected))
  (IF selected
    (
      (PRINTF "User has selected the following items:\n")
      (DOLIST (i selected)
        (PRINTF "\tnum: %i contents: <%s>\n" i (STR (NTH i options)))
      )
    )
  )
)
```

### 15.16.8 ASKBUTTON

ASKBUTTON prompts the user for pressing a button.

```
(ASKBUTTON title text buttons canceltext)
```

Pops up a requester with the specified window title (string or NIL for a default title) and specified description text (string or NIL for no text). The function waits until the user presses one of the buttons specified in *buttons* (list of strings) or the ‘Cancel’ button. The text of the cancel button can be set in *canceltext*. If you specify NIL here then a default text based on the number of buttons you specified is used.

ASKBUTTON returns the number of the pressed button (starting with 0 for the first (left-most) button) or NIL if the user pressed the ‘Cancel’ button.

## Examples

```
(LET ((buttons (LIST "At home" "In bed" "In front of my computer"))) index)
  (SETQ index (ASKBUTTON "Please answer:"
    "Where do you want to be tomorrow?" buttons "Don't know")
  )
  (IF index
    (PRINTF "User chose: <%s>\n" (NTH index buttons))
  )
)
```

```
(ASKBUTTON "Info" "MUIbase is great!" NIL NIL)
```

See also ASKCHOICE.

### 15.16.9 ASKMULTI

ASKMULTI prompts the user to enter various kinds of information.

```
(ASKMULTI title oktext itemlist)
```

ASKMULTI is a multi-purpose requester. It opens a window with the specified title, a set of GUI objects for editing data, and two buttons (‘Ok’ and ‘Cancel’) for terminating the requester. The text of the ‘Ok’ button can be set in *oktext* (string or NIL for default text).

The set of GUI objects are specified in *itemlist* which is a list of items where each item has one of the following forms:

```
(LIST title "String" initial [help [secret]]) for editing one line of text,
(LIST title "Memo" initial [help])           for editing multi-line text,
(LIST title "Integer" initial [help])         for editing an integer,
(LIST title "Real" initial [help])            for editing a real,
(LIST title "Date" initial [help])            for editing a date,
(LIST title "Time" initial [help])            for editing a time,
(LIST title "Bool" initial [help])            for a Boolean field,
(LIST title "Choice" initial
  (LIST choice ...) [help]
)                                              for a choice field.
(LIST title "ChoiceList" initial
  (LIST choice ...) [help]
)                                              for selecting one item of a list.
(LIST title "Options" initial
  (LIST option ...) [help]
)                                              for selecting several items of a list.
non-list-expr                                for static text
```

The title (string or NIL for no title) will be displayed to the left of the GUI object. If the initial value is NIL then a default value is used (e.g. an empty text field). For choice fields the initial value must be the index (starting with 0) for the initial active entry, for choice list fields the initial value may be NIL (no item is activated), and for options fields the initial value must be a list of integers representing the indices (starting with 0) of the items that are initially selected. The optional help field (string) can be used for giving more information to the user about the usage of the field. For string fields an additional parameter 'secret' can be specified. If non-NIL then the contents of the string field are made invisible by displaying a bullet sign for each string character.

ASKMULTI returns a list of result values which the user has edited and acknowledged by pressing the 'Ok' button. Each result value of a field has the same format as the one for the initial value, e.g. for a choice list field the result value is the index of the selected item (or NIL if no item has been selected), or for an options field the result value is a list of integers representing the indices of the selected items. For static text a value of NIL is returned.

E.g. if you have specified a date field, a static text field, a choice field, an options field, and a string field with initial value "world", and the user has entered 11.11.1999, selected the choice entry with index number 2, selected the 3rd and 4th item in the options field, and left the string field untouched then the function returns the list ( 11.11.1999 NIL 2 ( 3 4 ) "world" ).

If the user cancels the requester, NIL is returned.

## Example

```
(ASKMULTI "Please edit:" NIL (LIST
  (LIST "N_ame" "String" "")
  (LIST "_Birthday" "Date" NIL)
```

```

    (LIST "_Sex" "Choice" 0 (LIST "male" "female"))
    (LIST "_Has car?" "Bool" NIL)
    (LIST "_Likes" "Options" (LIST 0 2)
      (LIST "Beer" "Wine" "Whisky" "Wodka" "Schnaps"))
  ))
)

```

Please see also the project ‘AskDemo.mb’ for further examples.

## 15.17 I/O Functions

This sections lists functions and variables for input and output (e.g. printing) of data.

### 15.17.1 FOPEN

FOPEN opens a file for reading/writing.

```
(FOPEN filename mode [encoding])
```

Opens the file specified by *filename* (string). The *mode* parameter (string) controls the access mode. Use “w” to open a file for writing, “a” to append to a file, and “r” for reading from a file. You may also use other flags (or combination of flags) like “r+” for reading and writing. There is no check done that tests if you have specified a valid flag string. However if the file can’t be opened, NIL is returned.

The optional parameter *encoding* controls the encoding of the file and is one of the following strings:

“none”: No interpretation of characters is performed. Use this for binary files.

“UTF-8”: Text is encoded in UTF-8. Reading and writing converts from/to UTF-8.

“locale”:

Text is encoded in your system locale. On Windows this is the system’s code page. On Mac OS and Linux it is the setting of the LANG and LC\_\* environment variables, see `man locale`. On Amiga it is the default 8 bit encoding.

“8-bit”: Text is encoded in the default 8-bit codeset. On Windows, Mac OS and Linux this is the ISO-8859-1 (latin 1) encoding. On Amiga it is the system’s default 8 bit encoding (same as ‘locale’).

“auto”: The encoding is auto-detected. If the file is readable then the encoding is determined as follows: If all contents conform to UTF-8 then “UTF-8” is assumed. Else if the system’s locale is not UTF-8 then “locale” is assumed. Otherwise “8-bit” is assumed. When writing and the encoding has not been determined yet then first the system’s locale encoding is tried. If there were no conversion errors then “locale”, otherwise “UTF-8” is used.

If no *encoding* parameter is given then “auto” is used.

FOPEN returns a file handle on success. On failure NIL is returned. If *filename*, *mode* or *encoding* are NIL then NIL is returned.



## Examples

‘(FOPEN "index.html" "w" "utf-8")’ opens and returns a file handle for writing to the file ‘index.html’ and encoding it in UTF-8.

‘(FOPEN "output.txt" "a+")’ opens file ‘output.txt’ for appending it using the same text encoding as is already present in the file. Note that if you only specify “a” as mode then MUIbase might not be able to read the file and decide the existing encoding. In this case the encoding is decided when writing to the file (and might be different from the existing one).

See also FCLOSE, stdout, FFLUSH.

### 15.17.2 FCLOSE

FCLOSE closes a file.

(FCLOSE *file*)

Closes the given file. Returns 0 on success, NIL if an error has occurred. If *file* is NIL then 0 is returned (no error). After closing a file accessing the file handle is an illegal operation and results in aborting program execution with an error message.

See also FOPEN, FFLUSH.

### 15.17.3 stdout

The global variable `stdout` holds the file handle to MUIbase’s standard output file. The output filename can be set in menu item ‘Program – Output file’ (see [Section 7.2.11 \[Program output file\]](#), page 36).

The output file is opened on the first access of this variable (e.g. when calling ‘(FPRINTF stdout ...)’, or when calling ‘(PRINTF ...)’). The file is not pre-opened on program execution. This avoids opening the file when no output is generated, e.g. when you simply want to do some calculations and change some record contents.

When opening the output file the mode parameter is either “w” or “a+” depending on the ‘Append’ setting in menu item ‘Program – Output file’. The encoding is set to “auto”.

If MUIbase can’t open the program output file then execution is aborted and an error message is generated.

See also FOPEN, PRINTF.

### 15.17.4 PRINT

PRINT converts an expression to a string and prints it.

(PRINT *elem*)

Converts the value of *elem* to a readable string and prints it to `stdout`. This function mainly exists for debug purposes.

See also PRINTF, stdout.

### 15.17.5 PRINTF

PRINTF prints a formatted string.

`(PRINTF format [expr ...])`

Formats a string using the given format string and arguments and prints it to `stdout`. Formatting is done like in `SPRINTF` (see [Section 15.12.32 \[SPRINTF\]](#), page 102).

`PRINTF` returns the number of output characters or `NIL` on failure. If *format* is `NIL` then `NIL` is returned.

Example: `'(PRINTF "%i days and %i week" 7 1)'` prints the string "7 days and 1 week" to `stdout` and returns 17.

See also `PRINT`, `FPRINTF`, `stdout`.

### 15.17.6 FPRINTF

`FPRINTF` prints a formatted string to a file.

`(FPRINTF file format [expr ...])`

Formats a string using the given format string and arguments and prints it to the specified file. Formatting is done like in `SPRINTF` (see [Section 15.12.32 \[SPRINTF\]](#), page 102).

`FPRINTF` returns the number of output characters or `NIL` on failure. If *file* is `NIL` then `FPRINTF` still returns the number of potentially written characters but no output is actually written. If *format* is `NIL` then `NIL` is returned.

See also `PRINTF`, `FOPEN`.

### 15.17.7 FERROR

`FERROR` checks if an file I/O error has occurred.

`(FERROR file)`

returns `TRUE` if an error for the given file has occurred, `NIL` otherwise. If *file* is `NIL`, `NIL` is returned.

See also `FEOF`, `FOPEN`, `FCLOSE`.

### 15.17.8 FEOF

`FEOF` checks for an end of file status.

`(FEOF file)`

Tests the end-of-file indicator for the given file and returns `TRUE` if it is set. Otherwise `NIL` is returned. If *file* is `NIL`, `NIL` is returned.

See also `FERROR`, `FTELL`, `FOPEN`, `FCLOSE`.

### 15.17.9 FSEEK

`FSEEK` sets the read/write position in a file.

`(FSEEK file offset whence)`

Sets the read/write position for the given file. The new position, measured in bytes, is obtained by adding *offset* bytes to the position specified by *whence*. If *whence* is set to `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`, the *offset* is relative to the start of the file, the current position, or end-of-file, respectively.

On success, `FSEEK` returns 0. Otherwise `NIL` is returned and the file position stays unchanged. If *file*, *offset*, or *whence* is `NIL`, or if *whence* is not one of the constant values `SEEK_SET`, `SEEK_CUR`, or `SEEK_END` then `NIL` is returned.

Note that after a read operation calling FSEEK with *whence* as SEEK\_CUR is only supported for encoding “none”.

See also FTELL, FOPEN, Pre-defined constants.

### 15.17.10 FTELL

FTELL returns the read/write position of a file.

(FTELL *file*)

Determines the current read/write position of the given file relative to the beginning of the file and returns it as an integer. If an error occurs or if ‘file’ is NIL, then NIL is returned.

Note that after a read operation calling FTELL is only supported for encoding “none”.

See also FSEEK, FOPEN, FEOF.

### 15.17.11 FGETCHAR

FGETCHAR reads a character from a file.

(FGETCHAR *file*)

Returns the next character from the given file as a string or NIL if *file* is NIL, end of file has been reached, or an error has happened. If the next character is a null character, an empty string is returned.

See also FGETCHARS, FGETSTR, FPUTCHAR.

### 15.17.12 FGETCHARS

FGETCHARS reads characters from a file.

(FGETCHARS *num file*)

returns a string containing the next *num* characters from the given file. If end of file has been reached before reading *num* characters or if a null character has been read then only these characters are returned. If *num* or *file* are NIL, *num* is negative, end of file has been reached before reading the first character, or a read error has happened then NIL is returned.

See also FGETCHAR, FGETSTR.

### 15.17.13 FGETSTR

FGETSTR reads a string from a file.

(FGETSTR *file*)

returns the next line of the given file as a string or NIL if *file* is NIL, end of file has been reached, or an error happened. The end of a line is detected if either a newline character or a null character is read, or if end of file is detected. In either case the string does not contain any newline characters.

See also FGETCHAR, FGETCHARS, FGETMEMO, FPUTSTR.

#### 15.17.14 FGETMEMO

FGETMEMO reads a memo from a file.

(FGETMEMO *file*)

returns a memo that contains the contents of the given file up to the next null character or up to the end of file. If *file* is NIL, end of file has been reached before reading any characters, or an error occurred then NIL is returned.

See also FGETSTR, FPUTMEMO.

#### 15.17.15 FPUTCHAR

FPUTCHAR writes a character to a file.

(FPUTCHAR *str file*)

Writes the first character of *str* to the given file. If *str* is empty, a null character is written, if *str* or *file* are NIL, nothing happens. Returns *str* or NIL in case an output error occurred.

See also FPUTSTR, FGETCHAR.

#### 15.17.16 FPUTSTR

FPUTSTR writes a string to a file.

(FPUTSTR *str file*)

Prints *str* together with a newline character to the given file. If *str* or *file* are NIL, nothing happens. Returns *str* or NIL in case an output error occurred.

See also FPUTCHAR, FPUTMEMO, FGETSTR.

#### 15.17.17 FPUTMEMO

FPUTMEMO writes a memo to a file.

(FPUTMEMO *memo file*)

Prints *memo* to the given file. If *memo* or *file* are NIL, nothing happens. Returns *memo* or NIL in case an output error occurred.

See also FPUTSTR, FGETMEMO.

#### 15.17.18 FFLUSH

FFLUSH flushes pending data to a file.

(FFLUSH *file*)

Flushes all pending output for the given file. Returns 0 on success, NIL if an error occurred. If *file* is NIL then 0 is returned (no error).

See also FOPEN, FCLOSE.

### 15.18 Record Functions

This section lists functions that deal with records.

### 15.18.1 NEW

NEW allocates a new record for a table.

`(NEW table init)`

Allocates a new record in the given table. The parameter *init* specifies the record which should be used for initializing the new record. A value of NIL stands for the initial record.

NEW returns a record pointer to the new record.

The NEW function has the side effect of setting the program record pointer of the given table (see [Section 5.2 \[Tables\], page 18](#)) to the new record.

Example: ‘(NEW table NIL)’ allocates a new record in the given table and initializes it with the initial record.

See also NEW\*, DELETE, Tables.

### 15.18.2 NEW\*

NEW\* is the star version of NEW (see [Section 15.18.1 \[NEW\], page 121](#)).

`(NEW* table init)`

NEW\* checks if you have specified a ‘New’ trigger function for the given table (see [Section 15.29.5 \[New trigger\], page 139](#)). If so then this trigger function is called for allocating the record and its result is returned. The *init* parameter can be used to specify a record with which the new record should be initialized (use NIL for the initial record).

If no trigger function has been specified, the function behaves like the NEW function.

Warning: With this function it is possible to write endless loops, e.g. if you have defined a ‘New’ trigger function for a table and this function calls NEW\* to allocate the record.

See also NEW, DELETE\*.

### 15.18.3 DELETE

DELETE deletes a record in a table.

`(DELETE table requester)`

Deletes the current program record of the given table after an optional delete requester. The first argument specifies the table for which the current program record should be deleted, the second argument is a Boolean expression. If it is NIL then the record is deleted silently, if it is not NIL then the state of preferences menu item ‘Record delete requester’ is checked. If it is not set, the record is deleted silently, otherwise the standard delete requester appears asking for confirmation. If the users cancels the delete operation then the record will not be deleted.

The return code of the DELETE function reflects the selected action. If it returns TRUE then the record has been deleted, otherwise (the user has canceled the operation) NIL is returned.

On deletion, DELETE sets the program record pointer (see [Section 5.2 \[Tables\], page 18](#)) of the specified table to NIL.

Example: ‘(DELETE table NIL)’ deletes the current record in the given table silently.

See also DELETE\*, DELETEALL, NEW, Tables.

### 15.18.4 DELETE\*

DELETE\* is the star version of DELETE (see [Section 15.18.3 \[DELETE\]](#), page 121).

`(DELETE* table requester)`

DELETE\* checks if you have specified a ‘Delete’ trigger function for the given table (see [Section 15.29.6 \[Delete trigger\]](#), page 140). If so then this trigger function is called for deleting the record and its result is returned. The *requester* parameter can be used to specify if the trigger function should pop up a confirmation requester before deleting the record.

If no trigger function has been specified, the function behaves like the DELETE function.

Warning: With this function it is possible to write endless loops, e.g. if you have defined a ‘Delete’ trigger function for a table and this function calls DELETE\* to delete the record.

See also DELETE, DELETEALL, NEW\*.

### 15.18.5 DELETEALL

DELETEALL deletes all records of a table.

`(DELETEALL table[*])`

Deletes all records of the specified table. If you add a star behind the table name then only those records that match the current filter of the table are deleted. There is no safety requester before deleting the records.

DELETEALL returns TRUE on successful deletion of all records, otherwise NIL is returned. If *table* is NIL then NIL is returned.

Example: ‘(DELETEALL table\*)’ deletes all records in the given table that match the filter of the table.

See also DELETE, Tables.

### 15.18.6 GETMATCHFILTER

GETMATCHFILTER returns the match-filter state of a record.

`(GETMATCHFILTER rec)`

Returns TRUE if the specified record matches the filter of its table, NIL otherwise. If the filter of the table is currently not active then TRUE is returned. If *rec* is NIL (the initial record) then NIL is returned.

See also SETMATCHFILTER, GETISSORTED, GETFILTERSTR, SETFILTERSTR.

### 15.18.7 SETMATCHFILTER

SETMATCHFILTER sets the match-filter state of a record.

`(SETMATCHFILTER rec on)`

Changes the match-filter state of the specified record to the value of *on*. SETMATCHFILTER returns the new match-filter state of the given record. The new state may be different from the expected one because setting the match-filter state to NIL does only work when the filter of the corresponding table is currently active, otherwise TRUE is returned. Calling SETMATCHFILTER with a value of NIL for *rec* (the initial record) will always return NIL.

See also GETMATCHFILTER, SETISSORTED, GETFILTERSTR, SETFILTERSTR.

### 15.18.8 GETISSORTED

GETISSORTED returns the sorted state of a record.

```
(GETISSORTED rec)
```

Returns TRUE if the specified record is sorted in the order that has been defined for its table, NIL otherwise. If *rec* is NIL then NIL is returned.

See also SETISSORTED, GETMATCHFILTER, REORDER, GETORDERSTR, SETORDERSTR, Comparison function.

### 15.18.9 SETISSORTED

SETISSORTED sets the sorted state of a record.

```
(SETISSORTED rec on)
```

Changes the sorted state of the specified record to the value of *on*. Use this function if you think that a certain record is in the right order (*on* = TRUE), or it it should be reordered (*on* = NIL). Reordering all unsorted records can be done by calling the REORDER function (see [Section 15.20.4 \[REORDER\]](#), page 126).

SETISSORTED returns the new sorted state of the given record. Calling SETISSORTED with a value of NIL for *rec* (the initial record) will return NIL.

For an example on how to use this function, see [Section 15.29.7 \[Comparison function\]](#), page 140.

See also GETISSORTED, SETMATCHFILTER, REORDER, GETORDERSTR, SETORDERSTR, Comparison function.

### 15.18.10 RECNUM

RECNUM returns the record number of a record.

```
(RECNUM record)
```

Returns the record number of the given record. Please note that the numbering for records is different than e.g. for lists. For lists, strings and others the counting begins with zero, however for records it begins with 1 for the first record. The number 0 is reserved for the initial record. This seems to be inconsistent with the rest of the MUIbase programming functions, but it does really makes sense here as the record numbers are also used in the window display.

See also RECORDS, INT.

### 15.18.11 COPYREC

COPYREC copies records.

```
(COPYREC rec source)
```

Copies the contents of record *source* to record *rec*. If *source* is NIL then *rec* is set to the values of the initial record. If *rec* is NIL then an error message is generated.

COPYREC returns *rec*.

See also NEW.

## 15.19 Attribute Functions

This section lists functions that work on attributes of a table.

### 15.19.1 ATTRNAME

ATTRNAME returns the name of an attribute.

```
(ATTRNAME attr)
```

Returns a string containing the name of the specified attribute.

See also TABLENAME

### 15.19.2 MAXLEN

MAXLEN returns the maximum size of a string attribute.

```
(MAXLEN string-attr)
```

Returns the maximum number of characters that the given string attribute can hold.

See also LEN.

### 15.19.3 GETLABELS

GETLABELS returns all labels of a choice or string attribute.

```
(GETLABELS attr)
```

Returns the labels of the given choice or string attribute. In case of a choice attribute, the labels you entered in the attribute requester (see [Section 14.2.2 \[Type specific settings\]](#), [page 58](#)) are returned, in case of a string attribute, the static labels you entered for the list-view pop-up (see [Section 14.3.3 \[Attribute object editor\]](#), [page 62](#)) are returned (note that this function is only useful for static labels).

The labels are returned in one single string and are separated by newline characters.

For example, consider you have a choice attribute with the labels ‘Car’, ‘House’, and ‘Oil’. Then calling GETLABELS on that attribute will result to the string "Car\nHouse\nOil".

Note: you can easily convert the result string to a list by calling MEMOTOLIST (see [Section 15.13.3 \[MEMOTOLIST\]](#), [page 104](#)) on the result string.

See also SETLABELS.

### 15.19.4 SETLABELS

SETLABELS is used to set the labels of a string attribute.

```
(SETLABELS attr str)
```

Sets the static labels of the string attribute *attr* to the labels listed in the *str* argument. The *str* argument consists of lines each of which holds one label. The labels replace the ones you have entered for the pop-up list-view in the attribute object editor (see [Section 14.3.3 \[Attribute object editor\]](#), [page 62](#)). Note that this function is only useful for static labels.

SETLABELS returns the value of its *str* argument.

Example: ‘(SETLABELS Table.String "My house\nis\nyour house")’ sets the static list-view labels of the specifies string attribute to ‘My house’, ‘is’, and ‘your house’.



Note: you can easily convert a list of labels to the required string format by calling LISTTOMEMO on the list.

See also GETLABELS.

## 15.20 Table Functions

### 15.20.1 TABLENAME

TABLENAME returns the name of a table.

```
(TABLENAME table)
```

Returns a string containing the name of the specified table.

See also ATTRNAME

### 15.20.2 GETORDERSTR

GETORDERSTR returns the record order of a table.

```
(GETORDERSTR table)
```

Returns the current order expression of the given table. If the table uses an attribute list for ordering then the returned string contains the attribute names separated by spaces. Each attribute name is prepended by a '+' or a '-' sign indicating ascending or descending order.

If the table is ordered by a comparison function then the name of this function is returned.

An empty string means no ordering.

### Example

Consider a table 'Person' which is ordered by the Attributes 'Name' (ascending), 'Town' (ascending), and 'Birthday' (descending). Then '(ORDERSTR Person)' will result to the string "+Name +Town -Birthday".

See also SETORDERSTR, REORDER, REORDERALL, GETISSORTED, SETISSORTED, Order, Comparison function.

### 15.20.3 SETORDERSTR

SETORDERSTR sets the record order of a table.

```
(SETORDERSTR table order)
```

Sets the order of the given table according to the *order* string. The *order* string can either hold a list of attribute names or the name of a comparison function.

For sorting using an attribute list, the *order* string must contain the attribute names separated by any number of spaces, tabs or newlines. Each attribute name may be prepended by a '+' or a '-' sign indicating ascending or descending order. If you omit this sign then ascending ordering is assumed.

For sorting using a comparison function, the *order* string must hold the name of the function.

SETORDERSTR returns TRUE if it has been able to set the new order, NIL otherwise, e.g. if an unknown attribute has been specified or the type of the attribute is not allowed

for ordering. If you specify NIL for the *order* argument then nothing happens and NIL is returned.

Note: For building the order string you should not directly write the attribute names into the string because when you change an attribute name then the order string will not be updated. Better use the `ATTRNAME` function (see [Section 15.19.1 \[ATTRNAME\]](#), [page 124](#)) to copy the attribute's name into the order string.

## Example

Consider a table 'Person' with the attributes 'Name', 'Town', and 'Birthday'. Then '(SETORDERSTR Person (SPRINTF "+%s" (ATTRNAME Person.Name)))' will set the order of table 'Person' using 'Name' as (ascending) order attribute.

See also GETORDERSTR, REORDER, REORDERALL, GETISSORTED, SETISSORTED, Order, Comparison function.

### 15.20.4 REORDER

REORDER brings all unsorted records back into the right order.

```
(REORDER table)
```

Examines all records of the given table for unsorted ones and reinserts them to their right positions. After reinserting an unsorted record, the sorted state of the record is set to TRUE, thus on the return of REORDER the sorted state of all records is TRUE.

REORDER returns NIL.

Usually you only need to call this function, when you are using a comparison function for defining the order of a table. Orders defined by an attribute list are automatic, that is, a record is reordered automatically when needed.

For an example on how to use this function, see [Section 15.29.7 \[Comparison function\]](#), [page 140](#).

See also REORDERALL, GETORDERSTR, SETORDERSTR, GETISSORTED, SETISSORTED, Order, Comparison function.

### 15.20.5 REORDERALL

REORDERALL reorders all records of a table.

```
(REORDERALL table)
```

Reorders all records of the given table by setting the sorted state of all records to NIL and then calling REORDER for reordering everything.

REORDERALL returns NIL.

See also REORDER, GETORDERSTR, SETORDERSTR, GETISSORTED, SETISSORTED, Order, Comparison function.

### 15.20.6 GETFILTERACTIVE

GETFILTERACTIVE returns the filter state of a table.

```
(GETFILTERACTIVE table)
```

Returns TRUE if the filter of the specified table is currently activated and NIL otherwise.

See also SETFILTERACTIVE, GETFILTERSTR, GETMATCHFILTER.

### 15.20.7 SETFILTERACTIVE

SETFILTERACTIVE sets the filter state of a table.

```
(SETFILTERACTIVE table bool)
```

Sets the filter state of the specified table. If *bool* is non-NIL then the filter is activated, otherwise it is deactivated.

SETFILTERACTIVE returns the new state of the filter. The new state may not be the expected one in case you activate the filter but an error condition occurs and the filter can't be activated. However deactivating the filter always succeeds.

See also GETFILTERACTIVE, SETFILTERSTR, SETMATCHFILTER.

### 15.20.8 GETFILTERSTR

GETFILTERSTR returns the record filter expression of a table.

```
(GETFILTERSTR table)
```

Returns the record filter expression of the specified table as a string. An empty string means that no filter expression has been set for this table.

See also SETFILTERSTR, GETFILTERACTIVE, GETMATCHFILTER.

### 15.20.9 SETFILTERSTR

SETFILTERSTR sets the record filter expression of a table.

```
(SETFILTERSTR table filter-str)
```

Sets the record filter expression of the specified table to the expression in the *filter-str* argument (which must be a string and not the actual expression itself!). If the filter of the given table is currently active then the new filter expression is directly applied to all records and the match-filter state of all records are recomputed.

SETFILTERSTR returns TRUE if it has been able to compile the given filter string expression, otherwise NIL is returned. Note that you only get the result of the compilation. If the filter of the given table is currently active and recomputing all match-filter states of the corresponding records fails then you will not notice that from the result of this function. The recommended way to set a new filter expression is like follows:

```
(SETFILTERACTIVE Table NIL) ; always succeeds.
(IF (NOT (SETFILTERSTR Table filter-string))
  (ERROR "Can't set filter string for %s!" (TABLENAME Table))
)
(IF (NOT (SETFILTERACTIVE Table TRUE))
  (ERROR "Can't activate filter for %s!" (TABLENAME Table))
)
```

If SETFILTERSTR is called with a value of NIL for the *filter-str* argument then nothing happens and NIL is returned.

Example: '(SETFILTERSTR Table "> Value 0.0")'.

See also GETFILTERSTR, SETFILTERACTIVE, SETMATCHFILTER.

### 15.20.10 RECORDS

RECORDS returns the number of records in a table.

```
(RECORDS table)
```

Returns the number of records in the given table. You may append a star to the table name for counting the number of records that match the filter of the table.

See also RECORD, RECNUM.

### 15.20.11 RECORD

RECORD returns a record pointer for a given record number.

```
(RECORD table num)
```

Returns the record pointer to the *num*-th record in the given table or NIL if a record with this number doesn't exist. You may add a star to the table name to get the *num*-th record that matches the record filter of the table.

Please note that record numbers start with 1 and record number 0 is used for the initial record.

See also RECORDS, RECNUM.

### 15.20.12 SELECT

SELECT extracts and returns various data from records.

```
(SELECT [DISTINCT] exprlist FROM tablelist
      [WHERE where-expr] [ORDER BY orderlist])
```

where *exprlist* is either a simple star '\*' or a list of expressions with optional titles separated by commas:

```
exprlist:      * | expr "title", ...
```

and *tablelist* is a list of table names:

```
tablelist:      table [*] [ident], ...
```

For each table in the table list you can specify an identifier. This can be very useful if a table occurs more than once in the table list (see example of comparing ages below). If you add a star to a table then only the records matching the currently defined filter of that table will be examined.

The orderlist has the following syntax:

```
orderlist:      expr [ASC | DESC], ...
```

where *expr*, ... can be arbitrary expressions or field numbers. For example '(SELECT Name FROM ... ORDER BY 1)' will sort the result by the 'Name' attribute. You may specify ASC or DESC for an ascending or descending order. If none of them are present then an ascending order is assumed.

### How it works

The select-from-where query builds the (mathematical) cross product of all tables in the table list (it examines all sets of records in *table*, ...) and checks the where-expression (if any). If the where-expression results to TRUE (or there is no where-expression) then a list is build whose elements are calculated by the expression list in the select-part. If you have

specified a single star for the expression list then the list contains the values of all attributes belonging to tables in the table list (except virtual attributes and buttons).

The result of the query is a list of lists. The first list entry contains the title strings, the remaining ones contain the values of the from-list in the matching records.

## Examples

See [Section 13.4 \[Query examples\]](#), page 53, for some examples using the `SELECT` function.

See also `FOR ALL`.

## 15.21 GUI Functions

This section describes functions for manipulating GUI elements.

### 15.21.1 SETCURSOR

`SETCURSOR` sets the cursor on a GUI element.

`(SETCURSOR attr-or-table)`

Sets the cursor on the given attribute or table GUI object. The function also opens the window where the attribute/table resides in if the window was not open.

`SETCURSOR` returns `TRUE` if everything went ok (window could be opened) or `NIL` on failure.

See also `SETVIRTUALLISTACTIVE`.

### 15.21.2 GETWINDOWOPEN

`GETWINDOWOPEN` returns the open state of a window.

`(GETWINDOWOPEN attr-or-table)`

Returns the open state of the window where the given attribute/table resides.

See also `SETWINDOWOPEN`.

### 15.21.3 SETWINDOWOPEN

`SETWINDOWOPEN` opens and closes a window.

`(SETWINDOWOPEN attr-or-table open)`

Opens or closes the window in which the given attribute/table resides. If *open* is non-`NIL` then the window is opened, otherwise it is closed. You cannot close the main window of a project.

`SETWINDOWOPEN` returns the new open state of the window.

See also `GETWINDOWOPEN`.

### 15.21.4 GETVIRTUALLISTACTIVE

`GETVIRTUALLISTACTIVE` returns the index of the active line of a virtual attribute that uses the ‘List’ kind for display.

`(GETVIRTUALLISTACTIVE virtual-attr)`

Returns the logical index (starting with 1) of the currently active line in the specified virtual attribute. The logical index is the line number in the original list used when setting

the virtual attribute. It can differ from the display order in case the user changed the sorting, e.g. by clicking on a column title. If the GUI element of *virtual-attr* is not visible, or if it does not use the ‘List’ kind for display, or if no line is active, then NIL is returned.

See also SETVIRTUALLISTACTIVE.

### 15.21.5 SETVIRTUALLISTACTIVE

SETVIRTUALLISTACTIVE sets the active line of a virtual attribute that uses the ‘List’ kind for display.

(SETVIRTUALLISTACTIVE *virtual-attr num*)

Sets the active line of the given virtual attribute to the *num*-th logical line (starting with 1). The logical line is the line number in the original list used when setting the virtual attribute and can differ from the display order in case the user changed the sorting, e.g. by clicking on a column title.

Returns *num*, or NIL if the GUI element of *virtual-attr* is not visible, does not use the ‘List’ kind for display, or if *num* is out of range (less than 1 or greater than the number of lines).

SETVIRTUALLISTACTIVE does not set the cursor to the attribute’s GUI element, use SETCORSOR (see [Section 15.21.1 \[SETCORSOR\]](#), [page 129](#)) for doing so.

See also GETVIRTUALLISTACTIVE, SETCORSOR.

## 15.22 Project Functions

This section lists functions dealing with projects.

### 15.22.1 PROJECTNAME

PROJECTNAME returns the project name.

(PROJECTNAME)

PROJECTNAME returns the name of the current project as a string or NIL if no name has been defined yet. The project name is the pathname of the project directory in the file system.

See also CHANGES.

### 15.22.2 CHANGES

CHANGES returns the number of changes in the current project.

(CHANGES)

Returns an integer containing the number of changes since the last save operation of the current project.

See also PROJECTNAME.

### 15.22.3 GETADMINMODE

GETADMINMODE tells whether the current project is in admin mode or user mode.

(GETADMINMODE)

Returns TRUE if the current project is in admin mode, NIL otherwise.

See also SETADMINMODE, ADMINPASSWORD, onAdminMode.

### 15.22.4 SETADMINMODE

SETADMINMODE changes the current project to admin mode or user mode.

(SETADMINMODE *admin*)

If *admin* is NIL then the current project is set into user mode, otherwise into admin mode. Note that there is no authentication dialog when changing from user to admin mode using this function.

Returns TRUE if the project has been set to admin mode, or NIL if set to user mode.

See also GETADMINMODE, ADMINPASSWORD, onAdminMode, demo ‘Users.mb’.

### 15.22.5 ADMINPASSWORD

ADMINPASSWORD obtains the admin password as SHA1 hash string.

(ADMINPASSWORD)

Returns a string containing the SHA1 hash of the admin password of the current project. If no admin password has been set, NIL is returned.

See also GETADMINMODE, SETADMINMODE, SHA1SUM, demo ‘Users.mb’.

## 15.23 System Functions

This section lists functions accessing the operating system.

### 15.23.1 EDIT

EDIT launches the external editor.

(EDIT *filename*)

Starts the external editor for editing the specified file. The external editor can be set in menu item ‘Preferences – External editor’ (see [Section 7.1.2 \[External editor\]](#), page 32). EDIT starts the external editor synchronously, that is, it waits until the user exits the editor.

EDIT returns the return code of the external editor as an integer.

See also EDIT\*, VIEW, SYSTEM.

### 15.23.2 EDIT\*

EDIT\* is the star version of EDIT and has the same effect as EDIT (see [Section 15.23.1 \[EDIT\]](#), page 131). The only difference is that EDIT\* starts the external editor asynchronously, thus the function returns immediately.

EDIT\* returns 0 if it was successful in starting the editor, otherwise it returns a non-zero integer value representing a system specific error code.

See also EDIT, VIEW\*, SYSTEM\*.

### 15.23.3 VIEW

VIEW launches the external viewer.

(VIEW *filename*)

Starts the external viewer for displaying the specified file. The external viewer can be set in menu item ‘Preferences – External viewer’ (see [Section 7.1.3 \[External viewer\]](#), page 33). VIEW starts the external viewer synchronously, that is, it waits until the user exits

the viewer. Note that on some systems, the call might return immediately if an instance of the viewer is already running.

**VIEW** returns the return code of the external viewer as an integer.

See also **VIEW\***, **EDIT**, **SYSTEM**.

#### 15.23.4 **VIEW\***

**VIEW\*** is the star version of **VIEW** and has the same effect as **VIEW** (see [Section 15.23.3 \[VIEW\]](#), [page 131](#)). The only difference is that **VIEW\*** starts the external viewer asynchronously, thus the function returns immediately.

**VIEW\*** returns 0 if it was successful in starting the viewer, otherwise it returns a non-zero integer value representing a system specific error code.

See also **VIEW**, **EDIT\***, **SYSTEM\***.

#### 15.23.5 **SYSTEM**

**SYSTEM** calls an external program.

(**SYSTEM** *fmt* [*arg* ...])

Calls an external program. The command line to call the program is generated from *fmt* and the optional arguments like in the **SPRINTF** function (see [Section 15.12.32 \[SPRINTF\]](#), [page 102](#)). For interpreting the command line, a system specific shell is used (ShellExecute on Windows, /bin/sh on Mac OS and Linux, the user shell on Amiga). **SYSTEM** waits until the called program exits.

**SYSTEM** returns the return code of the executed command as an integer.

See also **SYSTEM\***, **EDIT**, **VIEW**.

#### 15.23.6 **SYSTEM\***

**SYSTEM\*** is the star version of **SYSTEM** and has the same effect as **SYSTEM** (see [Section 15.23.5 \[SYSTEM\]](#), [page 132](#)). The only difference is that **SYSTEM\*** executes the command line asynchronously, thus the function returns immediately.

**SYSTEM\*** returns 0 if it was successful in starting execution of the command line, otherwise it returns a non-zero integer value representing a system specific error code.

See also **SYSTEM**, **EDIT\***, **VIEW\***.

#### 15.23.7 **STAT**

**STAT** examines a filename.

(**STAT** *filename*)

Examines if the specified filename exists in the file system. **STAT** returns NIL if the filename could not be found, 0 if the filename exists and is a directory, and an integer value greater than 0 if the filename exists and is a regular file.

#### 15.23.8 **TACKON**

**TACKON** creates a pathname.

(**TACKON** *dirname* [*component* ...])



Joins *dirname* and all components in [*component* ...] to a pathname. **TACKON** knows how to deal with special characters used as path separators at the end of each argument. It returns the pathname as a string or NIL if any of the arguments is NIL. Note that **TACKON** does not perform any check whether the resulting path actually refers to an existing file or directory in the filesystem.

Example: ‘(TACKON "Sys:System" "CLI")’ results to "Sys:System/CLI".

See also FILENAME, DIRNAME.

### 15.23.9 FILENAME

FILENAME extracts the filename part of a path name.

(FILENAME *path*)

Extracts the last component of the given path name. There is no check whether *path* actually refers to a file, thus it is also possible to use FILENAME to get the name of a sub-directory. FILENAME returns its result as a string or NIL if *path* is NIL.

Example: ‘(FILENAME "Sys:System/CLI")’ results to "CLI".

See also DIRNAME, TACKON.

### 15.23.10 DIRNAME

DIRNAME extracts the directory part of a path name.

(DIRNAME *path*)

Extracts the directory part of the given path name. There is no check whether *path* actually refers to a file, thus it is also possible to use DIRNAME to get the name of a parent directory. DIRNAME returns its result as a string or NIL if *path* is NIL.

Example: ‘(DIRNAME "Sys:System/CLI")’ results to "Sys:System".

See also FILENAME, TACKON.

### 15.23.11 MESSAGE

MESSAGE displays a message to the user.

(MESSAGE *fmt* [*arg* ...])

Sets the window title of the pause/abort window (if it is open). The title string is generated from *fmt* and the optional arguments like in the SPRINTF function (see [Section 15.12.32 \[SPRINTF\]](#), page 102).

MESSAGE returns the formatted title string.

Example: ‘(MESSAGE "6 \* 7 = %i" (\* 6 7))’.

See also PRINT, PRINTF.

### 15.23.12 COMPLETMAX

COMPLETMAX sets the maximum number of progress steps.

(COMPLETMAX *steps*)

Sets the maximum number of steps for showing the progress of your MUIbase program to the user. The default (if you do not call this function) is 100 steps. The argument *steps* must be an integer value. If *steps* is NIL or 0 then no progress bar is displayed. The progress

bar is part of the pause/abort window that pops up after a short delay when executing a MUIbase program.

COMPLETMAX returns its argument *steps*.

See also COMPLETEADD, COMPLETE.

### 15.23.13 COMPLETEADD

COMPLETEADD increases the progress state.

```
(COMPLETEADD add)
```

Adds the integer expression *add* to the current progress value. Initially, the progress value is set to 0. If *add* is NIL then the progress value is reset to 0 and no progress bar is shown.

COMPLETEADD returns its argument *add*.

#### Example:

```
(SETQ num ...)
(COMPLETMAX num)
(DOTIMES (i num)
  (COMPLETEADD 1)
)
```

See also COMPLETMAX, COMPLETE.

### 15.23.14 COMPLETE

COMPLETE sets the progress state.

```
(COMPLETE cur)
```

Sets the current progress value to the integer expression *cur*. If *cur* is NIL or 0 then no progress bar is shown.

COMPLETE returns its argument *cur*.

#### Example:

```
(COMPLETE 10)
...
(COMPLETE 50)
...
(COMPLETE 100)
```

See also COMPLETMAX, COMPLETEADD.

### 15.23.15 GC

GC forces garbage collection.

```
(GC)
```

Forces garbage collection and returns NIL. Normally garbage collection is done automatically from time to time.

### 15.23.16 PUBSCREEN

PUBSCREEN returns name of public screen.

(PUBSCREEN)

On Amiga, PUBSCREEN returns the name of the public screen MUIbase is running on, or NIL in case the screen is not public.

On other systems PUBSCREEN returns NIL.

## 15.24 Pre-Defined Variables

MUIbase knows some pre-defined global variables.

In the current version there exists only one global variable: `stdout` (see [Section 15.17.3 \[stdout\]](#), page 117).

## 15.25 Pre-Defined Constants

The following pre-defined constants can be used in any expression for programming.

Name	Type	Value	Comment
-----			
NIL	any	NIL	
TRUE	Boolean	TRUE	
RESET	string	"\33c"	
NORMAL	string	"\33[0m"	
ITON	string	"\33[3m"	
ITOFF	string	"\33[23m"	
ULON	string	"\33[4m"	
ULOFF	string	"\33[24m"	
BFON	string	"\33[1m"	
BFOFF	string	"\33[22m"	
ELITEON	string	"\33[2w"	
ELITEOFF	string	"\33[1w"	
CONDON	string	"\33[4w"	
CONDOFF	string	"\33[3w"	
WIDEON	string	"\33[6w"	
WIDEOFF	string	"\33[5w"	
NLQON	string	"\33[2\"z"	
NLQOFF	string	"\33[1\"z"	
INT_MAX	integer	2147483647	Maximum integer value
INT_MIN	integer	-2147483648	Minimum integer value
HUGE_VAL	real	1.797693e+308	Largest absolute real value
PI	real	3.14159265359	
OSTYPE	string	<OS type>	"Windows", "MacOSX", "Unix" or "Amiga"
OSVER	integer	<OS version>	
OSREV	integer	<OS revision>	
MBVER	integer	<MUIbase version>	
MBREV	integer	<MUIbase revision>	
LANGUAGE	string	depends	Default language

SEEK_SET	integer	see <code>stdio.h</code>	Seek from beg. of file
SEEK_CUR	integer	see <code>stdio.h</code>	Seek from current pos
SEEK_END	integer	see <code>stdio.h</code>	Seek from end of file

See [Section 15.4.7 \[Constants\]](#), page 76, for more information about constants. For defining your own constants, use the `#define` preprocessing directive (see [Section 15.3.1 \[#define\]](#), page 71).

## 15.26 Functional Parameters

You can pass a function as an argument to another function. This is useful for defining functions of higher order, e.g. for sorting or mapping a list.

To call a function that has been passed in an argument you have to use the `FUNCALL` function (see [Section 15.6.6 \[FUNCALL\]](#), page 81).

### Example:

```
(DEFUN map (l fun)                # arguments: list and function
  (LET (res)                     # local variable res, initialized with NIL
    (DOLIST (i l)                # for all items one by one
      (SETQ res
        (CONS (FUNCALL fun i) res) # calls function and
      )                          # build new list
    )
    (REVERSE res)                # we need to reverse the new list
  )
)
```

You can now use the `map` function for example to increment all items of a list of integers:

`'(map (LIST 1 2 3 4) 1+)'` results to `( 2 3 4 5 )`.

See also `FUNCALL`, `APPLY`, `MAPFIRST`.

## 15.27 Type Specifiers

It is possible to specify the type of a variable by adding a type specifier behind the name. The following type specifiers exist:

### Specifier      Description

<code>:INT</code>	for integers
<code>:REAL</code>	for reals
<code>:STR</code>	for strings
<code>:MEMO</code>	for memos
<code>:DATE</code>	for dates
<code>:TIME</code>	for times
<code>:LIST</code>	for lists
<code>:FILE</code>	for file handles
<code>:FUNC</code>	for functions of any type

```
:table      for record pointers to table
```

The type specifier appends the variable name as in the following example:

```
(LET (x:INT (y:REAL 0.0) z) ...)
```

The example defines three new variables ‘x’, ‘y’ and ‘z’, where ‘x’ is of type integer and initialized with NIL, ‘y’ is of type real and initialized with 0.0, and ‘z’ is an untyped variable initialized with NIL.

The advantage of the type specifiers is that the compiler can detect more type errors, e.g. if you have a function:

```
(DEFUN foo (x:INT) ...)
```

and call it with ‘(foo "bar")’ then the compiler generates an error message. However, if you call ‘foo’ with an untyped value, e.g. ‘(foo (FIRST list))’ then no error checking can be done at compile time because the type of ‘(FIRST list)’ is unknown.

For reasons of speed no type checking is currently done at run time. It could be implemented but then would add a little overhead which is not really necessary as the wrong type will result in a type error sooner or later anyway.

Type specifiers for record pointers have another useful feature. If you tag a variable as a record pointer to a table then you can access all attributes of this table by using the variable name instead of the table name in the attribute path. E.g. if you have a table ‘Foo’ with an attribute ‘Bar’, and you define a variable ‘foo’ as:

```
(LET (foo:Foo)
```

then you can print the ‘Bar’ attribute of the third record by using:

```
(SETQ foo (RECORD Foo 3)) (PRINT foo.Bar)
```

Note that in a select-from-where expression, variables defined in the from list automatically have a type of record pointer to the corresponding table.

## 15.28 Semantics of Expressions

The semantics of expressions are very important for understanding what a program does. This section lists the semantics depending on syntactical expressions.

(*func* [*expr* ...])

Evaluates *expr* ... and then calls the function *func* (call by value). Returns the return value of the called function. In MUIbase there exists some non-strict functions, e.g. AND, OR and IF. These functions may not evaluate all expressions. For more information about non-strict functions, see [Section 15.4.2 \[Lisp syntax\]](#), page 73, [Section 15.9.1 \[AND\]](#), page 89, [Section 15.9.2 \[OR\]](#), page 90, and [Section 15.6.8 \[IF\]](#), page 81.

(*expr* ...)

Evaluates *expr* ... and returns the value of the last expression (see [Section 15.6.1 \[PROGN\]](#), page 79). An empty expression () evaluates to NIL.

*Table*

Returns the program record pointer of the given table.

*Table\**

Returns the GUI record pointer of the given table.

*AttrPath*

Returns the contents of the specified attribute. The attribute path specifies which record is used to extract the attribute value. For example ‘Table.Attribute’ uses the program record of ‘Table’ to extract the value of the attribute, ‘Table.ReferenceAttribute.Attribute’ uses the program record of ‘Table’ to extract the value of the reference attribute (which is a record pointer) and then uses this record to extract the value of ‘Attribute’.

*var*

Returns the contents of the global or local variable *var*. Global variables can be defined with DEFVAR (see [Section 15.5.3 \[DEFVAR\]](#), page 78), local variables e.g. with LET (see [Section 15.6.3 \[LET\]](#), page 79).

*var.AttrPath*

Uses the record pointer of *var* to determine the value of the specified attribute.

## 15.29 Function Triggering

For automatic execution of MUIbase programs you can specify trigger functions for projects, tables and attributes which are called in specific cases. This section lists all available trigger possibilities.

### 15.29.1 onOpen

After opening a project, MUIbase searches the project’s program for a function called `onOpen`. If such a function exists then this function is called without any arguments.

#### Example

```
(DEFUN onOpen ()
  (ASKBUTTON NIL "Thank you for opening me!" NIL NIL)
)
```

See also `onClose`, `onAdminMode`, `onChange`, demo ‘Trigger.mb’.

### 15.29.2 onClose

Before closing a project, MUIbase searches the project’s program for a function called `onClose`. If such a function exists then this function is called without any arguments. In the current version the result of the trigger function is ignored and the project is closed regardless of its return value.

If you do changes to the project in the `onClose` function then MUIbase will ask you for saving the project first before it is actually closed. If you use menu item ‘Project – Save & Close’ for closing the project, the trigger function is called before saving the project, thus the changes are saved automatically.

#### Example

```
(DEFUN onClose ()
```

```
(ASKBUTTON NIL "Good bye!" NIL NIL)
)
```

See also `onOpen`, `onChange`, demo `'Trigger.mb'`.

### 15.29.3 onAdminMode

Whenever a project is set into admin mode or user mode and a function called `onAdminMode` exists in the project's program then this function is called. The function receives one argument *admin* telling whether the project is in admin mode (*admin* is non-NIL) or in user mode (*admin* is NIL).

#### Example

```
(DEFUN onAdminMode (admin)
  (IF admin
    (ASKBUTTON NIL "Now in admin mode" NIL NIL)
    (ASKBUTTON NIL "Back to user mode" NIL NIL)
  )
)
```

See also `onOpen`, `onChange`, `SETADMINMODE`, demo `'Users.mb'`.

### 15.29.4 onChange

Whenever the user makes changes to a project or after saving a project, MUIbase searches the project's program for a function called `onChange`. If such a function exists then this function is called without any arguments. This can be used to count the changes a user does to a project.

#### Example

```
(DEFUN onChange ()
  (SETQ Control.NumChanges (CHANGES))
)
```

In the above example `'Control.NumChanges'` could be a virtual attribute somewhere in an `'exactly-one-record'` table for displaying the project's number of changes.

See also `onOpen`, `onClose`, `onAdminMode`, demo `'Trigger.mb'`.

### 15.29.5 New Trigger

When the user wants to allocate a new record by selecting one of the menu items `'New record'` or `'Duplicate record'` and the `'New'` trigger of that table has been set to a MUIbase program function then this trigger function is executed. The `'New'` trigger function can be set in the table requester (see [Section 14.1.1 \[Creating tables\]](#), page 56).

The trigger function receives NIL or a record pointer as the first and only argument. NIL means that the user wants to allocate a new record, a record pointer means that the user wants to duplicate this record. If the trigger function has more than one argument then these are initialized with NIL. The trigger function should allocate the new record by calling the `NEW` function (see [Section 15.18.1 \[NEW\]](#), page 121). The result returned by the trigger function will be examined. If it returns a record pointer then this record will be displayed.

The ‘New’ trigger is also called when a MUIbase program calls the NEW\* function (see [Section 15.18.2 \[NEW\\*\], page 121](#)).

### Sample New trigger function

```
(DEFUN newRecord (init)
  (PROG1
    (NEW Table init)
    ...
  )
)
```

See also NEW, NEW\*, Delete trigger.

### 15.29.6 Delete Trigger

When the user wants to delete a record by selecting menu item ‘Delete record’ and the ‘Delete’ trigger of that table has been set to a MUIbase program function then this trigger function is executed. The ‘Delete’ trigger function can be set in the table requester (see [Section 14.1.1 \[Creating tables\], page 56](#)).

The trigger function receives a Boolean argument as its only argument. If the argument is non-NIL then the function should ask the user if he really wants to delete the record. If so, the function should call DELETE (see [Section 15.18.3 \[DELETE\], page 121](#)) for deleting the record.

The ‘Delete’ trigger is also called when a MUIbase program calls the DELETE\* function (see [Section 15.18.4 \[DELETE\\*\], page 122](#)).

### Sample Delete trigger function

```
(DEFUN deleteRecord (requester)
  (DELETE Table requester)
)
```

See also DELETE, DELETE\*, New trigger.

### 15.29.7 Comparison Function

For defining an order on the records of a table you can use a comparison function. See [Section 10.4 \[Changing orders\], page 45](#), for information on how to specify such a function for a table. The function takes two record pointers as arguments and returns an integer value reflecting the order relation of the two records. The comparison function should return a value smaller than 0 if its first argument is smaller than the second, 0 if they are equal and a value larger than 0 if the first argument is greater than the second one.

For example, if you have a table ‘Persons’ with a string attribute ‘Name’ then you could use the following function for comparing two records:

```
(DEFUN cmpPersons (rec1:Persons rec2:Persons)
  (CMP rec1.Name rec2.Name)
)
```

This will order all records according to the ‘Name’ field using case sensitive string comparison. Note that by using an attribute list you could not archive the same ordering as string comparison is done case insensitive for attribute lists.



Using a comparison function you can define very complex order relations. Be careful not to create recursive functions that will call itself. MUIbase will stop program execution and prompt you with an error message if you try to do so. Also you should not use commands that cause side effects, e.g. setting a value to an attribute.

When using a comparison function, MUIbase does not always know when it has to reorder its records. E.g. consider in the above example another table ‘Toys’ with a string attribute ‘Name’ and a reference attribute ‘Owner’ referring to ‘Persons’, and the following function for comparing records:

```
(DEFUN cmpToys (rec1:Toys rec2:Toys)
  (CMP* rec1.Owner rec2.Owner)
)
```

This function is using the order of ‘Persons’ for determining the order of the records, thus the records of ‘Toys’ are sorted according to the order of ‘Persons’.

Now if the user changes one record in the ‘Persons’ table and this record gets a new position then also records in ‘Toys’ that refer to this record have to be reordered. MUIbase, however, does not know of this dependency.

Besides using menu item ‘Table – Reorder all records’ on the ‘Toys’ table for rearranging the order, you can implement an automatic reordering yourself by specifying the following trigger function for the ‘Name’ attribute of ‘Persons’:

```
(DEFUN setName (newValue)
  (SETQ Persons.Name newValue)
  (FOR ALL Toys WHERE (= Toys.Owner Persons) DO
    (SETISSORTED Toys NIL)
  )
  (REORDER Toys)
)
```

The function clears the sorted flag for all records that refer to the current record of table ‘Persons’ and then reorders all unsorted records of table ‘Toys’.

See also Order, CMP, GETISSORTED, SETISSORTED, REORDER, REORDERALL, GETORDERSTR, SETORDERSTR, demo ‘Order.mb’.

### 15.29.8 Attribute Trigger

In the attribute requester (see [Section 14.2.1 \[Creating attributes\], page 57](#)) you can define a trigger function that is called whenever the user wants to change the attribute contents.

If you have defined such a trigger function for an attribute and the user changes the value of that attribute then the record contents are not automatically updated with the new value. Instead the value is passed to the trigger function as first argument. The trigger function can now check the value and may refuse it. To store the value in the record you have to use the SETQ function.

The trigger function should return the result of the SETQ call or the old value of the attribute if it decides to refuse the new one.

The trigger function is also called when a MUIbase program calls the SETQ\* function (see [Section 15.6.5 \[SETQ\\*\], page 80](#)) for setting a value to the attribute.

## Sample attribute trigger function

```
(DEFUN setAmount (amount)
  (IF some-expression
    (SETQ Table.Amount amount)
    (ASKBUTTON NIL "Invalid value!" NIL NIL))
  )
  Table.Amount ; return current value
)
```

See also SETQ\*

## 15.29.9 Programming Virtual Attributes

In MUIbase virtual attributes are special attributes that calculate their value on the fly whenever it is needed. E.g. if you go to another record by clicking on one of the arrow buttons in a table's panel bar then a virtual attribute in that table is automatically recomputed and displayed (given appropriate settings for the virtual attribute, see [Section 14.3.3 \[Attribute object editor\]](#), page 62). For computing the value the attribute's 'Compute' trigger function is called. This trigger function can be specified in the attribute requester (see [Section 14.2.2 \[Type specific settings\]](#), page 58). The return value of this function defines the value of the virtual attribute. If you don't specify a 'Compute' trigger function for a virtual attribute then the attribute's value is NIL.

You can also trigger the calculation of a virtual attribute by simply accessing it in an MUIbase program, so e.g. if you have a button that should compute the value of the virtual attribute, you only need to specify a function for the button like the following one:

```
(DEFUN buttonHook ()
  virtual-attr
)
```

You can also set a virtual attribute to any value by using the SETQ function:

```
(SETQ virtual-attr expr)
```

However if you access the virtual attribute after the SETQ call then the value of the virtual attribute is recomputed.

There is no caching of the value of a virtual attribute because it's not easy to know when the value has to be recomputed and when not. Thus you should access virtual attributes rarely and cache the value in local variables for further use by yourself.

For an example on how to use virtual attributes please check the 'Movie.mb' demo.

See also Virtual, demo 'Movie.mb'.

## 15.29.10 Compute Enabled Function

For attribute objects and window buttons you can specify a trigger function for computing the enabled state of the object. See [Section 14.3.3 \[Attribute object editor\]](#), page 62 and [Section 14.3.10 \[Window editor\]](#), page 68, for information on how to specify this trigger function.

The trigger function is called without any arguments. It should return NIL for disabling the object and non-NIL for enabling it.

For example the compute enabled function for an object, that gets enabled when a certain virtual attribute using the ‘List’ kind has an active item, could look like this:

```
(DEFUN enableObject ()
  (GETVIRTUALLISTACTIVE virtual-list-attr)
)
```

See also Attribute object editor, Window editor, demo ‘Users.mb’.

### 15.29.11 Double Click Trigger

For virtual attributes that use the list kind for displaying its contents, a trigger function can be specified which gets called whenever the user double clicks on an item in the list. See [Section 14.3.3 \[Attribute object editor\]](#), page 62, for information on how to specify this trigger function for a virtual attribute object.

The trigger functions is called with three arguments. The first argument holds the row number of the clicked field, starting with 1 for the first line (row 0 refers to the list header). The second argument holds the column number starting with 0. The third argument is a pointer to the record the list field has been generated from, or NIL if the entry has not been generated directly from one. The return code of the function is ignored.

A typical example of a double click trigger is the following.

```
(DEFUN doubleClickTrigger (row col rec:Table)
  ...
)
```

Here *rec* is declared as a record pointer to *Table*. This way attributes of *Table* can be accessed directly from *rec*.

In case there is more than one possible table the record argument could belong to, the following construction using type predicates to different tables is useful.

```
(DEFUN doubleClickTrigger (row col rec)
  (COND
    ((RECP Table1 rec) (SETQ Table1 rec) ...)
    ((RECP Table2 rec) (SETQ Table2 rec) ...)
    ...
  )
)
```

The list item the user clicked on might not be related to any record. In this case, the third argument can be ignored and the list element be accessed like in the following example.

```
(DEFUN doubleClickTrigger (row col)
  (PRINT (NTH col (NTH row virtual-attr)))
)
```

See also Attribute object editor, demo ‘Movie.mb’.

### 15.29.12 Compute Listview Labels

For string attributes, the GUI object can contain a listview pop-up that when pressed opens a list of label strings the user can choose from. The labels in this list can be static labels or they can be computed by a trigger function. See [Section 14.3.3 \[Attribute object editor\]](#),

page 62, for information on how to select between static and computed labels and how to specify the trigger function.

The trigger function for computing the labels does not have any arguments. It should return a memo text with one label per line or NIL for no labels.

For example the compute function could look like this:

```
(DEFUN computeLabels ()
  "Tokyo\nMunich\nLos Angeles\nRome"
)
```

See also Compute reference records, Attribute object editor.

### 15.29.13 Compute Reference Records

For reference attributes, the GUI object usually contains a pop-up button that when pressed opens a list of records the user can choose from. The list of records in this pop-up can be computed by a trigger function. See Section 14.3.3 [Attribute object editor], page 62, for information on how to specify the trigger function for reference fields.

The function for computing the list of records does not have any arguments. It should return a list that is searched for the occurrence of records of the referenced table. Any such record found is added to the list shown in the reference pop-up. Items that are not records of the referenced table are ignored silently.

A typical compute function for reference records is the following example. Say a project contains a table 'Person' with a Boolean attribute 'Female'. Then the following compute function only shows the female persons in the reference popup:

```
(DEFUN computeFemaleRecords ()
  (SELECT Person FROM Person WHERE Female)
)
```

See also Compute list-view labels, Attribute object editor.

## 15.30 List of Obsolete Functions

The following functions are obsolete starting with MUIbase version 2.7.

- GETDISABLED
- SETDISABLED
- GETWINDOWDISABLED
- SETWINDOWDISABLED

Obsolete functions are not working as expected any longer and calling any of them is either ignored (resulting in a no-operation), opens a warning dialog, or causes an error, depending on the setting of menu item 'Program - Obsolete functions' (see Section 7.2.8 [Obsolete functions], page 36).

It is recommended to remove these functions from a project program and implement the functionality using the enabled/disabled setting of attribute objects and window buttons (see Section 15.29.10 [Compute enabled function], page 142).

## 16 ARexx Interface

The ARexx interface is only available in the MUIbase version for Amiga.

ARexx is a standard interface for Amiga programs to enable access to functions and data from other programs. MUIbase provides such an ARexx port with a small but well defined set of commands that enable an external program to virtually compute everything a MUIbase program can compute. Furthermore MUIbase' ARexx interface implements a transaction mechanism similar to other relational databases.

Example ARexx scripts for MUIbase can be found in the 'rex' directory.

### 16.1 Port Name

The port name of the MUIbase ARexx port is 'MUIbase.*n*' where *n* is a counter starting with 1. Usually, if you start MUIbase only once, the port name is 'MUIbase.1'.

You need the port name in the ARexx `address` statement before calling any of MUIbase' ARexx commands. The following program fragment shows how to check for the presence of a MUIbase ARexx port, start MUIbase if needed, and address the port.

```
if ~show(ports, MUIbase.1) then
do
    address command 'run <nil: >nil: MUIbase:MUIbase -n'
    address command 'waitforport MUIbase.1'
end

address MUIbase.1
```

See also sample ARexx script 'address.rex'.

### 16.2 Command Syntax

After addressing the MUIbase ARexx port you can call any of MUIbase' ARexx commands. The syntax is the same as for most other implementations:

```
cmd [arg1 ...]
```

where *cmd* is one of the commands described further down in this chapter, and *arg1* ... are optional arguments to the command.

Since the ARexx interpreter evaluates the command line before sending it to MUIbase, it is sometimes useful to quote some or all of the arguments. It is recommended to use single quotes (') around arguments which should not be further evaluated by the ARexx interpreter. This way you can still place double quotes ("), e.g. for string constants, in the arguments. Furthermore you can integrate the value of ARexx variables by unquoting them. Here is an example using MUIbase' `eval` command.

```
search = ''
eval handle 'select Name from Person where (like Name "*"search"*)'
```

See also Eval.

## 16.3 Return Codes

After calling one of MUIbase' ARexx commands several ARexx variables are updated with the result of the command. For reading all of the results of a command you should enable the ARexx results option by adding the following line at the beginning of your ARexx script.

```
options results
```

There are 3 ARexx variables that can be set by the MUIbase ARexx interface: *rc*, *results* and *lasterror*. Variable *rc* is always set and reflects the success or failure of a command. If a command was successful, *results* holds the actual result of the command, whereas in the case of an unsuccessful command, *lasterror* can hold additional information describing the error.

For variable *rc* the following return codes exist:

Return code	Meaning
0	Success. Variable <i>result</i> holds the actual result.
-1	Implementation error. Should never happen.
-2	Out of memory.
-3	Unknown ARexx command.
-4	Syntax error.
<= -10	Other error. Error description can be found in <i>lasterror</i> .
-12	Compilation error (only for compile command).

Note that only for *rc* <= -10 variable *lasterror* holds a valid error description. In the future more error codes might be added to enable a more detailed error handling.

Here is a typical code fragment showing how to examine the result of a MUIbase ARexx command.

```
eval handle 'select * from Accounts'
if (rc == 0) then
  say result
else if (rc == -1) then
  say "Implementation error"
else if (rc == -2) then
  say "Out of memory"
else if (rc == -3) then
  say "Unknown command"
else if (rc == -4) then
  say "Command syntax error"
else if (rc <= -10) then
  say lasterror
else
  say "Error: " rc
```

## 16.4 Quit

The quit command causes MUIbase to exit. See also the MUI documentation.

## 16.5 Hide

Command **hide** iconifies all open MUIbase windows. See also the MUI documentation.

## 16.6 Show

Command **show** deiconifies MUIbase and reopens the windows. See also the MUI documentation.

## 16.7 Info

The **info** command provides information about title, author, copyright, description, version, base and screen of a MUI application.

Command	Value of <i>result</i>
<b>info title</b>	Title of application
<b>info author</b>	Author of application
<b>info copyright</b>	Copyright message
<b>info description</b>	Short description
<b>info version</b>	Version string
<b>info base</b>	Name of ARexx port
<b>info screen</b>	Name of public screen

See also the MUI documentation.

## 16.8 Help

The **help** command writes a file containing all available ARexx commands of a MUI application.

**help filename**

ARexx commands are listed using the AmigaDos command line syntax. See the MUI documentation for more information and your AmigaDos manual for the command line syntax.

## 16.9 Compile

The **compile** command compiles an external program source file.

**compile source [update]**

The command compiles the external program source of the project whose external source filename points to the same file as *source*. On success, the command returns 0 and, if **update** has been specified, re-writes the external source file. Updating the source file allows to pretty print the MUIbase keywords. A successfully compiled program is then used as the project's program and used when executing trigger functions.

If compilation fails, an error code of -12 is returned and *lasterror* is set to a string containing 4 lines:

- The first line holds the name of the file the error occurred.
- The second line holds the error line number starting with 1.
- The third line holds the error column starting with 1.
- The fourth line describe the error in readable text.

Note that a project must have already been opened before sending the `compile` command for compiling its external source. In case no project whose external source file points to *source* is found, an error code  $\leq -10$  (but different from -12) is returned and *lasterror* is set.

## 16.10 Connect

The `connect` command opens the communication to a MUIbase project.

```
connect project-name [GUI]
```

The command first checks if the project specified by *project-name* has already been opened and loads it if needed. A project is only opened once and multiple connections to the same project share the access to the database. Next a unique communication handle is generated. A communication handle is an integer value not equal to zero. If the keyword `GUI` has been added to the command line then also the graphical user interface of the MUIbase project is opened. Otherwise no graphical user interface is generated allowing to process MUIbase ARexx commands in the background without direct user interaction.

On success, the command returns 0 and sets *result* to the value of the handle.

Example: ‘`connect "MUIbase:Demos/Movies.mb"`’ establishes a connection to the sample movie database.

See also Disconnect, Connections, Return codes.

## 16.11 Disconnect

The `disconnect` command closes an existing connection.

```
disconnect handle
```

Closes the database connection given in *handle*. If this was the only connection to the project *handle* refers to and there is no graphical user interface for the project then the project is closed and removed from memory. Otherwise the project stays open.

Example: ‘`disconnect 1`’ closes connection with handle value 1.

See also Connect, Connections, Return codes.

## 16.12 Connections

To find out about existing connections, use the `connections` command.

```
connections
```

On success, `connections` returns 0 and sets *result* to a readable string where each line shows a connection consisting of handle value and project name.

Example: the *result* variable after a `connections` call could look like the following:

```
3 MUIbase:Demos/Accounts.mb
5 MUIbase:Demos/Movies.mb
```



```
6 MUIbase:Demos/Movies.mb
7 MUIbase:Demos/Movies.mb
```

See also Connect, Disconnect, Return codes.

### 16.13 Eval

The main interface of MUIbase' ARexx port for retrieving and updating data is through the `eval` command.

```
eval handle lisp-cmd
```

The `eval` command interprets the given command *lisp-cmd* (written in MUIbase' lisp language) on the project specified by *handle*. A handle can be obtained by the `connect` command. The command *lisp-cmd* can be any expression of MUIbase' programming language. Optionally the outmost parenthesis around the expression can be omitted. It is recommended to surround *lisp-cmd* by single quotes as described in [Section 16.2 \[Command syntax\]](#), page 145.

If successful, `eval` returns 0 and sets *result* to a string representation of the return value of *lisp-cmd*. The string representation is done in a way that still allows to find out what kind of data is returned, e.g. strings are surrounded by double quotes and lists are surrounded by parenthesis with items separated by spaces or newline characters. If you want to ensure a certain format, use your own formatting withing the specified lisp command.

If you did changes to the database within the `eval` command and did not start a transaction first (see [Section 16.14 \[Transaction\]](#), page 150) then the changes are automatically made permanent (auto commit). Otherwise (you did start a transaction before calling `eval`) the changes are kept in memory until a `commit` command makes them permanent or a `rollback` command undoes the changes.

Example:

```
options results
address MUIbase.1
connect "MUIbase:Demos/Movie.mb"
if (rc == 0) then
do
    handle = result
    eval handle 'select Title, Director from Movies'
end
if (rc == 0) then
    say result
```

The output of the above example could look like this:

```
( ( "Title" "Director" )
  ( "Batman" "Tim Burton" )
  ( "Batman Returns" "Tim Burton" )
  ( "Speechless" "Ron Underwood" )
  ( "Tequila Sunrise" "Robert Towne" )
  ( "Mad Max" "George Miller (II)" )
  ( "Braveheart" "Mel Gibson" )
  ( "2010" "Peter Hyams" ) )
```

See also `Connect`, `Command syntax`, `Return codes`, `Transaction`, `Commit`, sample ARexx script `'movies.rexx'`.

## 16.14 Transaction

MUIbase' ARexx port allows to do transactions on a database. A transaction is a set of commands to a database (allowing modifications of data in the database) that is either executed and made permanent completely (`commit`) or withdrawn at any point within the transaction (`rollback`). A transaction can be started by issuing the following command:

```
transaction handle
```

where *handle* refers to a project as obtained by the `connect` command (see [Section 16.10 \[Connect\]](#), page 148).

After issuing a `transaction` command you can put as many `eval` commands as you like without actually changing the database. At some point, however, you have to decide if you want to make changes permanent (see [Section 16.15 \[Commit\]](#), page 150) or to go back to the state before issuing the `transaction` command (see [Section 16.16 \[Rollback\]](#), page 150).

After issuing a `transaction` command, access to the corresponding project is made exclusive to the specified handle. Thus other programs trying to access the database including the user accessing MUIbase by its graphical user interface are blocked (or delayed in case of another ARexx connection) until the exclusive access is released by calling the `commit` or `rollback` command.

Usually the `transaction` command returns 0. In case another ARexx connection gained exclusive access to the same project as *handle* refers to, the call is blocked until the other connection finished by committing or rolling back the database.

See also `Eval`, `Commit`, `Rollback`, `Return codes`.

## 16.15 Commit

The `commit` command is used at the end of a transaction to make changes permanent.

```
commit handle
```

The `commit` command ends a transaction (see [Section 16.14 \[Transaction\]](#), page 150) by saving the project the argument *handle* refers to. On success, `commit` returns 0. If you did not start a transaction before calling `commit` or if some other error occurs a value not equal to 0 is returned.

See also `Rollback`, `Transaction`, `Return codes`.

## 16.16 Rollback

To cancel changes made in a transaction use the `rollback` command.

```
rollback handle
```

All changes made in the project referred to by *handle* since the beginning of the current transaction (see [Section 16.14 \[Transaction\]](#), page 150) are withdrawn and the state of the project before the transaction is restored. If successful `rollback` returns 0.

See also `Commit`, `Transaction`, `Return codes`.

## Acknowledgments

Thanks to:

- Ralph Reuchlein (Ralphie) for bug reports, ideas and suggestions, and for the German translation of the MUIbase user manual.

Ralphie also created and maintains the MUIbase home page <http://muibase.sourceforge.net>.

- Pascal Marcelin for his believe in MUIbase and Amiga, and for the first web server connecting to MUIbase through its ARexx interface.
- Christoph Pölzl and Sébastien Pölzl for the artwork used in MUIbase, the PNG icon set and for testing on MorphOS.
- Alexandre Balaban for the French translation of the user manual (with great help from Lionel Muller and Gilles Mathevet) and for porting MUIbase to AmigaOS 4.
- Ilkka Lehtoranta for completing the port to MorphOS.
- Thomas Fricke and Adrian Maleska for various graphics improving the appearance of MUIbase.
- Martin Merz for his Mason icons.
- Mats Granstrom for beta testing of MUIbase and for writing the tutorial.
- Henning Thilemann, Joseph Durchalet, André Schenk, Klaus Gessner, and Oliver Roberts for ideas and beta-testing.
- Jernej Simoncic for permission to use his Windows install script for The Gimp as basis for the install script for MUIbase for Windows.

## Author

MUIbase is developed by:

Steffen Gutmann  
3-3-15 Shirokane 301, Minato-ku,  
Tokyo 108-0024  
Japan

Email: [muibase@yahoo.com](mailto:muibase@yahoo.com)

# Function Index

## #

#define .....	71
#elif .....	72
#else .....	73
#endif .....	73
#if .....	72
#ifdef .....	72
#ifndef .....	72
#include .....	71
#undef .....	71

## \*

* .....	93
---------	----

## +

+ .....	91
---------	----

## -

- .....	92
---------	----

## /

/ .....	93
---------	----

## <

< .....	90
<* .....	90
<= .....	90
<=* .....	90
<> .....	90
<>* .....	90

## =

= .....	90
=* .....	90

## >

> .....	90
>* .....	90
>= .....	90
>=* .....	90

## 1

1+ .....	92
1- .....	92

## A

ABS .....	94
ADDMONTH .....	107
ADDYEAR .....	107
ADMINPASSWORD .....	131
AND .....	89
APPEND .....	109
APPLY .....	81
ASC .....	101
ASKBUTTON .....	114
ASKCHOICE .....	112
ASKCHOICESTR .....	113
ASKDIR .....	111
ASKFILE .....	111
ASKINT .....	111
ASKMULTI .....	114
ASKOPTIONS .....	113
ASKSTR .....	111
ATTRNAME .....	124

## C

CASE .....	81
CHANGES .....	130
CHR .....	101
CMP .....	91
CMP* .....	91
COMPLETE .....	134
COMPLETEADD .....	134
COMPLETMAX .....	133
CONCAT .....	100
CONCAT2 .....	100
COND .....	82
CONS .....	108
CONSP .....	86
COPYREC .....	123
COPYSTR .....	100

## D

DATE .....	88
DATEDMY .....	106
DATEP .....	86
DAY .....	106
DEFUN .....	77
DEFUN* .....	78
DEFVAR .....	78
DEFVAR* .....	79
DELETE .....	121
DELETE* .....	122
DELETEALL .....	122
DIRNAME .....	133
DIV .....	93
DO .....	83

DOLIST .....	83
DOTIMES .....	82

## E

EDIT .....	131
EDIT* .....	131
ERROR .....	85
EXIT .....	84
EXP .....	95

## F

FCLOSE .....	117
FEOF .....	118
FERROR .....	118
FFLUSH .....	120
FGETCHAR .....	119
FGETCHARS .....	119
FGETMEMO .....	120
FGETSTR .....	119
FILENAME .....	133
FILLMEMO .....	105
FIRST .....	109
FOPEN .....	116
FOR ALL .....	84
FORMATMEMO .....	105
FPRINTF .....	118
FPUTCHAR .....	120
FPUTMEMO .....	120
FPUTSTR .....	120
FSEEK .....	118
FTELL .....	119
FUNCALL .....	81

## G

GC .....	134
GETADMINMODE .....	130
GETDISABLED .....	144
GETFILTERACTIVE .....	126
GETFILTERSTR .....	127
GETISSORTED .....	123
GETLABELS .....	124
GETMATCHFILTER .....	122
GETORDERSTR .....	125
GETVIRTUALLISTACTIVE .....	129
GETWINDOWDISABLED .....	144
GETWINDOWOPEN .....	129

## H

HALT .....	85
------------	----

## I

IF .....	81
INDENTMEMO .....	106

INDEXBRK .....	97
INDEXBRK* .....	97
INDEXSTR .....	96
INDEXSTR* .....	97
INSMIDSTR .....	96
INT .....	87
INTP .....	86

## L

LAST .....	109
LEFTSTR .....	95
LEN .....	95
LENGTH .....	108
LET .....	79
LIKE .....	101
LINE .....	104
LINES .....	104
LIST .....	108
LISTP .....	86
LISTTOMEMO .....	105
LISTTOSTR .....	99
LOG .....	95
LOWER .....	101

## M

MAPFIRST .....	110
MAX .....	93
MAXLEN .....	124
MEMO .....	87
MEMOP .....	86
MEMOTOLIST .....	104
MESSAGE .....	133
MIDSTR .....	96
MIN .....	93
MOD .....	93
MONTH .....	106
MONTHDAYS .....	107

## N

NEW .....	121
NEW* .....	121
NEXT .....	84
NOT .....	90
NOW .....	108
NTH .....	109
NULL .....	86

## O

onAdminMode .....	139
onChange .....	139
onClose .....	138
onOpen .....	138
OR .....	90

**P**

POW.....	94
PRINT.....	117
PRINTF.....	117
PROG1.....	79
PROGN.....	79
PROJECTNAME.....	130
PUBSCREEN.....	135

**R**

RANDOM.....	94
REAL.....	88
REALP.....	86
RECNUM.....	123
RECORD.....	128
RECORDS.....	128
RECP.....	86
REMCARS.....	98
REORDER.....	126
REORDERALL.....	126
REPLACESTR.....	98
REPLACESTR*.....	98
REST.....	109
RETURN.....	85
REVERSE.....	109
RIGHTSTR.....	96
RINDEXBRK.....	97
RINDEXBRK*.....	98
RINDEXSTR.....	97
RINDEXSTR*.....	97
ROUND.....	94

**S**

SELECT.....	128
SETADMINMODE.....	131
SETCURSOR.....	129
SETDISABLED.....	144
SETFILTERACTIVE.....	127
SETFILTERSTR.....	127
SETISSORTED.....	123
SETLABELS.....	124
SETMATCHFILTER.....	122
SETMIDSTR.....	96
SETORDERSTR.....	125

SETQ.....	80
SETQ*.....	80
SETVIRTUALLISTACTIVE.....	130
SETWINDOWDISABLED.....	144
SETWINDOWOPEN.....	129
SHA1SUM.....	100
SORTLIST.....	110
SORTLISTGT.....	110
SPRINTF.....	102
SQRT.....	95
STAT.....	132
stdout.....	117
STR.....	86
STRP.....	86
STRTOLIST.....	99
SYSTEM.....	132
SYSTEM*.....	132

**T**

TABlename.....	125
TACKON.....	132
TIME.....	89
TIMEP.....	86
TODAY.....	108
TRIMSTR.....	98
TRUNC.....	94

**U**

UPPER.....	101
------------	-----

**V**

VIEW.....	131
VIEW*.....	132

**W**

WORD.....	99
WORDS.....	99

**Y**

YEAR.....	106
YEARDAYS.....	107

# Concept Index

## 1

1:1 relationships .....	23
1:n relationships .....	23

## A

Accessing record contents .....	74
Acknowledgments .....	151
Active object .....	38
Active table .....	38
Admin mode .....	30
Admin password .....	30
Advance on Enter .....	33
Amiga version .....	2
ARexx .....	145
ARexx command syntax .....	145
ARexx commit .....	150
ARexx compile .....	147
ARexx connect .....	148
ARexx connections .....	148
ARexx disconnect .....	148
ARexx eval .....	149
ARexx help .....	147
ARexx hide .....	147
ARexx info .....	147
ARexx port name .....	145
ARexx quit .....	146
ARexx return codes .....	146
ARexx rollback .....	150
ARexx show .....	147
ARexx transaction .....	150
Attribute functions .....	124
Attribute management .....	57
Attribute object editor .....	62
Attribute objects .....	26
Attribute trigger .....	141
Attribute types .....	19
Attribute types (table) .....	21
Attributes .....	19
Author .....	152

## B

Balance objects .....	27
BetterString .....	2
Boolean attributes .....	20
Boolean functions .....	89
Browsing records .....	40
Buttons .....	21

## C

Change filter .....	42
Change order .....	45

Changing attributes .....	59
Changing tables .....	57
Check data integrity .....	30
Choice attributes .....	20
Cleanup external program source .....	35
Clear project .....	28
Close project .....	31
Command syntax .....	77
Comparison function .....	140
Comparison functions .....	90
Compute enabled function .....	142
Compute list-view labels .....	143
Compute reference records .....	144
Confirm quit .....	34
Confirm save & reorg .....	35
Constants .....	76
Copying attributes .....	59
Copying MUIbase .....	1
Corrupted database .....	30
Creating attributes .....	57
Creating tables .....	56

## D

Data retrieval .....	51
Data types for programming .....	75
Date attributes .....	20
Date functions .....	106
Defining commands .....	77
Delete record .....	40
Delete trigger .....	140
Deleting attributes .....	60
Deleting tables .....	57
Disclaimer .....	1
Display field .....	61
Display management .....	60
Distribution .....	1
Donation .....	1
Double click trigger .....	143
Duplicate record .....	38

## E

Empty order .....	44
Entering Boolean values .....	39
Entering choice values .....	39
Entering date values .....	39
Entering integer values .....	39
Entering NIL value .....	40
Entering reference values .....	40
Entering time values .....	39
Export records .....	50
Export structure .....	69
External editor .....	32



External program source .....	70
External viewer .....	33
Extra buttons in Tab chain .....	33

## F

File format .....	28
File format for import and export .....	49
Filename attributes .....	19
Filename conventions .....	7
Filter .....	42
Filter examples .....	43
Filter expression .....	42
Font-name attributes .....	19
Formats .....	32
Forward/backward search .....	47
Function triggering .....	138
Functional parameters .....	136

## G

Group editor .....	68
Groups .....	27
GUI functions .....	129

## I

I/O functions .....	116
Icons .....	2
Image attributes .....	19
Image editor .....	67
Images .....	26
Import and Export .....	49
Import file example .....	49
Import records .....	50
Info .....	28
Initial record .....	18
Input requesting functions .....	111
Installing MUIbase on Amiga .....	6
Installing MUIbase on Linux .....	5
Installing MUIbase on Mac OS .....	5
Installing MUIbase on Windows .....	5
Integer attributes .....	19
Internal errors in database .....	30

## K

Kinds of programs .....	74
-------------------------	----

## L

Label editor .....	59
Linux version .....	2
Lisp syntax .....	73
List functions .....	108
List of obsolete functions .....	144

## M

Mac OS version .....	2
Mailing list .....	1
Main window .....	25
Many to many relationships .....	24
Masks .....	26
Mathematical functions .....	91
Memo attributes .....	21
Memo context menu .....	39
Memo functions .....	104
Memory consumption .....	22
MUI .....	2
MUI preferences .....	34

## N

n:m relationships .....	24
Name conventions for program symbols .....	74
New attribute .....	57
New project .....	28
New record .....	38
New table .....	56
New trigger .....	139
NList .....	2

## O

Obsolete functions .....	36, 144
onAdminMode .....	139
onChange .....	139
onClose .....	138
One to many relationships .....	23
One to one relationships .....	23
onOpen .....	138
Open project .....	29
Order .....	44

## P

Panel editor .....	62
Panels .....	26
Pre-defined constants .....	135
Pre-defined variables .....	135
Preferences .....	32
Preprocessing .....	71
Printing queries .....	52
Program control functions .....	79
Program debug information .....	36
Program editor .....	70
Program include directory .....	36
Program output file .....	36
Program source .....	35
Programming .....	70
Programming language .....	73
Programming virtual attributes .....	142
Project functions .....	130
Project settings .....	34
Projects .....	18

**Q**

Query editor .....	51
Query examples .....	53
Quitting MUIbase .....	7

**R**

Real attributes .....	20
Record delete requester .....	35
Record filter .....	42
Record functions .....	120
Record memory .....	34
Record-editing .....	38
Records .....	18
Reference attributes .....	21
Reference filter .....	43
Register group editor .....	68
Register groups .....	27
Relational operators .....	90
Relationships .....	23
Relative paths .....	35
Reorder all records .....	46
Reorganization .....	29

**S**

Save as default .....	37
Save project .....	29
Search for .....	47
Search pattern examples .....	48
Search requester .....	47
Select-from-where queries .....	51
Semantics of expressions .....	137
Sort trigger functions .....	36
Sorting attributes .....	60
Sorting tables .....	57
Space editor .....	67
Space objects .....	27
Starting MUIbase .....	7
String attributes .....	19

String functions .....	95
Structure editor .....	56
Swap records .....	30
System functions .....	131

**T**

Table functions .....	125
Table management .....	56
Tables .....	18
Tabular display .....	51
Text editor .....	67
Text objects .....	26
TextEditor .....	2
Third party material .....	1
Time attributes .....	20
Time functions .....	106
Tutorial .....	9
Type conversion functions .....	86
Type predicates .....	86
Type specific settings .....	58
Type specific settings for attribute objects .....	63
Type specifiers .....	136

**U**

Updating from a previous version .....	6
User interface .....	25
User mode .....	30
User settings .....	32

**V**

Virtual attributes .....	21
--------------------------	----

**W**

Why lisp? .....	73
Window editor .....	68
Windows .....	25
Windows version .....	2