Smile Reference Manual

Satimage-software
http://www.satimage-software.com

Copyright © 2003 Satimage (France)

May 1st, 2003

Contents

I Smile, the better script editor for AppleScript 11

1 About Smile Reference Manual 12

1.1 How Smile's documentation is organized 12

1.1.1 The text files included in the distribution 12

1.1.2 Smile's on-line help menu 12

1.1.3 The resources available on the Web 13

1.2 The scope of Smile Reference Manual 13

1.3 Conventions used in Smile Reference Manual 13

1.4 What you should read 14

1.5 Reference Manual Change History 14

2 An introduction to Smile 15

2.1 Overview 15

2.2 To get started quickly 15

2.3 Short features list 15

3 Installation 17

3.1 What you should install 17

3.2 Installing Smile from scratch 18

3.3 Installing a new release 18

3.3.1 Full install 18

3.3.2 Partial upgrade 18

3.4 Installing the Satimage osax 19

3.4.1 First install of the Satimage osax 19

3.4.2 Installing an upgrade of the Satimage osax 19

3.5 Installing additional components 19

3.6 Multiple users support 19

4 Entering the world of Smile: scripting and debuging in text windows 21
4.1 Smile, a different experience of scripting . . . 21
4.2 Scripting and debug in text windows . . . 21
4.2.1 Making a new text window . . . 21
4.2.2 Executing scripts in text windows . . . 21
4.2.3 Displaying the result of execution . . . 22
4.2.4 The scope of the variables — Smile's context . . . 22

5 Working with scripts in script windows 24
5.1 About script documents formats . . . 24
5.2 Making a new script window . . . 24
5.3 Opening a script document . . . 25
5.4 Working with a script window . . . 25
5.5 Editing an applet or a droplet . . . 26
5.6 Saving a script document . . . 26
5.7 Saving an applet or a droplet . . . 27
5.8 Saving a script without its source . . . 27

6 Editing text in text windows 28
6.1 About text documents formats . . . 28
6.1.1 Unicode support . . . 28
6.1.2 ISO-8859-1 support . . . 28
6.2 Making a new text window . . . 28
6.3 Opening a text document . . . 29
6.4 Closing a text window . . . 29
6.5 Saving a text window . . . 29
6.6 Using drag and drop in text windows . . . 29
6.7 Editing text . . . 30
6.8 Selection keyboard shortcuts — Mouse tricks . . . 31
6.9 Text searches . . . 31

7 Using the dictionaries 33
7.1 Searching a term's definition . . . 33
7.2 Opening the dictionary of an application — Opening the dictionary of a Scripting Addition . . . 33
7.3 Opening the dictionary of an application which is running — Opening the dictionary of a Scripting Addition which is installed . . . 34
7.4 Opening the dictionary of the target application of a window . . . 34
7.5 Opening AppleScript's dictionary . . . 34

8 Scripting faster with the "Balance" command **35**
8.1 Syntax pre-typing . . . 35
8.2 Parentheses balancing . . . 35
8.3 Wrappers balancing . . . 36
8.4 "Balance()" call to your script . . . 36

9 The Scripts menu **37**
9.1 How to use the "Scripts" menu . . . 37
9.2 Adding and removing menu items to/from the "Scripts" menu . . . 38
9.3 Displaying hierarchical menus in the "Scripts" menu . . . 38
9.4 Grouping items in the "Scripts" menu . . . 38
9.5 Using aliases in the "Scripts" menu . . . 39
9.6 Sorting the items of the "Scripts" menu . . . 39
9.7 Providing shortcuts to the items of the "Scripts" menu . . . 39

10 Connecting a window to an application — The "tell ..." feature **41**
10.1 When to connect a window to an application . . . 41
10.2 How to connect a window to an application . . . 41
10.3 Making scripts into "raw code" . . . 41
10.4 Targeting an application by script . . . 42
10.5 The context of a window connected to an application . . . 42
10.6 "Find definition" in a window connected to an application . . . 42
10.7 Known bugs . . . 42

11 Comfort and productivity **43**
11.1 The Worksheet . . . 43
11.2 Handling windows efficiently . . . 43
11.3 The "Recent files" menu . . . 44
11.4 The "Favorites" menu . . . 44
11.5 Preferences . . . 44
11.5.1 The "General" pane . . . 44
11.5.2 The "AppleScript" pane . . . 45
11.5.3 The "Windows" pane . . . 46
11.6 Programmer's tools . . . 46
11.7 Variables that are saved when you quit Smile . . . 47

II The AppleScript-based automation engine **49**

12 Advanced text editing **50**

12.1 Advanced text searches 50
12.1.1 The Enhanced Find panel 50
12.1.2 Searching in folders 50
12.1.3 Regular expressions 50
12.2 Comparing files 51
12.3 Text tools 51
12.3.1 Make an AppleScript string 51
12.3.2 ISO-Latin1 to Mac and Mac to ISO-Latin1 51
12.3.3 Open ISO-Latin1 51
12.3.4 Measure Text 52
12.3.5 Sort paragraphs 52

13 The scriptable text editor — The Text Suite 53
13.1 Specifying a text range in a window of Smile 53
13.2 whose, where and every 54
13.3 before and after 54
13.4 The properties of the text 54

14 The UTF-16 editor 55
14.1 Overview 55
14.2 Using the UTF-16 editor 55

15 Smile custom dialogs 56
15.1 Overview 56
15.2 Running a custom dialog 56
15.3 Running a custom dialog by script 56
15.4 The basics of custom dialogs 57
15.5 Creating your own custom dialogs 57
15.5.1 Making a new custom dialog 57
15.5.2 Populating a new custom dialog 57
15.6 Editing a custom dialog 58
15.6.1 The edit mode 58
15.6.2 Dialog editing features 59
15.6.3 The dialog editing tools 59
15.7 Scripting a custom dialog 60
15.7.1 The basic properties of the controls 60
15.7.2 Events received by the scripts 61
15.8 Making a custom dialog multi-lingual 64
15.8.1 What is localization? 64
15.8.2 How to localize a dialog 64

15.8.3 How to localize Smile . . . 66
15.8.4 How to localize "Localize" . . . 66
15.9 Making a custom dialog into a stand-alone application . . . 66
15.9.1 Why to make a custom dialog into a stand-alone application . . . 66
15.9.2 Why not to make a custom dialog into a stand-alone application . . . 66
15.9.3 The limits of a stand-alone application . . . 67
15.9.4 Making a stand-alone application . . . 67
15.10 Attaching a custom dialog to an object . . . 67

16 Scripting Smile — The basics 69
16.1 Overview . . . 69
16.2 Manipulating objects — The object model . . . 69
16.2.1 Accessing an object . . . 69
16.2.2 Making a new object by script . . . 70
16.3 Programming the objects — The object scripts . . . 70
16.3.1 Introduction to object scripts . . . 70
16.3.2 How to write object scripts . . . 71
16.3.3 How to send commands to an object script . . . 73
16.3.4 The object script, a better script object . . . 73
16.4 Opening a file by script . . . 74
16.5 Providing a GUI — The Smile dialogs . . . 74
16.6 Scheduling tasks . . . 74

17 Scripting Smile — Advanced features 75
17.1 Overview . . . 75
17.2 Making and editing scripts by script . . . 75
17.3 The Class scripts — Defining new classes . . . 76
17.3.1 An introduction to class scripts . . . 76
17.3.2 Creating custom classes . . . 76

18 About Smile's libraries 78
18.1 Overview . . . 78
18.2 Documentation about Smile's libraries . . . 78

19 General purpose library 80
19.1 Strings . . . 80
19.2 Lists and records . . . 83
19.3 Files and resources . . . 84
19.4 Scripts . . . 85
19.5 User interaction . . . 85

20 Mathematical library **88**
20.1 Functions 88
20.2 Lists and arrays of numbers 89

21 RS232 library **91**
21.1 Overview 91
21.2 Instructions of use 91

22 Digital I/O library **93**
22.1 Overview 93
22.2 Instructions of use 93

23 PDF library — The Graphic Kernel **94**
23.1 Overview 94
23.2 Producing a graphic in a window 94
23.2.1 The basics 94
23.2.2 The graphic window 95
23.2.3 The graphical objects 95
23.3 The basics of the PDF language 95
23.3.1 The paths 95
23.3.2 The Graphic State 96
23.4 The graphic commands 96
23.5 Producing PDF data 96
23.5.1 The basics 96
23.5.2 Producing a PDF file 97
23.5.3 Appending PDF to a PDF file 97
23.5.4 Setting the background or the foreground picture of a SmileLab plot 97
23.5.5 Displaying an animation in a graphic window 98
23.5.6 Displaying graphics in a picture view 98
23.5.7 Displaying animated graphics with picture views 98
23.5.8 Displaying graphics in a custom dialog 99
23.6 Additional information and examples 99
23.6.1 How Smile's PDF engine really works 99
23.6.2 Additional resources 100

24 Smile's folders **102**
24.1 The roles of Smile's folders 102
24.2 Where Smile locates its folders 103

25 If you are curious about Smile 105

25.1 The history of Smile . . . 105
25.2 The philosophy of Smile . . . 105
25.3 Why Smile is free . . . 105

III Appendices 107

A Satimage regular expressions 108

A.1 Overview . . . 108
A.2 Defining a search pattern . . . 108
A.2.1 Metacharacters and "escape" character . . . 109
A.2.2 Anchors . . . 109
A.2.3 Character classes . . . 109
A.2.4 Operators . . . 110
A.2.5 Flags . . . 111
A.3 Defining a replace pattern . . . 112

B Portability and raw codes 113

B.1 What portability is about . . . 113
B.2 Referring to applications by creator code . . . 113
B.3 Why to use raw codes . . . 114
B.4 How to get the raw codes . . . 114

C The components for custom dialogs 116

C.1 Push Button . . . 116
C.2 Static Text Box . . . 116
C.3 Editable Text Box . . . 117
C.4 Password Text Box . . . 117
C.5 Popup Menu Button . . . 118
C.6 Slider . . . 118
C.7 Little Arrows . . . 118
C.8 Radio Button . . . 118
C.9 Check Box . . . 118
C.10 Time Clock . . . 118
C.11 Date Clock . . . 119
C.12 Progress Indicator . . . 119
C.13 Chasing Arrows . . . 119
C.14 Visual Separator . . . 119
C.15 Disclosure Triangle . . . 119

C.16 PDF Holder . . . 119
C.17 Icon Control . . . 120
C.18 Image Well . . . 120
C.19 Bevel Button . . . 121
C.20 List Box . . . 121
C.21 Menu Group Box . . . 121
C.22 Group Box . . . 121
C.23 Tabs Holder . . . 121

D The dictionary of Smile 123
D.1 Smile . . . 123
D.2 Misc . . . 127
D.3 Satimage utilities . . . 128
D.4 Smile drawings Suite . . . 129
D.5 SmileLab Suite . . . 130

E The dictionary of the Satimage osax 135
E.1 Satimage text Additions . . . 135
E.2 Satimage files Additions . . . 137
E.3 Satimage utilities Suite . . . 138
E.4 Resource Suite . . . 139
E.5 Math . . . 140

F Built-in routines 145
F.1 Handlers which display text . . . 145
F.2 Handlers which sort lists . . . 146
F.3 Miscellaneous helpers . . . 146
F.4 Handlers which open files . . . 147
F.5 Handlers which help manipulating Smile objects . . . 147

G Reference of the PDF commands 149
G.1 Overview . . . 149
G.2 Graphic state . . . 149
G.2.1 Handling States . . . 149
G.2.2 Stroke and Fill Settings . . . 149
G.2.3 Applying transformations . . . 150
G.3 Paths . . . 150
G.3.1 Operations on paths . . . 150
G.3.2 Building paths . . . 150
G.4 Text . . . 151

G.4.1 Text styles . . . 151
G.4.2 Drawing Text . . . 152
G.5 2D geometry . . . 152

H GeomLib, a graphical library for 2D geometry 153
H.1 Text Utilities . . . 153
H.2 Marking . . . 154
H.2.1 Marking Angles . . . 154
H.2.2 Marking Points . . . 154
H.2.3 Arrows . . . 155
H.3 Basic Geometry . . . 155
H.4 Basic Geometric Figures . . . 155

Part I

Smile, the better script editor for AppleScript

Chapter 1

About Smile Reference Manual

1.1 How Smile's documentation is organized

The documentation about Smile comes in three forms: as text files included in the distribution, as a Help Viewer module available from Smile, and as resources available via the World Wide Web.

1.1.1 The text files included in the distribution

When you first download Smile, you find in the distribution two documentation files, which will open in TextEdit if double-clicked: the *Read Me* and the *Release notes* files.

- The *Read Me* file
  The *Read Me* file located in Smile's folder contains:
  - a summarized information about Smile
  - the basic installation procedure
  - release information that users should be aware of, even if already familiar with Smile, e.g. important changes
  - URL's for further downloads, for additional information, for feedback, for support, and for licensing information.
- The *Release Notes* file
  The *Release Notes* file provides a summary of the release information concerning the particular version downloaded and the previous versions. It does not include information about minor fixes or improvements, nor about features which concern only expert users.

1.1.2 Smile's on-line help menu

Smile's Help menu has several items: the main Smile help item, and several other items named the help files.

- On-line Smile help
  Selecting the Smile help menu will launch Smile's Help Viewer module. Smile help is where the user not yet familiar with Smile will find the basic information about Smile's fundamentals and about how to use Smile.
  - one chapter describes the basics: scripting in script windows and debugging in text windows

  - one chapter is devoted to the helpful features that Smile offers to scripters
  - quickly documented are the enhancements to AppleScript that Smile offers, and the use of Smile as a Graphical User Interface editor.

- On-line help files
  The Help menu displays the files in the *More stuff:Documentation* folder: you can add your own. These additional files provide a quick help about some advanced features of Smile.

1.1.3 The resources available on the Web

You will find still more information if you connect your web browser to Satimage-software's World Wide Web site at *www.satimage-software.com*:

- Smile's official home page
  URL: *http://www.satimage-software.com/en/softx.html#smile*
  Latest breaking news about Smile.

  Smile's official home page provides links to all resources about Smile.

- Smile Release Notes
  URL: *http://www.satimage-software.com/en/releasesmilex.html*
  Up-to-date exhaustive history of the full release notes, detailing all bug fixes and new features.

- Smile Reference Manual
  URL: *http://www.satimage-software.com/downloads/RefMan_en_may03.pdf*
  The exhaustive manual which contains all information regarding all the aspects of Smile.

1.2 The scope of Smile Reference Manual

Smile Reference Manual presents all aspects of Smile. This includes features that the minimal installation of Smile may not include and features that may require a specific registration. Section 3.1 explains where the various components of Smile may be found.

Smile Reference Manual addresses Smile for MacOS X. There is no such document about Smile for MacOS 8/9. Information given here may or may not apply to Smile for MacOS 8/9.

1.3 Conventions used in Smile Reference Manual

The present manual uses the following conventions:

- file paths use the UNIX convention: directories are separated with the slash / rather than with the colon *:* like in AppleScript.
  Example:
  Store libraries in the *Class Scripts/Context additions/* directory.

- file names, folder names and paths are printed in italics.
  Example:
  Library/Application Support/Smile/

- menus and boutons are printed using a sans serif font.
  Example:
  the Save item of the File menu

- examples of scripts are printed using the typewriter font.
  Example:
  `tell application "Finder" to get window 1`

- keys of the keyboard are printed in slanted style.
  Example:
  Press *apple-shift-C*

- if you are viewing Smile Reference Manual with Acrobat Reader, the hypertext links are printed in purple.
  Example:
  See chapter 4.

- if you are viewing Smile Reference Manual with Acrobat Reader, the links to web addresses are printed in dark blue .
  Example:
  Visit *AppleScript home page*.

The manual includes hypertext links. You may require *Acrobat* or *Acrobat Reader* in order to have the hypertext links working.

1.4 What you should read

Smile Reference Manual is divided into thematic chapters. Each chapter covers one given issue in as self-consistent a fashion as possible. Depending on your current personal knowledge, and on what kind of information you are seeking, you will want to read one or several particular sections or chapters.
In all cases it is best that you be familiar with the concept described in Chapter 4, the use of Text windows as an AppleScript Terminal.

1.5 Reference Manual Change History

- May 1st, 2003: first public release of the Smile Reference Manual. Version number: _en_may03. The _en_may03 version of the Smile Reference Manual is in English and is up to date with the 2.5.2 release of Smile for OS X. Partial revision by Charles Ross.

- May 4th, 2003: for convenience, uploaded two versions of the Reference Manual on the Internet: *RefMan_en_may03.pdf*, 223 pages, and *RefMan_en_may03xs.pdf*, same contents in 155 pages.

- May 6th, 2003: provided additional instructions about the `mark` keyword.

- May 7th, 2003: provided additional instructions about the Regular Expressions flags and about how to open the Worksheet. Page numbers in top of the pages, no longer in footer.

Chapter 2

An introduction to Smile

2.1 Overview

Smile is an environment which uses all the hidden power of AppleScript to offer an automation center for controlling your machine, in a spirit of comfort and efficiency.
To this effect, Smile includes a script editor for AppleScript that is more ambitious than Apple's *Script Editor*. In particular, Smile includes an editor of graphical User Interfaces.
Furthermore Smile offers various libraries including mathematical libraries, a pdf generation library, libraries for performing serial and digital I/O, and the SmileLab library, an environment for plotting numerical data interactively.

2.2 To get started quickly ...

Smile provides on line information that you will require to get started. If no particular problem occurs, you should be able to take the following steps:

- download the latest Smile package
- expand it
- open the *Read Me* file located at the first level of the package
- install Smile following the instructions provided in the *Read Me* file
- launch Smile
- select "Smile Help" in the Help menu

you will get sufficient help to get you started.

2.3 Short features list

As of its version 2.5.2, Smile includes the following features:

1. a script editor for AppleScript
2. a persistent shell environment for AppleScript
3. an editor and runtime environment for scripted custom Aqua graphic interfaces
4. a scriptable styled text editor
5. a Unicode editor supporting in-line input
6. a library for creating and manipulating pdf graphics programmatically
7. a library for math which includes commands for fast processing of large arrays of real numbers

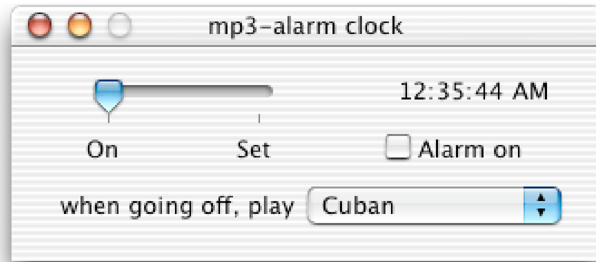

Figure 2.1: Using Smile's graphical user interface editor you design virtually any utility in few minutes.

8. a library for graphical (2D and 3D) representation of numerical data (pay features of SmileLab)

9. a library to drive RS232/422 serial ports via the Keyspan Twin USB/Serial Adapter

10. a library to handle digital I/O via the Delcom digital IO Development Board

11. a library with all-purpose commands which enrich the basic AppleScript

Chapter 3

Installation

Each release, upgrade or component of Smile comes as a compressed archive, and it should include a *Read Me* file which contains the instructions required to install it.

3.1 What you should install

The Smile download is a minimal install. Depending on what you intend to do with Smile you may need additional components. Here is what you have to install.

Smile's folder When you download the latest version of Smile, you get one folder, the Smile folder. Once you install that folder, Smile is fully functional: its double-clickable icon is inside that folder. However, not all of the features documented in the present manual will be available and you may have to install additional resources as described below.

Satimage osax The Satimage osax ("osax" means Scripting Addition, in other words an extension to the basic AppleScript) contains a number of commands that you may want to use from your scripts or applets: for details see section 18.1. The very basic operation of Smile does not require the Satimage osax.

Optional "user scripts" Smile supports a customizable menu, the Scripts menu — the menu with the script icon — that the Chapter 9 describes in all details. The *More User Scripts* folder located in the *Smile Extras* folder contains additional items that you can optionally install in the Scripts menu. To install a given command or a set of commands, copy the corresponding file or folder into the *User Scripts* folder.

Other additions Still more user scripts, and possibly other kinds of extensions to Smile, can be found on the web site of Satimage-software. Those are sometimes named the "Goodies" for Smile. A link to the download page for the Goodies is supplied on Smile's official home page *http://www.satimage-software.com/en/-softx.html#smile*

3.2 Installing Smile from scratch

Smile comes as a compressed self-mounting disk image, whose name is in the form *Smilexyz.dmg.gz* (e.g. *Smile252.dmg.gz*). The *.gz* suffix means that the file was compressed using *gzip*. Most probably, your Web browser will automatically expand the *.gz* file.

Please note: before you download *Smilexyz.dmg.gz*, make sure that an expanded file *Smilexyz.dmg* does not already exist in the location where the new file will expand.

Once the file expanded into the *Smilexyz.dmg* disk image, an installation volume *Smilexyz* may mount. If it does not, double-click the *Smilexyz.dmg* file to mount the installation volume. Double-click the icon of the installation volume: it contains the Smile folder. Do not attempt to run Smile, nor to open any file, from the installation volume.
Install Smile by copying the Smile folder from the installation volume into your *Applications* folder. The Smile folder contains the double-clickable icon of the Smile application.

Please note: before moving or renaming the Smile application, or any of the folders located at the same level, or any of its container folders and volume, quit Smile.

3.3 Installing a new release

Smile upgrades come either as a full install or as separate files (i.e., new versions of existing files and/or new files) which should be copied individually to their proper respective locations.

3.3.1 Full install

If you have deleted files from, added files to, or modified files in the *User Scripts* folder which is located in the *Applications* folder, you should at first make a copy of the *User Scripts* folder and place it outside of your current Smile folder.

Copy the new Smile folder into the *Applications* folder of your startup disk, then (if it applies) copy your personal scripts from the copy of your *User Scripts* folder into the new *User Scripts* folder.

We recommend that you move the older Smile application to the trash and that you empty the trash. When releasing a new version of Smile, the previous working version remains available for download — possibly without a link to it however. The URL for a given version *x.y.z* of Smile is:
http://www.satimage-software.com/downloads/-Smilexyz.dmg.gz

3.3.2 Partial upgrade

The location to install the files of the upgrades is described in the *Read Me* file which ships with the ugrade.

The locations where you may have to install such files are particular folders inside the Smile folder. See section 24 for more details regarding what roles those folders play, and where they are located.

It is required that you relaunch Smile after having installed files in the following cases:

- (you are installing a new version of the application itself)
- you install a file in the *Class Scripts* folder
- you install a file in the *Context additions* folder of the *Class Scripts* folder
- you install a file in the *Documentation* folder of the *More stuff* folder
- you install a file in the *Initialization* folder of the *More stuff* folder
- you install a new version of the Satimage osax.

3.4 Installing the Satimage osax

The Satimage scripting addition (or "Satimage osax") makes available to all scripts and applets, independently of Smile, a significant subset of Smile's features, described in section 18.1.
The Satimage osax is not required for the basic operation of Smile: you can choose to install it or not.
Note however that you must not run Smile with an outdated version of the Satimage osax installed.

3.4.1 First install of the Satimage osax

To install the Satimage osax, copy the *Satimage* file, either into the *ScriptingAdditions* folder of the *Library* folder located at the root of your startup disk, or into the *ScriptingAdditions* folder of the *Library* folder located in your user's space (*Home* directory). If a *ScriptingAdditions* folder does not exist at the said location, create one with that exact name (i.e., without a space).

3.4.2 Installing an upgrade of the Satimage osax

Refer to the section just above about where to install the Satimage osax.
To replace the *Satimage* file with a more recent version, quit all applications, then copy the new file in place of the old one.
If the system does not grant you permission to do so, proceed as follows: move the older *Satimage* file to the desktop, install the newer *Satimage* to where it belongs, then restart the computer, finally trash the older *Satimage* file.

3.5 Installing additional components

Installing additional components is described in the *Read Me* file which ships with the component.

3.6 Multiple users support

Smile fully supports multiple users. Setting it to work with multiple users requires specific steps when installing Smile. If you are the only user of your machine install Smile in your *Applications* folder as described above in this chapter, and you may skip the present section.

We give the instructions below for the administrator when needing Smile to work in a multiple users environment. More information regarding how Smile's folders work, and for providing a

more user account-specific behavior is given in section 24.

1. Smile creates a *Smile* folder in the *Application support* folder of the user's domain. That folder is called the user Smile folder.

2. when quitting, Smile saves to disk some variables, which include the user's preferences settings. These variables are stored in a file named *Globals*. Smile saves the *Globals* file in the user Smile folder.

3. when launching, Smile loads the preferences settings from the *Globals* file in the user Smile folder. If that file does not exist, Smile loads the preferences settings from the *SmileGlobals* file of the *Class Scripts* folder that is located in the Smile application's folder.

4. when quitting, Smile saves the contents of a particular text window, the *Worksheet*, as a file in the user Smile folder (see 11.1).

Chapter 4

Entering the world of Smile: scripting and debuging in text windows

4.1 Smile, a different experience of scripting

Smile is a script editor for AppleScript which is conceptually different from other script editors. Smile features text windows in which you can compile and run any piece of script on the fly. Pressing the *Enter* key (not *Carriage Return* like in Terminal) in a text window will compile and execute the current line (or text selection).

Text windows thus offer a different experience of scripting — interactive scripting. This unique feature will help you a great deal as you test and debug scripts.

Once you enter the world of Smile, you test and improve your script at the same time as you are writing it, in a text window. Once your script is sufficiently checked you will make it into a regular script or applet: this is where you will use Smile's script windows, those windows of Smile which mimic Script Editor's windows.

4.2 Scripting and debug in text windows

4.2.1 Making a new text window

Pull down the File menu, select New text.
You can set the respective keyboard shortcuts for New text and New script in the Preferences panel (see section 11.5).
You can change the default settings for the new text windows in the Preferences panel (see section 11.5.

4.2.2 Executing scripts in text windows

When you press the *Enter* key in a text window, the current line or the selected text is compiled, and executed if it is executable. The selected text may include declarations of properties and global variables, handlers, and executable lines.

This is what is called "running a script in/from a text window".

Note that, unlike in the script windows, the

lines are processed in their order in the window. Thus, in text windows, it is better to place declarations and handlers at the beginning of the scripts.

4.2.3 Displaying the result of execution

By default Smile appends the result of the execution to the Console window. You can have the result appended to the text window itself instead: pull down the Scripting menu and disable Output to console. You may want to do so, e.g., if you are using the window as an advanced calculator.

The Console is a special text window. When you close it, it remains open but invisible: the next time it opens, its contents are unchanged.

You can toggle the default behavior regarding where Smile prints the result in the Preferences panel (described in section 11.5).
The default behavior has no effect on the Worksheet: the Worksheet always appends the results of execution at the end of its own text (about the Worksheet, see section 11.1).
Except in the Console, results may be prefixed with the double hyphen (`--`). This is a setting that you can change using the Preferences panel (see section 11.5).

You may want to turn off displaying the result of execution. Such may be the case e.g. if the result of the script is too large. To instruct Smile not to print the result of a script, append a colon : to the script. Actually this prevents Smile from generating the string that it would otherwise have printed, resulting in a possibly faster execution.

Note that, due to some internal limitation of some versions of AppleScript, AppleScript may be unable to generate the string required to display the result of execution — this happens in cases where the string would be a very large one. If such is the case, Smile will display an error message `Could not display the result.` instead of displaying the result. This is not an error of execution: the script has run and returned normally.

4.2.4 The scope of the variables — Smile's context

A variable that you define in a script running from a text window is persistent. It will remain defined until you quit Smile. This unique feature makes it possible to debug scripts from text windows. For instance, you change the value of any variable by running at any moment from any text window a line such as below.

Example 1

```
set myVariable to [whatever]
```

You read the value of a variable by running simply:

Example 2

```
myVariable
```

If some lines appear to cause trouble, you can modify them, adjust the values of the variables which are affected, and re-execute only those lines.

The text windows share a common context, Smile's context. The variables and handlers that you define by compiling them in a text window augment this context and are available

to any other text window.

When you launch a script from Smile, it runs in Smile's context. Though, lines within a `tell application` wrapper that targets another application do not run in Smile's context: within such a wrapper you have to encapsulate lines with `tell me ... end tell` in order to have them run in Smile's context.

The AppleScript expression `every variable of context` returns — as a list of records — the list of Smile's context's variables and of their values. To get only the names of the variables, use `name of every variable of context`. You can also get the names of the handlers available to Smile's context with `name of every handler of context`.

Variables that are created (and that get accessed) using the `my` prefix are saved when the user quits Smile. Such variables are called "permanent" variables. They are available (still, using the `my` prefix) next time the user launches Smile. Example:

Example 3

```
set my toDoList to {"call Vlad", "lunch
w Donald", "prepare talk f/Congress"}
```

Next time the user launches Smile — even if the machine was shut down in between — `my toDoList` is still available.

Permanent variables are described in more detail in section 11.7.

Smile's context includes a set of routines which are available to your scripts, and that you may want to use: those are described in section F.

Note: a window which is connected to an application (see section 10 about the `tell` ... feature) has its own context, that it does not share with any other window. The context of such a window is its `context` property. To access a variable `myVar` owned by such a window `myWind` from any script, specify:

Example 4

```
myVar of (get context of myWind)
```

Chapter 5

Working with scripts in script windows

Smile opens compiled scripts, applets and droplets in script windows, which have a colored background. Smile's script windows work much like Script Editor's windows. This section presents how Smile's script windows work.

5.1 About script documents formats

Mac OS X supports two formats for storing scripts as files: a file can store a script either "in its resource fork" (the file's name has no extension; it is compatible with pre-OS X systems), or "in its data fork" (the file's name has the *.scpt* extension; older editors cannot open it).

Smile supports both formats. See section 5.6 about how to create files of both formats.

5.2 Making a new script window

To create a new script window, select New script in the File menu. Script windows have a colored background in order to differentiate them from the text windows, which are white. You can set the color of the script windows in the Preferences dialog panel (section 11.5).

You can set the respective keyboard shortcuts for New script and New text in the Preferences panel (see section 11.5).
You can change the default settings for the new script windows in the Preferences panel (see section 11.5.

The Handlers menu visible in the upper bar of the window displays the list of the handlers and script objects (scripts encapsulated within a `script ... end script` wrapper) in the window. Selecting an item in the Handlers menu will bring the said handler or script object into view. Option-clicking the Handlers menu displays the items in alphabetic order.
The Handlers menu can also display additional comments. A line starting with the double hyphen followed by the `mark` keyword generates an entry in the Handlers menu. The entry displays the fraction of the line following `mark` (31 characters maximum including possible formatting characters as described below). Use

this feature to tag a long script. A `mark` followed by a hyphen yields a blank line in the menu:

Example 5

```
-- mark -
-- mark Initialization handlers
-- mark -
```

If you append `<B` to a `mark` line, then the Handlers menu will display the corresponding comment (the text after `mark`) in bold. You can also use `<U` (for underline) or `<I` (for italics).

Also you may want to start the comment with a space (you would leave two spaces after `mark`). This way:

- the tags are slightly indented in the menu
- if you click the pop-up menu with the *option* key down the tags will show on top of the menu.

To show/hide the Handlers menu in a script window, pull down the Scripting menu and select List handlers. Text windows also accept a Handlers menu.

5.3 Opening a script document

To open a script document in Smile, select Open in the File menu, browse so as to select the desired file, then click Open.
You can drag any icon from Finder to the Open dialog.
You can open any script document by dragging its icon onto Smile's icon.
If a script document displays Smile's script document icon, you can double-click the document's icon to open it with Smile.
The document will open in a new colored script window.

If the document was saved as run-only (i.e., without its source), a new script window opens, but it displays only an error message. You can nothing about that: the source of a script which was saved as run-only is not recoverable.
Saving a window obtained by opening a script saved as run-only may yield unpredictable results.

5.4 Working with a script window

When you work with a script window, you use the same menu items as if you were editing normal text (described in chapter 6) and you also use commands which are specific to scripts, and that we describe here.

The commands specific to scripts are the menu items of the Scripting menu:

- Run script will attempt to compile and run the whole script displayed in the active script window. The result of the execution is printed in a special text window named Console. Sometimes, the Console gets hidden by other windows: to make it the front window choose Console in the Window menu.

  To interrupt a script while it is running, press *apple-period* or the *esc* key.
- Check syntax will attempt to compile the script which is displayed in the active script window, and will display any compilation error. Pressing the *Enter* key has the same

effect as choosing Check syntax — except that the keyboard shortcut remains available even if you did not make any change to the script since its last compilation.

- Start recording (toggles to Stop recording) will record in the active script window, in the form of AppleScript script lines, the user's actions until Stop recording is finally selected.
  As of MacOS X 10.2.1, not all applications are recordable. Finder is not fully recordable.

- the tell ..., logout, Copy translate menu items are related to connecting windows to applications. You may want to use this feature to perform interactive scripting of a given application, and also if you hit portability issues of your scripts. See section 10 for instructions to connect a window to an application. See Appendix B for more information regarding the portability issues and the use of raw codes.

- Find definition retrieves, if available, the definition of the term which is currently selected. The term may be a verb (e.g. `copy`) or a class name (e.g. `window`). Smile searches first its own dictionary, then it searches for all definitions of the term which can be found in AppleScript's basic dictionary or in the dictionaries of the currently installed Scripting Additions. If the active window is connected to some application (see section 10), Smile searches the application's dictionary first instead of its own dictionary.
  If Smile finds the definition for the term, it opens the dictionary (-ies) where the term was found.

- Output to console. When the active window is a script window, Output to console is greyed out and enabled: script windows always print their result to the Console — such is not the case for text windows, as you have seen above.

- List handlers, hides and shows the Handlers menu of the active window. By default, script windows have a Handlers menu, text windows do not.

5.5 Editing an applet or a droplet

To open an applet or a droplet in Smile, use the Open item of the File menu, or drag its icon on Smile's icon.

The `path to me` expression, when used in an applet, returns the path to the applet. You can debug scripts which use `path to me`: the `path to me` expression, when run from a script window in Smile, returns the path to the window's document (or applet).

5.6 Saving a script document

To save a script, use the Save or the Save as ... menu items of the File menu. If you are saving an unsaved script window, or if you are using Save as ..., a Format menu in the Save dialog box lets you choose one out of several options. If you choose to save the script as a regular script Document (the default option), then you may still choose to save it in either of the two formats supported (see section 5.1):

- to save the script as a file compatible with

pre-X systems, do not supply an extension to its name

- to save the script as a data-only file only for OS X, supply the *.scpt* extension to the new file name.

5.7 Saving an applet or a droplet

Applets and droplets are stand-alone applications that AppleScript makes out of one script. To save a script as an applet, choose Application in the Format menu of the Save Navigation dialog.
When double-clicked, an applet launches the execution of its `run` (or unnamed) handler, then it quits. To have the applet remain open after having executed its `run` handler (which may be empty), so that another script or application can communicate with it, choose Stay-open application in the Format menu.
If the script includes a `open` handler, Smile saves it as a droplet instead of an applet. A droplet launches its `open` handler when the user drags a file or a folder or several of them on its icon.

5.8 Saving a script without its source

The three last items of the Format menu of the Save dialog box are for saving the script as run-only, that is, without retaining the source of the script. After you save with this option, you can no longer edit nor view the script: proceed with care and keep an editable copy.

Chapter 6

Editing text in text windows

Smile includes a styled text editor. Smile's text editing features work also in the script windows.

Smile's styled text editor is fully scriptable: you can script and automatize the operations that this chapter describes. Chapter 13 describes text editing by script.

6.1 About text documents formats

6.1.1 Unicode support

Smile's text windows do not support multilingual in-line input. Smile's text windows can display Unicode imported by dragging or by pasting, or when the result of a script is a Unicode string, and they store Unicode reliably when you save the window to disk.

To make and to read Unicode text, use Smile's Unicode text editor windows (see 12.3).

6.1.2 ISO-8859-1 support

By default, Smile saves text as Macintosh-encoded. You can save a text window using the ISO-Latin 1 (ISO-8859-1) encoding: the text window being the active window, select Save as in the File menu. In the Save panel, pull down the Format menu, select Plain text, ISO-8859-1, then click Save. Plain text means that any text style will be suppressed.
If you use the standard Open menu to open an existing ISO-8859-1 encoded file, the window will display unreadable characters. To open an existing ISO-8859-1 encoded file, use the Open ISO-Latin1 ... command provided to this effect. The Open ISO-Latin1 ... command is part of Smile's text tools: see 12.3.

6.2 Making a new text window

To create a new text window, select New text in the File menu. Text windows have a white background in order to differentiate them from the script windows, which are colored.

You can set the keyboard shortcuts for New script and New text in the Preferences panel (see section 11.5).
The default settings for the new text window can be changed in the Preferences panel (see section 11.5).

6.3 Opening a text document

To open a text document in Smile, select Open in the File menu, browse until you select the desired file, then click Open.
You can drop any icon from Finder into the Open dialog.
You can open any text document by dragging its icon on Smile's icon.
To open a text document which displays Smile's text document icon, you can double-click the document's icon.
The document will open in a new white text window.

To display the contents of any file as text in a new text window, select Open data fork ... in the Scripts menu (the menu with a parchment icon), browse to the desired file, then click Open.
If the file size is large, opening its data fork may take a long time. If you press the *Escape* key while waiting, Smile will stop reading the file and will display what was read so far.

6.4 Closing a text window

Select Close in the File menu.

To quickly dismiss a window without saving its contents, use the Close without saving command of the Scripts menu (the menu with a parchment icon), shortcut *apple-shift-W*. If used accidentally, this shortcut can cause data loss, thus you may want to un-install the concerned user script. You may need your administrator's clearance to do so. See Chapter 9 about how to disable items of the Scripts menu.

6.5 Saving a text window

To save a text document, use the Save or Save as ... menu items of the File menu. If you are saving an unsaved text window, or if you are using Save as ..., a Format menu in the Save dialog box lets you choose one of three options:

- Document creates a Macintosh-encoded document containing styled text. The file will store the window's settings, and it will have an icon of a Smile text document. The file has the "TEXT" file type and a resource fork.

- Plain text creates a Macintosh-encoded document containing raw monostyled text. The file will not store any settings of the window, but it will have an icon of a Smile text document. The file has the "TEXT" file type and no resource fork.

- Plain text, ISO-8859-1 creates a ISO-8859-1-encoded document containing raw (monostyled) text. The file will not store any settings of the window, and it will not have an icon. The file has no file type and no resource fork.

Included in those window's settings which get saved only if you choose the Document option in the Format menu is the window's object script if it owns one. See section 16.3.1 about using object scripts.

6.6 Using drag and drop in text windows

Smile's text windows implement fully drag and drop.

- you can drag text from one location to another inside a given text window
- you can duplicate text from one location to another inside a given text window, by dragging it while holding the *option* key down.
- you can drag text from one window to another. The text is not deleted from the source window.
- you can drag text from one window of Smile to the desktop or to any Finder's window: this will create a text clipping file.
- you can drag the icon of any file into a window of Smile: this will insert the full path of the file at the drop location as a string, unless the file is a text clipping: dragging a text clipping insert its contents at the drop location.
- you can drag the icon of any file into a window of Smile while holding the *shift* key down: this will insert the file's POSIX path at the drop location.
- you can drag the icon of any file into a window of Smile while holding the *apple* key down: this will insert the file's AppleScript reference at the drop location.
- you can drag the icon of any script file into a window of Smile while holding the *apple* and *ctrl* keys down: this will insert the script's source as a string at the drop location.
- you can drag the icon from the title bar of a document window of Smile. Dragging it to the desktop or to any Finder's window will create a copy of the file. You can drop that icon into a Navigation Services dialog, or to other applications which may use it as a file reference: for instance you can drag the icon of an *.html* file to your browser's icon in the Dock.

6.7 Editing text

Smile's Text menu offers the menu items that are common for editing styled text:

- Font, Size, Style and Color
- Line width ... prompts you to enter a new value for the width of the text in pixels. Smile considers this value only if the Fit to window menu item is disabled.
- Fit to window when checked, the width of the text adjusts automatically to the size of the window: lines wrap automatically.
- Tab width prompts you to enter a new value for the size of the tabulation, in pixels. Use a small value such as 32 to indent a script. Use a larger value such as 200 to display a table.
- Shift right and Shift left change the indentation level of the selected lines.

You can set the default values for the settings above and also for the window size in the Preferences panel (described in section 11.5).

You will find also more commands in the Scripts menu (the menu with a parchment icon):

- Duplicate duplicates the selected text. Same as the sequence Copy Paste Paste but preserves the contents of the clipboard.
- Copy style and Paste style

- Iso-Latin-1->Mac and Mac->Iso-Latin-1 convert text between the Windows and Macintosh ASCII encodings. These menu items work on the selected text or on the whole text if no text is selected.

- the Text section of the Scripts menu offers additional tools for working on text. Those are are presented below, in section 12.3 of the chapter devoted to advanced text editing.

6.8 Selection keyboard shortcuts — Mouse tricks

You can move the insertion point and rapidly select text with the selection keyboard shortcuts. These shortcuts belong to the Mac OS, most applications support them. Here are the rules:

- use the arrows to move the insertion point one character left/right or one line up/down

- use the arrows while holding down the *alt* key to move the insertion point one word left/right or one page up/down

- use the arrows while holding down the *apple* key to move the insertion point to the beginning/end of the current line/document. Note that, with the left and right arrows, the insertion point moves to the corresponding end of the current line, not the end of the current paragraph. In other terms, the left and right arrows move the cursor horizontally only.

- holding down the *shift* key while using any of the combinations above selects text according to the same rules, instead of moving the insertion point.

Some mouse manipulations, which are available in almost any text editing software, are not known to all users. We briefly describe them here:

Selecting words Double-clicking selects one word. Double-clicking then dragging (sometimes called one click and a half) selects a range of words.

Selecting paragraphs Triple-clicking selects one paragraph (the block of text included between two Carriage Return characters). Triple-clicking then dragging (sometimes called two clicks and a half) selects a range of paragraphs.

6.9 Text searches

To perform text searches and replacements, use the Find dialog. You can also perform some searches without opening the dialog, using the menu items of the Edit menu:

- Find opens (or brings into view) the Find dialog

- Find again finds the next occurrence of the current search string in the active window

- Enter selection sets the text currently selected as the search string, but does not perform the search

- Find selection sets the text currently selected as the search string, and searches its next occurrence in the active window

Note regarding the keyboard shortcuts: since it is standard under OS X to have *apple-H* hide the application, the original *apple-H* keyboard shortcut for Find selection was changed into

apple-shift-H. This shortcut, in turn, conflicts with a shortcut of the Programmer's tools — if you have installed the Programmer's tools.

Using Smile's Find dialog, you can perform searches in multiple files, and also use regular expressions (see 12.1.3). These features are described in section 12.1.
A more sophisticated Find dialog, the Enhanced Find dialog, is available as an addition to Smile (a separate download). The Enhanced Find dialog supports more options such as file renaming, background tasking and working on all open windows, and it includes more help about Regular expressions.

Chapter 7

Using the dictionaries

Any scriptable application and scripting addition has a dictionary, which presents its AppleScript terminology. Smile offers several ways of opening a given dictionary.

Smile opens dictionaries in text windows which display a Index pop-up menu in their toolbar. You can handle dictionary windows as standard text windows. The Index menu displays the list of the suites (bold, underlined), of the verbs (plain text), of the classes (italics). Selecting any of these items will scroll the window so as to make it visible.

7.1 Searching a term's definition

To display the definition of a given term, select the term, then select Find definition from the Scripting menu. The term may be a verb (e.g. `copy`) or a class name (e.g. `window`). Smile searches first its own dictionary, then it searches for all definitions of the term which can be found in AppleScript's basic dictionary or in the dictionaries of the currently installed Scripting Additions. If the active window is connected to some application (see section 10), Smile searches the application's dictionary first instead of its own dictionary.
If Smile finds the definition for the term, it opens the dictionary or dictionaries where the term was found, and it brings the relevant entry into view.

7.2 Opening the dictionary of an application — Opening the dictionary of a Scripting Addition

Select Other ... in the Open dictionary ... submenu of the File menu. The Open dialog box shows only those files which have a dictionary. Browse to select the desired file then click Open.

7.3 Opening the dictionary of an application which is running — Opening the dictionary of a Scripting Addition which is installed

The Open dictionary ... submenu of the File menu displays a list of items intended for quickly opening some dictionaries:

- AppleScript itself gives you access to the basic AppleScript dictionary
- Scripting additions, a sub-menu, displays all the scripting additions that are currently installed
- the scriptable applications which are currently running
- System Events, a background application which supplies several useful commands, particularly concerning files, folders and volumes.

Depending on various factors, the Open dictionary ... submenu of the File menu displays may be lengthy to build, slowing down your computer when you pull down the File menu. If you experience such a slow-down, disable the Open dictionary ... sub-menu in the Preferences panel (see 11.5).

7.4 Opening the dictionary of the target application of a window

If a window is connected to an application (see section 10 regarding that topic), then the menu in its toolbar provides a fast access to the application's dictionary.

7.5 Opening AppleScript's dictionary

You open AppleScript's own dictionary using the Open dictionary ... submenu of the File menu. Alternately, type some basic AppleScript class name such as `list`, select it and choose Find definition in the Scripting menu.

You will find in AppleScript's basic dictionary, in a short form, some of the information available in the official documentation of AppleScript. For instance, you will find there as a reminder, at the entry `list`, all the properties of lists, `length`, `reverse` and `rest`. You'll also find, e.g., all the operators that AppleScript supports, and also all the prepositions that you may use in your handler definitions.

Thus, this dictionary is very precious for avoiding syntax errors, and fully benefiting from the features offered by AppleScript. But be cautious! Not all the terms of that dictionary are supported by AppleScript or by the applications. Some of them may even be discontinued eventually.

Chapter 8

Scripting faster with the "Balance" command

Balance is a command available as a user script (see Chapter 9 for information about the user scripts). It is available through a keyboard shortcut, *apple-shift-E*. Balance performs several elementary functions for you.

8.1 Syntax pre-typing

If some text is selected, Balance will try to find it as a verb in one of the dictionaries: Balance applies the same rules as the Find definition menu item described in section 7.1.
If if finds the verb, Balance will write the whole set of parameters for the verb, required or optional. Variable names that Balance writes are intended only to suggest the type of variable required.

Example 6

```
display dialog
```

will expand into:

Example 7

```
display dialog MyString default answer
MyString buttons MyList default button
MyNumber with icon MyNumber with icon
MyStop giving up after MyInteger
```

8.2 Parentheses balancing

If no text was selected, or if the selected text was not found in any dictionary, then Balance will check that parentheses, brackets, braces and double-quotes are balanced. If some balancing error is detected, it will highlight the character where the error was detected. If some pair or pairs are not closed, Balance will close them. In which case, it will highlight the characters that it added, so that you can check its action. For example, Balance will complement:

```
[...]  & (item i of {"Ga", "Bu", "Zo",
"Meu
```

into:

```
[...]  & (item i of {"Ga", "Bu", "Zo",
"Meu"})
```

8.3 Wrappers balancing

If there was no problem with parentheses, Balance will try closing a wrapping structure. For instance, Balance will expand:

```
if (someCondition)
```

into:

```
if (someCondition) then
    | (<-- insertion point here)
else

end if
```

Balance handles the following AppleScript wrappers: `tell`, `on [handler]`, `to [handler]`, `repeat`, `if`, `script`, `try`, `with timeout`, all `considering`'s, and all `ignoring`'s. If you trigger Balance after a `tell`, it will present a dialog to let you choose any of the scriptable applications available — and it will complete the `tell` line for you.

8.4 “Balance()” call to your script

If the object script of the active window contains a `Balance()` handler, then Balance will perform none of the above: it will call that `Balance()` handler (object scripts are described in section 16.3.1). This mechanism allows for providing a keyboard shortcut to some (supposedly repetitive) script specific to one given window.
The `Balance()` handler must not require any parameter. In the handler — as in any object script — use the `container` invisible property of the script to refer to the owner of the script.

Example 8

```
on Balance()
    change "http://" into "" in
selection of container
end Balance
```

Chapter 9

The Scripts menu

The rightmost menu of Smile's menu bar (left to the Windows and Help menus) is an icon of a script. This menu is called the Scripts menu of Smile — not to be confused with the Scripting menu, its left neighbor.

(If Smile's menu bar does not display such a menu, probably Smile could not locate the associated *User Scripts* folder. Regarding installing Smile's folders, including the *User Scripts* folder, see Chapter 24.)

The Scripts menu provides a way to activate a script — as well as opening documents — by menu. Smile's mechanism for the Scripts menu supports hierarchical menus, separation lines, and keyboard shortcuts.

Smile's Scripts menu ships with a number of menu items. You can make your personal scripts available in this menu. Additional items, e.g. sets of specialized scripts, are available in other places. For instance, you will find more items to populate the Scripts menu in the *Smile Extras/More User Scripts* folder located in the Smile folder. You can also download more additions for Smile, including additional user scripts, from the links provided in *Smile's home page*.

This section describes how to use the Scripts menu, and how to customize it, including how to add your own "user scripts".

9.1 How to use the "Scripts" menu

Selecting one of the items of the Scripts menu launches the corresponding user script, or opens the corresponding document, respectively.

In addition to scripts and documents, you can put aliases to scriptable applications and aliases to scripting additions in the *User Scripts*. Selecting the corresponding item of the Scripts menu will open the dictionary of the application (if it has one) or of the scripting addition, respectively.

To open one of the user scripts — for instance, to view or to edit the script — select its name in the Scripts menu while holding down the *alt* key. (Of course, you can use the Open ... item of the File menu as well).
You may want to use the scripts that come

included in the *User Scripts* as samples.

While a user script is running, a script can get its path as the `user script file` property of Smile. While no user script is running, the `user script file` property returns the path to the user script which ran last.

9.2 Adding and removing menu items to/from the "Scripts" menu

The Scripts menu displays the names of the scripts and the documents which are located in the *User Scripts* folder. The scripts located in the *User Scripts* folder are called user scripts. The documents that can be opened by the Scripts menu are those documents that Smile can open: the text documents, Smile's custom dialogs, the applets and droplets (the Scripts menu does not launch applets, it opens their scripts), QuickTime movies and most kinds of sound files.

To add an item to the Scripts menu, copy the corresponding file to the *User Scripts* folder.
To remove an item from the Scripts menu, remove the corresponding file from the *User Scripts* folder.
When you change the contents of the *User Scripts* folder, you do not need to quit Smile. Pull down the Scripts menu once to update the menu.

To add or remove a sub-menu, add or remove the corresponding folder to or from the *User Scripts* folder (see below).

You can also use the Favorites sub-menu of the Scripts to add items dynamically to the `Scripts` menu. Use Add window 1 to add the document opened in the front window to the menu. Use Add Finder's selection to add the file or the folder that is currently selected in Finder, or several of them, to the menu.
It is handy, when you work for a while on a given folder, to make it available in the Favorites menu.

9.3 Displaying hierarchical menus in the "Scripts" menu

The Scripts menu displays any folder located in the *User Scripts* folder as a sub-menu; like for the *User Scripts* folder itself, the sub-menu displays the list of the files located in that folder.
The same is true for folders nested at any level in such folders, making possible to design hierarchical menus of any depth.

The space character "␣" as the first character for the name of a folder located in the *User Scripts* folder has a special meaning, that the following section describes.

9.4 Grouping items in the "Scripts" menu

Folders that are located in the *User Scripts* folder and whose name begins with the space character "␣" are treated differently: their contents are displayed directly in the Scripts menu (not as a sub-menu), as a group, i.e. separated from the previous and from the subsequent items by a separating (blank) menu item.

Any file whose name begins with the hyphen –, or an alias to such a file, will get displayed in the Scripts menu as a separating menu item. You cannot open it via the Scripts menu.

9.5 Using aliases in the “Scripts” menu

The *User Scripts* folder may contain aliases to files. The Scripts menu displays the name of the original file, not the name of the alias file.

As mentioned above, copying the alias of an application or the alias of a scripting addition into the *User Scripts* provides a convenient way of opening its dictionary.

The *User Scripts* folder may also contain aliases to folders. If you use personal user scripts, it may be handy (in particular with respect to installation of future versions of Smile) to store them in a folder in your user domain, and to make an alias to this folder in the *User Scripts* folder.

9.6 Sorting the items of the “Scripts” menu

By default, the items of the Scripts menu are alphabetically sorted. You can use the two following mechanisms to alter the order of the menu items:

- The entry for an alias file displays the name of the original file, but it uses the name of the alias file for the alphabetical sort. By providing specific names to alias files, you can have the Scripts menu's items sorted arbitrarily. Here is an example:

  You have created two user scripts: *Open database* and *Close database*. Their order in the menu is: Close database, Open database. Now you want them to display in the reverse order, and in the bottom of the Scripts menu. Move those files to another location. Create aliases to them in the *User Scripts* folder. Name the aliases, respectively, *zzaOpen database* and *zzbClose database*. The *zz* prefix will ensure that they be the last items, and the *a* and *b* prefixes will provide for the order of the items.

- The exact same mechanism can be used on folders, in order to customize the order of the sub-menus of the Scripts menu.

- You can create separating menu items at arbitrary places, by naming adequately aliases to files whose names would begin with –.

- The groups of items (the contents of those folders whose name starts with the space character “␣”) can also be re-arranged by naming adequately the concerned folders. For instance, naming a folder ␣0*First Items* is a good way of having its contents displayed at the top of the Scripts menu.

9.7 Providing shortcuts to the items of the “Scripts” menu

If the name of a user script ends with the slash character / followed by another character, for instance *Send mail/@*, the final character, (@ in the example) can be used in combination with the *apple* key as the keyboard shortcut for selecting the menu.

The shortcut is case-sensitive: if the final character of the name of the script is a low case character, for instance *Just a Test /j*, the shortcut combination is simply *apple + 'key'*, for instance *apple-J*. If the final character is an uppercase character, for instance *Just a Test /J*, the shortcut combination is *apple-shift + 'key'*, for instance *apple-shift-J*.

If this shortcut is a number *0 ... 9*, you must use the numeric keypad to activate it. (Mac OS Classic only: if the settings of your system allow for it, you can alternately use the function keys to activate such shortcuts: *F1* for *apple-1*, ... , *F9* for *apple-9* and *F10* for *apple-0*.)

By script, you can assign any keyboard shortcut to any menu item. You can even assign a keyboard shortcut that will not require the *apple* key. The section about the `menu` and `menu item` classes in the Chapter 19 describes that feature.

Chapter 10

Connecting a window to an application — The "tell ..." feature

10.1 When to connect a window to an application

You can connect a text or a script window to an application provided the application is currently running. Script lines that you execute in such connected windows are sent directly to the target application.
Furthermore, linking a window to an application is how you can generate the translation of a piece of script into AppleScript's internal "raw codes". Raw codes can serve various purposes, in particular the portability of your scripts.

10.2 How to connect a window to an application

To connect the front window to an application, pull down the Scripting menu then select the application in the tell ... menu.
To connect a window to an application which is running but does not show in the tell ... menu use a script as described in section 10.4.

When a window is connected to an application, a menu with the target application's name is created in the window's toolbar. This menu provides a quick access to the target application's dictionary and to the `Copy translate` command.

To terminate the connection of a window, select Logout in the Scripting menu.

10.3 Making scripts into "raw code"

Windows connected to an application support a special Copy command named Copy translate. Copy translate is a menu item both of the Scripting menu and of the menu which appears in the tool bar. Copy translate works only on a piece of text which makes a compilable piece of script.
Copying a piece of script with Copy translate copies to the clipboard the raw codes instead of the original text. Raw codes are like a precompiled version of the script, that AppleScript can understand even outside the relevant `tell ... end tell` wrapper. Raw codes are helpful

for making scripts more portable, and possibly for other purposes. Portability is discussed in Appendix B.

10.4 Targeting an application by script

If thetell ... menu does not display the application that you want to target, or if you have disabled the tell ... menu in the Preferences panel (see 11.5), or if you want to automatize that process, you can proceed by script; you must set the «class targ» property of the window to `application theAppName` where `theAppName` contains the name of the application. `theAppName` may include the `.app` extension. You can run from the window itself:

Example 9

```
set «class targ» of window 1 to
application "URL Access Scripting"
```

To logout, set the «class targ» property to the empty string `""`.

10.5 The context of a window connected to an application

When you run a piece of script in a window which is connected to an application, the script runs in a private context whose parent is the target application. To access a variable of that context from another context (e.g. from a regular text window), you must use the `context` property of the window:

Example 10

```
set theRemoteVar to theVar of (get
context of window 2)
```

10.6 "Find definition" in a window connected to an application

If the front window is connected to an application, the Find definition command described in section 5.4 will search first the dictionary of the target application, before searching the Scripting Additions and AppleScript's basic dictionary.

10.7 Known bugs

According to Apple's documentation, running a script from a window connected to an application should be equivalent to running the same script encapsulated within the proper `tell ... end tell` wrapper. Unfortunately, this is not entirely true. There are two known issues:

- the properties of the application itself do not require a suffix to be recognized as such when the `tell ... end tell` is present. In a window connected to an application, the `its` prefix is sometimes required in front of the properties of the application. The `its` prefix is allowed also within the `tell ... end tell` wrapper.
- some terms of some applications which have too much of the flavor of a basic AppleScript term, for instance the `item` and `alias` terms of the Finder, sometimes yield confusion, resulting in anomalous errors.

Chapter 11

Comfort and productivity

11.1 The Worksheet

Smile offers a special text window, the *Worksheet*. By default Smile opens the Worksheet when it launches. When you quit Smile, Smile automatically the Worksheet as a text file in the *Library/Application Support/Smile/* directory of the user domain.
By default, the Worksheet returns the result of scripts at the end of its own window — not in the Console.
You can choose not to use the Worksheet in the Preferences panel, see 11.5.
To open or re-open the Worksheet, open the Preferences panel, General tab and use the Open the Worksheet check box: disable it if it comes enabled, then enable it.

11.2 Handling windows efficiently

Smile offers several helpers for those users who have to handle lots of windows and/or who are concerned with productivity. For optimal efficiency, all those commands can be activated via a keyboard shortcut.

Close without saving (*apple-shift-W*) closes the active window, skipping the Save changes before closing alert. This is what is called deleting the window, and can be performed by script using the `delete` verb.
You may want to use Close without saving to execute rapidly a line of script, or a short script, yet keeping the screen neat. Make a new text window, type the script, press the *Enter* key, then delete the window.
(As for French keyboards, *apple-D*, for "**D**on't save", is the shortcut for the Ne pas enregistrer button of the alert.)

Toggle windows (shortcut keyboard-dependent, often *apple-alt-T*) brings the second window to front while the previously active window becomes the second frontmost.
You may want to use Toggle windows when you work with two windows simultaneously. For instance, Toggle windows, in conjunction with Compare, Copy and Paste, allows you to synchronize two files without using the mouse.

Send to back (*apple-shift-B*) sends the active window behind all others.
You may want to use Send to back when you have to simultaneously handle several projects, each of them possibly involving

several windows. Typing *apple-shift-B* the required number of times will have the effect of switching projects.
Send to back is also a solution when you accidentally activate a window that would rather remain in the background.

Duplicate (*apple-D*) if no text is selected, duplicates the active window. (When some text is selected, Duplicate duplicates the selection).
Duplicating the active window may be useful if for some reason (e.g. an accidental deletion of text) you have to synchronize the currently edited version of some document with its last saved version. Duplicate the active window, then close the original document without saving the changes, re-open it, then use Compare to synchronize the windows.

11.3 The "Recent files" menu

The Recent files sub-menu of the File menu displays the list of the most recently opened documents. Selecting any item of the list opens the corresponding document — if it is still available.
The maximum number of files proposed by the Recent files sub-menu is stored in the application's «class MReF» property, that you can change.

Example 11

```
set my «class MReF» to 40
```

By default the maximum number of recent files is 20. Changes are effective only once the user has pulled down the Recent files menu.

11.4 The "Favorites" menu

The Favorites sub-menu of the Scripts menu provides a way of opening files at a click.
The Favorites sub-menu displays folders as sub-menus which list the contents of the folder.
Selecting the name of an application in the Favorites sub-menu opens the dictionary of the application.
By default, the Favorites sub-menu offers three menu items:

- Add Finder's selection appends to the Favorites menu item the file(s) and/or folder(s) that are selected in Finder
- Add window 1 appends to the Favorites menu item the document that is open in the front window
- Clear menu clears the menu

11.5 Preferences

To open the Preferences panel, pull down the Smile menu and select Preferences. The Preferences panel allows you to change the settings described below. Changes take effect immediately.

11.5.1 The "General" pane

Cmd-N is the keyboard shortcut for: By default, the two first items of the File menu, namely New script and New text are given the *apple-N* and *apple-shift-N* combinations as their keyboard shortcuts, respectively. You can invert this choice and choose *apple-N* for the New text.

When running a script in a text window: This section is where you choose where and how the result of the execution of a script from a text window (see 4.2) will be displayed.

Note that the In text windows, prefix result with '–' option does not concern the Console: the results printed to the Console are never prefixed.

When starting up You can choose to have the Worksheet open as Smile launches.

You can choose to have Smile re-open the files which were open last time you quit Smile.

Use the including dialogs option with caution. Some dialogs may require specific initialization steps that are normally performed by a script, and re-opening them directly may result in unexpected behavior.

At the moment when Smile launches, holding down the *shift* key prevents Smile from re-opening the files. Even if Smile has started re-opening the files, pressing the *shift* key will cancel the re-opening phase, and the initialization of Smile will resume normally. The *shift* key has a temporary effect: the preference is not changed.

Remember insertion point of text windows By default, when Smile re-opens a Smile text document, the text insertion point assumes the position it had last time the document was saved. Though this behavior is generally helpful, you may in some cases want to change it. For instance, if you have lots of files to compare using the Compare feature of Smile (see section 12.2), you will prefer that they open with the insertion point at the beginning of the text.

When opening a file: This section is to protect you against the effect of possible malicious Smile documents. Documents created with Smile — e.g. a text document or a custom dialog — may optionally store a script (the object's script). If a document has such a script, and if that script contains a `prepare` handler, Smile will execute the `prepare` handler when it opens the file: many objects of Smile get initialized properly by a suitable `prepare` handler. You may choose as a preference to have Smile notify you before actually opening the document if a `prepare` handler is present. You may choose to restrict the protection to only those files which belong to a given folder (for instance, your downloads folder or the folder of your mail attachments).

Once the file is open, it is your responsibility to check that nothing in the script be harmful to you. Any of the routines of an object's script can trigger a line capable of an adverse effect, such as `click in`, `close`, `save` or `delete` — even if the script does not include a `prepare` handler.

11.5.2 The "AppleScript" pane

The AppleScript pane is where you set the AppleScript formats. In a compiled script the different categories of terms such as comments, variables names, literal strings, assume different text styles: the AppleScript formats.

The Edit format for menu lets you choose one of the eight categories of terms. To change the style for a given category, choose the category in the Edit format for menu, then use the Font,

Size, Style and Color sub-menus of the Edit menu.

You can also choose one of the built-in preset formats with the Preset formats menu. Note that, whichever the preset format you may have selected, the menu always reads Preset formats.

The AppleScript pane is also where you can customize two particular menu items of Smile: the tell ... item of the Scripting menu, and the Open dictionary ... item of the File menu.
By default, the latter displays a list of all the applications which are currently running on the machine and which have an AppleScript dictionary. Some applications may make this list take longer to build when they are running. If this happens, and if it is an aggravation for you, you can disable the option of building the list. This setting is effective next time you launch Smile.
By default, the tell ... menu displays a list of all the applications which are able to handle at least basic scripting calls such as `quit` or `open`. For reasons similar to those described just above, the tell ... menu can prove slow to display. The corresponding checkbox of the AppleScript pane allows for disabling the tell ... menu.
Once the tell ... menu is disabled, you must use a script to connect a window to an application, as described in section 10.4.

11.5.3 The "Windows" pane

The Windows pane is where you change the default settings for new windows (both script windows and text windows).
Smile use the default settings whenever:

- when you make a new text window using the New text command
- when you open a new script window, either by using the New script command, or by opening a script document or an object's script.

The set default size button sets the default size (height and width) for new windows to the size of the window just behind the Preferences panel.

You can also choose one of four colors for the background of the script windows, in order to adapt to your personal preferences and your monitor's performances. Script windows which already exist when you change the setting keep their background color: only newly created windows will assume the new background color.

11.6 Programmer's tools

The programmer's tools can be found can be installed by copying the *Smile Extras/More User Scripts/Programmer's Tools/* into your *User Scripts* folder. The programmer's tools offer a set of conversions. The conversions work only on the selected text — if no text is selected, they just beep. The Undo command cancels the latest conversion. Once the conversion is performed, the result remains selected, so you can directly perform another conversion if required.

Decimal to Hexa (*apple-shift-H*) will convert a positive integer written in decimal format, e.g. `10000000000`, into its hexadecimal representation, e.g. `02 540B E400`. The hexadecimal digits are separated by spaces into 4-characters blocks.
The input string may contain spaces, e.g. `10 000 000 000`. You can convert integers

up to `281474976710656` (hexa: `FFFF FFFF FFFF`).

Hexa to Decimal (*apple-shift-X*) will convert a number written in hexadecimal format into its decimal representation.
The input string may contain spaces and *CR* (carriage return) charagers, e.g. `02 540B E400`. You can convert numbers up to `FFFF FFFF FFFF`.

Hexa to String (*apple-shift-G*) will convert a string of hexadecimal digits, e.g. `68 65 6C 6C 6F 21`, into characters, e.g. `hello!`, using AppleScript's ASCII table.
The input string must contain an even number of hexadecimal digits, and it may include spaces and carriage returns. You can convert strings of any length.

String to Hexa (*apple-shift-J*) will convert a string of any length, e.g. `ascr`, into hexadecimal digits, e.g. `61736372`, using AppleScript's ASCII table.
You can convert strings of any length.

11.7 Variables that are saved when you quit Smile

Smile supports "permanent" variables, variables whose contents are saved when you quit Smile, and are available next time you launch Smile. A permanent variable can store any kind of quantity, just like any AppleScript variable.

To define such a permanent variable, create a variable with the `my` prefix:

Example 12
```
set my varName to whateverQuantity
```

To access such a permanent variable, always use the `my` prefix and the `get` explicit verb:

Example 13
```
set my globalCount to 1 + (get my
globalCount)
```

When Smile quits, it stores the permanent variables in the *Globals* file located in the *Library/Application Support/Smile* directory of the user domain. That *Globals* file stores both the permanent variables that you have created and those that Smile uses internally to store your personal preferences.

Deleting or moving that *Globals* file while Smile is running has no effect: Smile will store all the permanent variables into a new *Globals* file when you quit.
To reset all preferences to their built-in default values, delete (or move) the *Globals* file while Smile is not running. This will suppress the permanent variables that you may have defined — you may want to do so if some permanent variable has reached a very large size.

The AppleScript expression `every variable of globals` returns — formatted into a record — all the permanent variables together with their values. To get only the names of the variables, use `name of every variable of globals`. You can also get the names of the permanent handlers with `name of every handler of globals`. By default, there are no permanent handlers, but you can define your own:

Example 14
```
on ShowThem()
    dd("see, it works!")
```

```
end ShowThem

set my DemoHandler to ShowThem

 -- months later ...
my DemoHandler() -- will display the
successful dialog
```

Part II

The AppleScript-based automation engine

Chapter 12

Advanced text editing

12.1 Advanced text searches

12.1.1 The Enhanced Find panel

Smile's standard Find panel includes a lot of features and is enough for most users.
A Find panel offering more features such as searching in all the open windows and renaming files, the Enhanced Find panel, is available as a separate download from Satimage-software's site. The present manual covers some features that are enabled only if you have installed the Enhanced Find dialog.

12.1.2 Searching in folders

Using the Find panel, you can select a folder and search the files it contains for the search string. Enable the Multifile search check box, then use the Choose folder button to select a target folder. Finally click Find to launch the search.

The Find panel will search all text files and compiled script files (including applets) which are located inside the selected folder or inside any nested folder in the selected folder.

A Result window displays the status of the search in its toolbar, and the results in its main frame. Each hit results in three lines: one line which starts with `show`, one line which quotes the line where the string was found, and one empty line. To display the hit, put the insertion point in the line that starts with `show` then press the *Enter* key.

If you have installed the Enhanced Find panel, you get the following differences and enhancements:

- you can perform Replace, Replace all, optionally then save, on the files of the folder
- you can interrupt the search with the Stop button
- there is no Multifile search check box. Instead, you select all files in ... in the Find target: menu.
- the dialog stores the recently searched folders in a menu.

12.1.3 Regular expressions

The Find panel offers a "Regexp" (for "regular expressions") option. If you enable "Regexp", the string that you enter as the search string

defines a regular expression pattern: most characters keep their literal meaning, but some of them become metacharacters, which allow for defining advanced text searches. Instructions and references about regular expressions can be found in Appendix A.

The Enhanced Find panel offers a self-explanatory menu to help entering regular expression patterns.

Some of the more advanced features of the regular expressions can only be used from a script, as described in section 18.1 about the Satimage osax — they are not available from the Find panel. Such is the case for the `regexpflag` parameter, whose use is described in the Appendix about Regular Expressions, section A.2.5.

12.2 Comparing files

The Compare command of the Edit menu compares the text of the two frontmost windows, starting at the current location of the insertion point in each window, and then selects the first block in each window which makes a difference — or the location where a block is missing.

Once you have hit a difference, you may choose or not to synchronize the windows by Copy then Paste, and then selecting again Compare (or hitting the keyboard shortcut *apple-K*) will jump to the next difference.

For optimal speed, Smile offers in the Scripts menu a Toggle windows menu item which swaps the two frontmost windows. You should be able to rapidly synchronize two files by using the keyboard shortcuts for Compare, Copy, Paste and Toggle windows.

12.3 Text tools

Smile provides specialized tools to perform some operations on text, and to manipulate the various kinds of text files that you may need to work with.

12.3.1 Make an AppleScript string

Make an AppleScript string adds double quotes around the text currently selected, performing all necessary changes to make it suitable as a string for AppleScript. Selecting Make an AppleScript string with the *ctrl* key depressed has the inverse effect.

12.3.2 ISO-Latin1 to Mac and Mac to ISO-Latin1

ISO-Latin1->Mac and Mac->ISO-Latin1 perform the encoding conversions of the text selected in the frontmost window. If no text is selected, the conversions apply to the whole text.
ISO-Latin1 is the encoding used traditionally by Microsoft. For instance, if you read the data fork of a Microsoft Word (*.doc* extension) document, performing the ISO-Latin1->Mac conversion will fix all the unreadable punctuation and accented characters.

12.3.3 Open ISO-Latin1 ...

Open ISO-Latin1 ... opens a text document using the ISO-Latin1 encoding. For instance, if you receive a text file (*.txt* extension) which was saved by a Microsoft application, you will probably have to use Open ISO-Latin1 ... to

have it display correctly.

Once you have opened a document by using the Open ISO-Latin1 ... command, the document keeps the ISO-Latin1 encoding when you save it. Note that the data saved in the file does not include the specification of the encoding: next time you open the document, you must still use the Open ISO-Latin1 ... command.

12.3.4 Measure Text

Measure Text opens a dialog box which continuously displays some information regarding the text contained in the frontmost text or script window. The Show: menu lets you choose to display one of the following pieces of information:

Selection the boundaries of the current selection

Characters the number of characters in the text

Words the number of words in the text. The word count is computed using AppleScript's definition for words, which depends on various settings of your system.

Paragraphs the number of paragraphs in the text. A paragraph is a block of characters which ends with a CR (ASCII 13) character.

If you enable the Selection Only option, then the counts of characters, words and paragraphs restrict to the text currently selected in the front window.

12.3.5 Sort paragraphs

Sort paragraphs alphabetically sorts the paragraphs of the selected text in the frontmost text or script window. If no text is selected, the sort applies to the whole text.
To have paragraphs sorted in the reverse alphabetical order, press the *ctrl* key while selecting the Sort paragraphs command.

Chapter 13

The scriptable text editor — The Text Suite

Smile sort of supports a double Text Suite: text editing commands work both on AppleScript variables that contain text and on references to some text chunk in some window of Smile.

13.1 Specifying a text range in a window of Smile

To refer to a range of text in a Smile text document, you can use the following keywords :

- `character`
- `word`
- `paragraph`
- `text` or `string`
- `selection`

By default, these descriptors return lists of strings, for instance `paragraphs 1 thru 3 of window 1` will return a list of three strings.

If you `get` such an expression `as list` Smile returns a list of two integers, the boundaries of the text described by the expression.

If you `get` such an expression `as text` Smile returns one string, the text described by the expression.

If you `get` such an expression `as styled text` Smile returns the text described by the expression as styled text: you can store the text and its style altogether in one variable and thus copy styled text from one window to another.

Note that it is more consistent to use the `boundaries` property to get the offsets of the ends of a range of text rather than coercing it to `list` since the coercion may lead to unexpected behaviors in some complex situations.

Examples 15

```
first character of paragraph after
selection of window 1
words 1 thru 2 of window 1 as text
words 1 thru 2 of window 1 as list
boundaries of words 1 thru 2 of window
1
set theText to words 1 thru 2 of window
1 as styled text
```

The `selection` property of a text window returns a list of two integers, the boundaries of the text selected.

If you `get` the selection `as text` (resp. `as styled text`) Smile returns one string, the text selected (resp. the text selected as styled text).

13.2 whose, where and every

When you use the `whose` or `where` clause, AppleScript builds a list of descriptors. You can get the whole list with the `every` keyword, or any of its items by using the `first` or `last` keywords, or by specifying any integer index.

Examples 16

```
paragraph 3 of window 1 where it begins
with "If"
-- the third paragraph of the front
window that begins with "If"
first word of (first paragraph of
window 1 where it contains "first")
(every paragraph of window 1 where it
contains "every") as list
```

13.3 before and after

You can specify a range of text by its location with respect to another one with `before` and `after`. You can also use `before` and `after` to specify a location for inserting new text.

Examples 17

```
paragraph after words 1 thru 2 of
window 1
set after text of window 1 to return
set before paragraph 2 of window 2 to
"Name:  "
```

13.4 The properties of the text

The properties of text are provided in the dictionary under the entry for `Class text`. Both `get` and `set` — when it makes sense — can be applied to any property of such text description(s), or to their contents, or to the beginning or the end of their content(s).

Examples 18

```
get color of every paragraph of window
1 where it contains "get"
get paragraph index of first paragraph
of window 1 whose text size is 12
set end of (first paragraph of window 1
whose (length > 30)) to " :)"
set first word of window 1 where it is
"Smile" to "Smile's"
set first paragraph of window 1 whose
color is purple to "About it"
```

Chapter 14

The UTF-16 editor

14.1 Overview

Smile includes an experimental Unicode editor. In its current version, the Unicode editor handles only UTF-16, not UTF-8. Not all features work normally, including some basic features. Though, Smile's Unicode editor saves documents reliably.

14.2 Using the UTF-16 editor

Selecting the New UTF-16 and Open UTF-16 ... items of the Unicode sub-menu of the Scripts menu will open a Unicode editor window.
In Unicode editor windows you can use the in-line multilingual input system.

In the standard text windows of Smile, you cannot use the in-line multilingual input system, but you can paste a Unicode string from a Unicode editor window into a standard text window of Smile. It will get saved properly when you save the window.

In an Unicode editor window like in any standard text window of Smile (see 4 if you are not familiar with that) you can run or compile a line or a block of lines by pressing the *Enter* key.

There are some instances where Smile uses UTF-16 files and edits them in Unicode editor windows: in particular when you localize a Smile dialog you edit translation dictionaries in Unicode editor windows.

If the window that gets opened with the Open UTF-16 command does not display correctly the characters, you may fix the problem by copy-pasting its contents to, then back from, a standard text window.

Chapter 15

Smile custom dialogs

Smile includes an editor and a runtime environment for scripted custom Aqua dialogs, commonly called Smile dialogs. The dialogs that you build can use all the features of Smile — including, e.g., handling text windows.

15.1 Overview

Custom dialog is the name given to a specific kind of window which supports the standard Aqua controls, and that can be stored to disk as a "dialog file". In the Finder, dialog files display an icon of a Lego with Smile's saucer on one side.
For instance, Smile's Preferences and Find panels are custom dialogs, that you can use as samples.
The *Smile Extras/Custom Dialogs/Examples* folder contains basic examples of custom dialogs.

15.2 Running a custom dialog

Running an existing custom dialog consists in opening the dialog file in Smile.
To open a dialog, double-click its icon, drag it on the icon of Smile, or use the Open ... item of Smile's File menu.
You can store a custom dialog in the *User Scripts* folder: the Scripts menu will display its name, and you can open it by selecting it in the menu. (See Chapter 9 for information about the Scripts menu)

15.3 Running a custom dialog by script

To open a dialog file by script, instruct Smile to make a new object out of the dialog file:

Example 19

```
make new basic object with properties
{path name:  thePath}
```

where `thePath` should contain the path to the dialog file.
This line returns a reference to the newly created dialog.
Each time this line is executed, a new copy of the dialog window is created. To prevent creation of multiple copies even if the handler gets called several times use the high-level handler `DoOpen`:

Example 20

```
DoOpen (thePath)
```

15.4 The basics of custom dialogs

Dialogs are defined (in Smile's dictionary) as objects of class `dialog`. The `dialog` class inherits from the virtual `window` class. A `dialog` may have elements, the objects of class `dialog item`, the Aqua "controls" of the dialog.

Smile's dialog windows support most Aqua control objects such as buttons, menus and text fields. Some of the controls may themselves contain other controls. You will find all details about each kind of control in Appendix C.

The dialog window and each control may have an attached script. The scripts handle the user's actions. Usually it is enough to provide a script to the dialog itself only: user's actions directed to a given control (e.g., a click in a button) trigger calls to the script of the container of the control — not to the script of the control itself. The container of the control may be the dialog itself, or a control such as a Group Box.

15.5 Creating your own custom dialogs

Creating a new dialog window requires three steps. First, you open a new empty dialog window. Then you create new controls in the new dialog. Finally, you write the scripts.

15.5.1 Making a new custom dialog

Select New Dialog in the Dialogs section of the Scripts menu. A new empty dialog window opens, in "edit mode". Another dialog window, the Dialog components palette, opens at the same time, also in edit mode.

As with any document, use the Save item of the File menu to save the dialog to disk.

To create a new empty dialog window by script, use the standard `make` verb, as in:

Example 21

```
make new dialog with properties
{name:"untitled", bounds:{50, 50, 300,
250}}
set mode of result to true
```

15.5.2 Populating a new custom dialog

The Dialog components palette window contains controls that you can install in your new dialog window, either with drag-and-drop or by using Copy and Paste. The Dialog components palette window can be re-opened at any moment by selecting Palette in the Dialogs section of the Scripts menu.

Some kinds of controls (the Group Boxes for instance) can be a container for other controls. When you drag a control in such a container control, the new control gets added as an element of that parent control — not as an element of the dialog. To view the hierarchy to which a given control belongs, use the contextual menu (in edit mode). The upper items

of the contextual menu of a control display the hierarchy of its containers.

The clipboard commands Cut Copy and Paste work on dialog items, provided the dialog is in edit mode; the Undo command is ineffective. The Paste command pastes into the selection:

- if no dialog item is selected, Paste pastes the dialog item(s) contained in the clipboard into the dialog, at the first level.
- if a dialog item is selected and if it can accept dialog items (a Group Box for instance), Paste pastes the contents of the clipboard into that dialog item.
- if a dialog item is selected and if it does not accept dialog items (a Editable Text Box, for instance) nothing happens.

When you Paste a control, it assumes the same location (its `bounds` property) in its new container as it had in its original container. Thus, if you copy a control from a large dialog into a smaller one, you may be pasting it out of view. If this happens, use the *arrow* keys to bring it into view, or use the Clear item of the Edit menu to remove the new item and then use drag and drop instead of Copy-Paste.

You can use the Copy style and Paste style user scripts. These commands work both on text windows and on dialog items, provided they are selected and the dialog is in edit mode.

Select all (Edit menu) selects all the elements of the selected item(s):

- if no dialog item is selected, Select all selects all the dialog item(s) contained in the dialog at the first level.
- if one or several dialog item(s) are selected Select all selects all the elements that they contain — possibly, no item at all.

15.6 Editing a custom dialog

15.6.1 The edit mode

Dialogs have a boolean property called `mode`. When `mode` is set to `false`, the dialog is in running mode, the mode for normal use: this is the default mode for the dialog when you open it.

To toggle a dialog's `mode` property, use the Edit Mode menu item of the Edit menu.

Setting its `mode` property to `true` switches the dialog into "edit mode". The edit mode is the mode which allows to make changes to the dialog.
In edit mode you can observe the following:

- the lower-right corner of the dialog window displays a resize icon
- you can select dialog item(s) using the mouse and *shift-click*
- a selected dialog item draws a black rectangular frame and displays its index number in its lower-right corner.

By using a script, you can set any property of the dialog and of the controls at any moment, even while the dialog is not in edit mode.

Example 22

```
set contained data of dialog item 3 of
dialog "Nuke codes" to ""
```

15.6.2 Dialog editing features

When a dialog is in edit mode, you can perform the following operations:

Resize the dialog: drag the window's bottom right corner.

Resize a control: drag the control's bottom right corner.

Move the dialog: move the dialog window to the location where you want it to open later: the location is saved in the dialog file.

Move a control: drag it with the mouse. To move a control by exactly one pixel, select it and press the keyboard arrows. To move a control by exactly 20 pixels at a time, press the keyboard arrows while holding the *Shift* key down.

Add a control: use drag-and-drop or Copy-Paste to add any control in the Palette (or in any dialog) to your dialog. Both dialogs must be in edit mode.

Remove a control: drop it to the Trash on the Desktop, or use the Cut or Clear items of the Edit menu.

Cut, Copy, Paste, Select All: these commands apply to the selected control(s), if any. When you Paste a control, it gets created as a new element — the last element — in the control that is currently selected, or in the dialog itself if no control is selected. Cut and Paste can therefore be used to renumber items.

Edit the common settings of a control: double-clicking a control will open its "settings dialog". In the settings dialog of a control you can perform the following:

- set its name or, for the Static Text Box, its contents.
- set its font to one of the two system fonts Large size and Small size. (Under a standard US English install of Jaguar, the system fonts are Lucida Grande 13 and 11, respectively).
- enable or disable the use script option. use script specifies whether Smile will send a `click in` event when the user performs an action in the control. Note that the `click in` event is sent, not to that control which is the user's action's target, but to that control's container.

Edit the specific settings of a control: clicking a control with the *ctrl* key down displays its contextual menu. The contextual menu is where you find the settings which are specific to each particular kind of control.

Edit the script of the dialog window: *option-apple-click* on the dialog window.

Edit the script of a control: *option-apple-click* on the control.

View all the settings of a control: drag-and-drop it onto any text window.

Get the reference of a control: drag-and-drop it onto any text window while holding the *apple* key down.

15.6.3 The dialog editing tools

The following tools may help you easily achieve a handsome dialog:

Arrange items: use the Arrange items dialog to align controls, copy bounds from one control

to another, and set the font of controls to one of the system fonts.

Aligning controls does not always align the texts they contain. To align the base line of texts, use the selection tool (the “marching ants” rectangle) as a visual ruler.

Object eXpert: use the Object eXpert dialog to view and edit all the properties of the dialog or of a control (actually, Object eXpert can be used on any object in Smile). The Object eXpert shows more properties than dragging a control onto a text window. If you check more properties, the dialog shows still more properties, including properties that may require some caution.

The Object eXpert’s input field supports any AppleScript expression.

In the select some object group of Object eXpert, you can enter any valid AppleScript descriptor for the object you want to edit, such as `window "Data"`. This feature is broken in the original 2.5.2 distribution, so if you are using the original 2.5.2 distribution you have to download a working version from *Smile’s home page*.

Rescale dialog: a command which resizes the whole dialog. You provide any real number (or an expression evaluating to a real number) as the scaling factor: `1.0` means no change.

15.7 Scripting a custom dialog

15.7.1 The basic properties of the controls

Although there are different kinds of controls, Smile’s dictionary defines only one `dialog item` class for all controls. The `dialog item` class inherits from the `basic object` class. All `dialog items` have the following properties:

`control kind:` an integer which specifies the kind of the control. You can change the value for the `control kind` property, provided you use one of the values given in Appendix C. The change is effective only next time the control will be created, for instance by Cut-Paste or by closing then re-opening the dialog.

`dialog:` a reference to the dialog window where the control is installed.

`container:` a reference to the object which contains the control: this may be the dialog window itself, or some control installed in the dialog window.

`enabled:` a boolean which specifies whether the control is active. For instance, an Editable Text Box whose `enabled` property is set to `false` is visible but its contents are locked. A Check Box whose `enabled` property is set to `false` is visible but grayed out.

`visible:` a boolean which specifies whether the control is visible. For instance you use `visible` to show and hide the Chasing Arrows control.

`contained data:` the contents of the control. The meaning of the `contained data` property depends on the kind of the control. Essentially, `contained data` contains the data that is required to handle the user’s action, e.g. the string that it contains for a Editable Text Box. See Appendix C for all details.

(`contents` is a deprecated alternate for `contained data`. Do not use it.)

`font:` a record describing the font and style information for the text displayed by the control. Only those controls that display some text own a `font` property. The `font` property is a record, with all fields optional. The fields are:

- `font:` an integer that specifies one of the default System fonts, or a string that specifies the name of a font. -1 is the large System font, -2 is the small System font. 1 et 2 are for OS9-like System fonts: 1 is for Geneva 9 and 2 is for New York 9.
- `text size:` an integer that specifies the size of the font. If a font name was specified as the `font` field and no `text size` is provided, then the controls assume the dialog's `text size`.
- `color:` a list of 3 integer values (Red, Green, Blue) in the range 0..65535. Only the Static Text Box and the Editable Text Box support text coloring.
- `style:` a record containing one or both of the two labels `on styles` and `off styles`. The values for those labels are lists which should contain one or several of the following constants: `bold, italic, underline, outline, shadow, condensed, expanded`, e.g. {`on styles: {bold, italic`}}.

`call script:` a boolean which specifies whether the user's actions directed to the control will trigger a `click in` event. When the control's `call script` property is set to `true` Smile sends `click in` to the script of the control's container (the dialog itself, or another control) when the user acts on the control. The use script check box of the control's settings dialog reflects its `call script` property.

`want idle:` a boolean which tells whether the dialog window will receive the `idle` event periodically, or not. Using the `idle` event is a way of performing background tasking in Smile.

Some controls have additional properties. Use the contextual menu in edit mode to set the specific properties of a control. Use the drag-and-drop from the dialog window to a text window to view the list of a control's properties. Use its `whole` property to get still more properties, including its script.

15.7.2 Events received by the scripts

The scripts of a dialog are where you define how the dialog will handle the events that it is to receive. The dialog will receive events created by the user, as well as events that Smile sends automatically in several circumstances.
All handlers are optional: you provide only those which make sense for your application. All parameters are required in a handler's declaration: the first line of the handler should include all the parameters as specified in the dictionary (or below), including those that you do not intend to use.

`prepare` When Smile creates an object, it sends a `prepare` event to the object's script before bringing it in view.
You handle `prepare` with a handler such as:

Example 23

```
on prepare theTarget
```

```
    -- do whatever required
end prepare
```

where `theTarget` is a reference to the owner of the script, the object which is being created. You will probably want to perform any required initialization in that handler. You may provide a `prepare` handler to the dialog, and also to any of its dialog items. `prepare` is sent, first to the dialog, then to the dialog items in the order of the increasing `index` values.

While its `prepare` handler is executing, the dialog is not visible yet, but it is already the frontmost window, `window 1`. The previously frontmost window is now `window 2`.

Your `prepare` handler may — e.g. if some checking has failed — delete the dialog which is being created. In other terms you can cancel the opening of the dialog in the `prepare` handler.

`store` When Smile saves an object, or when some script requests the value of an object's `whole` property — probably in order to save it in one form or another — Smile sends the `store` event to the object's script.
You handle `store` with a handler such as:

Example 24

```
on store theTarget
    -- do whatever required
end store
```

`theTarget` contains a reference to the object being saved, the owner of the script. The `store` event is intended for performing any necessary update or cleaning which could be required before saving the object.

`close` Whenever the user closes a dialog's window, Smile sends a `close` event to the dialog's script.
You handle `close` with a handler such as:

Example 25

```
on close theTarget saving whatever
    -- do whatever suitable
    continue close theTarget saving
whatever
end close
```

`theTarget` contains a reference to the dialog being closed, the owner of the script. The `continue` command is required to have Smile actually close the object.
Note that the `saving` parameter of the `close` event is required: your script must specify `saving no`, `saving yes` or `saving ask`.

`delete` Whenever Smile deletes an object (for instance, when the user closes a dialog), it sends a `delete` event to the object's script.
You handle `delete` with a handler such as:

Example 26

```
on delete theTarget
    -- do whatever suitable
    continue delete theTarget
end delete
```

`theTarget` contains a reference to the object being deleted, the owner of the script. The `continue` command is required to have Smile actually dispose of the object.

As for the 2.5.2 version of Smile, controls do not receive `delete`.

`click in` (the verb is `click in`, not `click`) As

Appendix C describes, most controls send `click in` to their container in response to the user's actions.
You handle `click in` with a handler such as:

Example 27

```
on click in theTarget item number
theIndex
    -- handle user's action
end click in
```

`theTarget` contains a reference to the container of the concerned control — the owner of the script.
`theIndex` contains the index of the control. To view the index of a control, select the control while the dialog is in edit mode.
`click in` does not provide a reference to the control that was addressed by the user. A reference to that control is `theTarget's dialog item theIndex`.

Keep in mind that the `click in` call is issued only if the `call script` property of the control is set to `true` (the use script option is activated in the control's settings dialog).

By default, Smile keeps silent the execution errors that occur in a `click in` handler. If you want to be notified of execution errors that might occur in a `click in` handler, encapsulate the body of your handler within a `try ... end try` wrapper.

`drop` When the user drops an item onto one of those controls which accept drag-and-drop (see Appendix C), the script of the control receives a `drop` event.

You handle `drop` with a handler such as:

Example 28

```
on drop theThing onto theTarget at
theLocation
    -- handle the drop event
end drop
```

`theThing` contains a reference to the dragged item
`theTarget` contains a reference to the control which is receiving the drag
`theLocation` contains the relative coordinates where the item was dropped (you can omit `at theLocation` in the handler's declaration).

The control's ≪`class flav`≫ property, a list of 4-character strings, specifies what kind of objects (what "flavors") the control accepts. Standard flavors include:

- `"hfs␣"`: a file reference (for exemple, an icon from Finder)
- `"long"`: an integer
- `"doub"`: a real number
- `"alis"`: an alias
- `"reco"`: a record
- `"TEXT"`: a string
- `"obj␣"`: a reference to an object of Smile

An exception is the List Box control, which receives `export` (see below) but not `drop` when the user performs drag-and-drop inside the list.

Like for `click in`, by default Smile keeps silent the execution errors that occur in a `drop` handler.

`export` When the user drags from one of those controls which accept a mouse drag (see Appendix C), the control's script receives an `export` event. The result returned by `export` is the data that will be carried by the drag-and-drop — and that will be passed to the `drop` handler if dropping occurs on a control of a custom dialog of Smile.
You handle `export` with a handler such as:

Example 29

```
on export theSource
    return someValue
end export
```

`theSource` contains a reference to the control where dragging started
`someValue` is what the control will export.

The control's ≪`class flav`≫ property, a list of 4-character strings, specifies what kind of objects (what "flavors") the control may export. See `drop` above for a list of standard flavors.
No event will be triggered if the quantity returned by the `export` handler fits none of the declared flavors.

Like for `click in`, by default Smile keeps silent the execution errors that occur in a `drop` handler.

`idle` When Smile is idle, it periodically sends the `idle` event to those dialog windows (i.e.: to their scripts) whose `want idle` property is set to `true`. Controls do not receive `idle` calls.

You handle `idle` with a handler such as:

Example 30

```
on idle theDialog
    -- do whatever suitable
    return 3
end idle
```

`idle` should return a number. This number specifies how much time, in seconds, Smile will wait before sending `idle` to the dialog window again.

15.8 Making a custom dialog multi-lingual

15.8.1 What is localization?

MacOS X implements a standard mechanism for making multi-lingual software. Localizing software consists in providing the resources necessary for this mechanism, so that as often as possible users manipulate strings in their own language.

Smile includes an interface in order to help you supply the localization resources, so that you can in a few steps localize any dialog into any language.

15.8.2 How to localize a dialog

Most of the dialogs provided in the standard distribution are localized in English and French, and you can use them as samples.

To localize your new dialog, proceed as follows.

Fill your dialog in 7-bit English Provide the contents of the static texts, of the menus, of the list items, and the names of the items as strings in English: you must use only the minimal character set (7-bit ASCII) and no double quotes. Those strings are keys: you will provide as many localized translations to them as necessary, including English. The localized translations of the keys can include any Unicode character such as punctuation or Cangies.
Finally, make sure to save your dialog to disk.

In the scripts of the dialog, use `localize` Your script may include strings that require to be localized. Here, too, you will use keys, and you `tell` the dialog to `localize` the key:

Example 31

```
tell theDialog to set
theLocalizedString to localize theKey
display dialog theLocalizedString
```

`localize` supports lists of strings:

Example 32

```
tell theDialog to set theButtons to
localize {"New drink", "Sandwich",
"Nothing thanks"}
display dialog "Shall I fix you
something?" buttons theButtons
```

Enter the localization dictionaries While your dialog is the active window, select Localize in the Dialogs section of the Scripts menu. This opens the Localize panel. Use the Localize panel to add and suppress languages (i.e., localization dictionaries) to/from the dialog, and to make changes to the existing localization dictionaries. The first time you create a dictionary by choosing a language in the add language menu, Smile builds a "trivial" dictionary, with as many lines of the kind:

```
"someKey" = "someKey";
```

as Smile could detect localizable keys in the dialogs. Smile gathers the relevant names and contents of all the dialog items, and searches the scripts for the `localize` keyword.
At this step, Smile may have missed some items: check the list and add the missing items if necessary.

Smile opens the dictionary in a new window. The left member of each line must remain strictly identical to the key that you have used, between quotes. The right member of the line is the translation of the key: you may change it into any Unicode string, e.g. you may write punctuation or Cangies. Localization dictionaries open in a Unicode text editor window: you can use in-line input for any language that the OS supports.

Conform strictly to the syntax for the localization dictionary: bracket each member with double quotes `"..."`, separate them with the equal `=` sign, end the line with a semi-colon `;`. Use `/*` and `*/` to bracket comments.

When that first "trivial" dictionary opens, it is not saved: make the modifications required (add punctuation, etc.) and save it (use the Save item of the File menu).

You can now define additional languages. When you add a new dictionary, Smile opens a copy of an existing dictionary: you will edit the right members of each line.

Later, the dialog may have changed, and you may want to again view the list of strings generated automatically. Use the extract strings button to do so.

15.8.3 How to localize Smile

Smile has its own localization dictionaries, that you can use from any script running in Smile. Thus, you may want to add strings of your own to Smile's dictionaries if you have to use localized strings when there is no specific custom dialog available.

To call Smile's dictionary, your script uses the `localize` verb. You can apply `localize` to a string or to a list of strings.

To edit Smile's localization dictionaries and make new ones, select Localize in the Dialogs section of the Scripts menu while the active window is not a dialog. You are prompted to confirm the command. Once you do so, use the menus of the Localize dialog.
You may have to relaunch Smile to make the changes in the dictionaries effective. To test dictionaries, use the settings in the Get info window in Finder.

15.8.4 How to localize "Localize"

The load dialog button of the Localize panel allows to load the active dialog for localization. Pressing down the *Option* key while clicking load dialog will load the Localize dialog itself.

15.9 Making a custom dialog into a stand-alone application

15.9.1 Why to make a custom dialog into a stand-alone application

You may want to make the dialog that you have designed into a stand-alone application if you want to distribute it to users who do not have Smile.
The stand-alone application is as simple to install and use as possible: it consists of one file (a Finder's package actually, like most OSX applications): the user copies only one double-clickable icon, and there is nothing else to install.

15.9.2 Why not to make a custom dialog into a stand-alone application

- in its present version, the stand-alone application does not have any useful menu.
  If you include a *User Scripts* folder in the same folder as the stand-alone application, though, the application shows a corresponding Scripts menu.

- the current version of Smile makes stand-alone applications with the same creator code as Smile's (`VIZF`). This may yield some cosmetic problems and/or some confusion for the users who would use both Smile and a stand-alone application.

- the stand-alone application is a rather big file (4.6 MB), no matter how simple your dialog is.

15.9.3 The limits of a stand-alone application

The stand-alone application can do what Smile can do — this is why it is such a big file [1]. Like Smile, it can process text files, as well as generate graphics or drive a digital I/O board.

15.9.4 Making a stand-alone application

Make a new dialog as described in this chapter. Save it to disk. Select the Make stand-alone application ... menu. You will be prompted to provide a name and a release number for your application. If the script runs successfully, you are presented with the new application in the Finder which you can double-click to launch.

The new application is a Finder package. You can browse its contents using the Show package contents item of Finder's contextual menu. The files and folders that you may want to change or to customize are located in the *Contents/Resources* directory of the package.

You can improve the stand-alone application by the following means:

icon Replace the *smile.icns* file with your own icon. This will provide a new icon to the application.

help Provide your own help files in the *Help* folders located in the *[language].proj* folders. Your help will be available in the Help menu.

[1] Probably Smile 2.5.3 will make smaller stand-alone application files, the user would have to install the Smile framework once for all.

Read Me Provide a Read Me file in the same folder as the stand-alone application. Do not make the Read Me with Smile: use TextEdit.

localization Use the mechanism described in section 15.8 to localize the dialog for any language. You may also need to perform some localization of Smile, which is also described.

customization In addition to the dialog which was made into the application, the application can use other files, provided you store them in the *More stuff* folder and provided you use the global `my gMoreStuffFolder` to access these files.

15.10 Attaching a custom dialog to an object

When you create a custom dialog, you can attach it to an existing object of Smile, such as a text window for instance. To attach the newly created dialog to an object `theObject`, use the high-level handler `DisplayDlog`:

Example 33

```
DisplayDlog (theObject, thePath)
```

Such a dialog is said to be owned by the object.
Attaching a dialog to an object has the following effects:

- when the dialog's owner is deleted, Smile closes the dialog automatically
- the dialog's `owner` property, containing a reference to `theObject`, is available to the dialog's `prepare` handler, which allows for

performing suitable initialization at the proper times, i.e. just before the dialog comes into view

- the dialog's dialog items whose `tag` property is set to the 4-character raw code of some property of the object is linked to that property: changing the contents of the dialog item immediately changes the property. This feature is limited to the built-in properties of objects.

Chapter 16

Scripting Smile — The basics

16.1 Overview

When you want to use Smile as an automation platform, you have to program Smile's behavior. To program Smile's behavior, you make objects (such as a window) and you provide them with the required behavior by supplying scripts to them.

In this chapter you become familiar with the basic aspects of scripting Smile:

- how you create objects and how you address them
- how you program the objects
- how you open files
- how you interface your scripts, using Smile dialogs
- how you schedule the execution of your scripts

Chapter 17 describes more advanced facets of Smile scripting.

16.2 Manipulating objects — The object model

16.2.1 Accessing an object

Accessing objects in Smile complies fully with the rules of AppleScript: you describe an object by specifying its class, its index — its creation index in the class — or its name or its `id`. The "canonical" description of objects (the unique description that AppleScript and Smile use internally) is by `id`.

Consistently with the general rule, windows support an alternate description: although `window` is not the class of any object in Smile, the expression `window n` (`n` being a positive integer) returns the nth window in the front-to-back order.

Thus `text window 1` returns the `text window` which was created first of all the text windows currently opened, while `window 1` returns the front window, whatever its class.

The `index` property of a window relates to the order as a `window`: you can change it to change the front-to-back ordering. You cannot change

the `index` of objects that are not windows.

The `name` property of objects is limited to 256 characters.

Smile supports the `whose` and `every` clauses:

Examples 34

```
reveal window 2 whose class is dialog
close every text window whose name ends
with ".html" saving yes
```

16.2.2 Making a new object by script

Like with any scriptable application, you can make new objects from scratch by applying `make new` to any of the classes displayed in the dictionary. Smile supports also more sophisticated ways of creating objects, intended to make non empty objects or objects that are fully customized.

Loading a document into a new object

When Smile opens a document file (a text file, or a compiled script, for instance), it not only creates a window for that document, but it also reads data from the file in order to customize and fill the window. To have Smile open a file into a new window, provide the file path as the `path name` property:

Example 35

```
make new text window with properties
{path name: thePath}
```

Creating a customized new object

The objects of Smile own two special properties, `properties` and `whole`. The `whole` property returns as a record the "structural" information about the object, including its elements and its script, but not the data it may contain such as the contents of a text window or the current state or contents of an item of a dialog. The `properties` property returns a record that contains the same information except that it does not contain the object's elements nor its script.

You can use the record returned by those properties to make a new object with those same properties. Since the record includes the class of the new object, you just ask Smile to make a new `basic object`:

Example 36

```
set theRecord to whole of theObject
make new basic object with properties
theRecord
-- will clone theObject
```

For instance you can create a custom Smile dialog by the usual means (about Smile dialogs see Chapter 15), then store it as a record in a script, and re-open it later by script without ever saving it to disk.

16.3 Programming the objects — The object scripts

16.3.1 Introduction to object scripts

To view or edit the object script of a object which is visible click its contents with the *apple-option* keys pressed. If the object is a dialog item, its container dialog must be in edit mode.

If the object is not visible, or to open an object script by script, use the EditObjectScript routine.

To close or to save an object script use the corresponding items of the File menu. Saving an object script saves it into the copy of the object which is currently loaded in Smile: the object script (like most of an object's properties) is saved to disk when you save the object.

Its object script is the object's script property. You can dynamically attach a script to an object, or change its script, by setting its script property by script. Provide a string containing a compilable script as the script property. To get the object's script's source, get its script property as text.

16.3.2 How to write object scripts

Smile defines invisibly a container property to all object scripts. An object script's container property returns the object it belongs to.

You may install two kinds of handlers in an object script: your own AppleScript subroutine handlers, and handlers for the AppleEvents sent by Smile, which belong to Smile's dictionary.

prepare When Smile creates an object, it sends a prepare event to the object's script before bringing it in view.
You handle prepare with a handler such as:

Example 37

```
on prepare theTarget
    -- do whatever required
end prepare
```

where theObject is a reference to the owner of the script, the object which is being created. You will probably want to perform any required initialization in that handler.

While its prepare handler is executing, a window is not visible yet, yet it is already the frontmost window, window 1. The previously frontmost window is now window 2.

Your prepare handler may — e.g. if some checking has failed — delete the object which is being created.

When an object contains other objects, Smile sends prepare to the container object's object script first, then to the contained objects' object scripts.

store When Smile saves an object, or when some script requests the value of an object's whole property — probably in order to save it in a form or another — Smile sends the store event to the object's script.
You handle store with a handler such as:

Example 38

```
on store theTarget
    -- do whatever required
end store
```

theTarget contains a reference to the object being saved, the owner of the script. The store event is intended for performing any necessary update or cleaning which could be required before saving the object.

close Whenever the user closes an object, Smile sends a close event to the object's script.

You handle `close` with a handler such as:

Example 39

```
on close theObject saving whatever
    -- do whatever suitable
    continue close theObject saving
whatever
end close
```

`theObject` contains a reference to the object being closed, the owner of the script. The `continue` command is required to have Smile close eventually the object.
Note that the `saving` parameter of the `close` event is required: your script must specify `saving no`, `saving yes` or `saving ask`.

`delete` Whenever Smile deletes an object (for instance, when the user dismisses a window), Smile sends a `delete` event to the object's script. You handle `delete` with a handler such as:

Example 40

```
on delete theObject
    -- do whatever suitable
    continue delete theObject
end delete
```

`theObject` contains a reference to the object being deleted, the owner of the script. The `continue` command is required to have Smile dispose eventually of the object: you can choose — e.g. if some checking has failed — not to send it.

As for the 2.5.2 version of Smile, the dialog items do not receive `delete`.

`idle` When Smile is idle, it sends periodically the `idle` event to those windows (i.e.: to their scripts) whose `want idle` property is set to `true`.

You handle `idle` with a handler such as:

Example 41

```
on idle theWindow
    -- do whatever suitable
    return 3
end idle
```

`idle` should return a real number. This real number specifies how much time, in seconds, Smile will wait before sending again `idle` to the window.

`click in, drop, export` Those events are specific to the controls, those items that receive the user's actions. They are described in detail in the chapter specific to Smile dialogs, in Section 15.7.

By default, Smile keeps silent the execution errors that occur in the `click in`, `drop` and `export` handlers. If you want to be notified of execution errors that might occur in one of those handlers, encapsulate the body of your handler within a `try` wrapper.

Each object script has as its parent script the class script of the object, a script shared by all the objects of the same class. Class scripts are described in section 17.3. If the object script includes a handler which intercepts a call previously handled by the class script (an advanced programming usage), use `continue` to propagate the call to the script's parent — the class script.

16.3.3 How to send commands to an object script

You can call the handlers that an object script contains from any script. The calls to AppleScript subroutines must be encapsulated in a `tell theObject` statement, where `theObject` should refer to the owner of the script.
In a `tell` statement you refer to the object with the special variable `it`:

Example 42
```
tell window 1 to AddOne(it's name)
```

Any object script has a parent script, as described in the section 17.3 about Class scripts. If the object script itself does not handle the call, then the call will be sent to its parent script, and so on.

The verbs ("AppleEvents") that are given in the dictionary and that support a direct parameter follow a special rule concerning their target: Smile automatically redirects them to the direct parameter. Thus the following script:

Example 43
```
close window 1 saving ask
```

is equivalent to:

Example 44
```
tell window 1 to close it saving ask
```

However, you need to use the `tell` wrapper if the caller script itself already defines a `close` handler and you really want to call the target's handler.

16.3.4 The object script, a better script object

Although they have not been designed with that purpose in mind, Smile's object scripts offer a favorable framework to work with AppleScript's script objects.

Suppose that your project has to handle several AppleScript's scripts. Normally, you would use the various ways AppleScript offers to define scripts. If instead you create some objects in Smile (windows or interface items in dialogs, typically) and you provide your scripts as these object's scripts, you will be able to use your scripts much more simply and reliably — finally, you will script faster:

- you access your scripts through the object model:

  Example 45
  ```
  tell element "Beeper" of element
  "Sound palette" of dialog " to
  buzz(3)
  ```

  Indeed, unless the object handles directly the command (in which case the command would belong to the dictionary), the `tell` will really send the command to the object's script.

- no more ambiguity about the script's instanciation: scripters often meet issues where they unwillingly make multiple copies of scripts in AppleScript variables, and that require much care to be handled properly. When your scripts are script objects, those problems no longer exist: the object model provides one unique access to one unique script that "lives" as long as its owner exists.

(Of course, making copies of those scripts into AppleScript variables remain possible).

- you can edit your script by script: most of the scriptable features of scripts (described in the chapter about the advanced scripting of Smile, Chapter 17) work only on object scripts.

16.4 Opening a file by script

Like any scriptable application, Smile handles the `open` event. Though, it may be more convenient in order to open files to use some higher-level handlers that Smile defines. For instance `open thePath` would open a new copy if the document is already open, while the high-level handler `DoOpen(thePath)` will instead bring into view the already open document.
The high-level handlers for opening files are described in Appendix F about Smile's built-in routines.

16.5 Providing a GUI — The Smile dialogs

The `dialog` class, a sub-class of the `window` class, is a special kind of window which supports the Aqua interface components, the `dialog items`, as its elements. How to build, script and use custom dialogs is described in chapter 15.

Smile includes a standard user interface layer to build custom dialogs — you use drag and drop or Copy-Paste from a Palette — and to launch them — you save and you open custom dialogs with the standard File menu.
Associating scripts to such a dialog is how you provide an interface to a script or to a set of scripts.

16.6 Scheduling tasks

Smile includes a set of features helpful for scheduling tasks:

`smilepause:` pauses a script without hanging the application

`idle:` when Smile is idle, it sends periodically the `idle` event to those windows (i.e.: to their scripts) whose `want idle` property is set to `true`.

You handle `idle` with a handler such as:

Example 46

```
on idle theWindow
    -- do whatever suitable
    return 3
end idle
```

`idle` should return a number. This number specifies how much time, in seconds, Smile will wait before sending again `idle` to the window.

`notify:` when used with the `with delay` option, `notify` schedules for a given time in the future the broadcast of a message to a given recipient. If the recipient's script includes a `notify` handler, the handler will receive the message and the sender's identification.

Chapter 17

Scripting Smile — Advanced features

17.1 Overview

This chapter is for experts who wish to go still one step beyond towards automation of tasks. It describes advanced features. Inappropriate use of those features may prevent Smile from launching or documents from opening, and can induce data loss: proceed with caution.

17.2 Making and editing scripts by script

The `script` property of an object returns its script as a script object (in the strict AppleScript sense). If you get it `as text` then the `script` returns the source text of the script. You can set the `script` property to any compilable text.

When you manipulate scripts by script, you may want to get the source of a compiled script document. Use exactly the following construct: `get script of file thePath` where `thePath` is the path to the file, as a string.

You can create and edit dynamically the handlers of the object scripts:

```
name of every handler of script of
window 1
handler "foobar" of script of window 1
as text
set handler "foobar" of script of
window 1 to theString
set theHandler to handler "foobar" of
script of window 1
set handler "barfoo" of script of
window 1 to theHandler
```

You can create and edit dynamically the properties of the object scripts, using the `variable` keyword:

```
every variable of script of window 1
- - returns the properties and their
values as a record
name of every variable of script of
window 1
get variable "foobar" of script of
window 1
set variable "foobar" of script of
window 1 to anyValue
```

You can also create on the fly properties to objects:
`set barfoo of window 1 to anyValue`
but only the properties created using the

`variable` keyword will be returned by the `every variable` expression.

Note that the changes made to the handlers and the properties of object scripts are effective although they are not viewable when you edit the script of the object by the normal means.

17.3 The Class scripts — Defining new classes

17.3.1 An introduction to class scripts

The active *Class Scripts* folder contains script files. Each of these files handles the behavior of one object class of Smile — as described in more details in Chapter 24. The `class script` property, owned by every object, refers to this script.

When an object has an non-empty object script, this script has automatically the object's class script as its parent.

The object classes support an inheritance mechanism, as you can see by browsing the `Class` entries of the dictionary. For instance, `script window` is a sub-class of `text window`, itself a sub-class of `window`. The inheritance applies to the class scripts: the `window` class' class script is the `text window` class' class script's parent.

Through inheritance, every class is a sub-class of the same virtual class `basic object`. The class script of the `basic object` is *Context*. Thus *Context* is a common parent to all object scripts.

You can open a read-only version of the class script of an object with the `EditClassScript` handler.

You can customize the class scripts by using the same terms as described above for object scripts, for instance:

```
set handler "foobar" of class script of
window 1 to theString
```

Smile uses three class scripts that are special, the *Context* script, described above, the application's Class script (*Application*), and another special script named *Globals*. Smile saves the properties of *Globals* when you quit Smile. The properties of *Globals* are those properties which are accessed with the `my` prefix or the `of me` suffix.

- to refer to the *Context* script, use the special keyword `context`
- to refer to the *Globals* script, use the special keyword `globals`
- to refer to the *Application* script, use `class script of me`

17.3.2 Creating custom classes

Class scripts can be used as libraries, where you store handlers which will be available from any object of some given class. You can use the existing class scripts to store your personal handlers, if you know what you are doing.

You can also create dynamically new classes, and attach class scripts to them. Smile lets you create classes dynamically (i.e., by script), attach class scripts to them, which can have

another class script as their parent, and that way create your own hierarchical system of classes.

This system of custom class scripts is intended for defining and using easily custom libraries of scripts in objects of Smile.

Here are the steps to follow to create a new class:

1. Write the library script. Save this script preferably in the *Class Scripts* folder. In the following, suppose you save it as the *MyClass* file.

2. Make the new class: create a new object of class `class script`. You need to provide the following information:

   - the `class script` property, a 4-characters code
   - optionally a `parent` property, another class, or its 4-characters code if that parent property is itself a custom class
   - the `path name` property, the path to the new class script. If the file is located in the *Class Scripts* folder, just provide the file name

   Example 47

   ```
   make new class script with
   properties {class script:"MyCl",
   parent:text window, path
   name:"MyClass"}
   ```

   Here the new class inherits from the `text window` class.
   You can now create and open documents of the ”MyCl” class: they will be provided with *MyClass* as their class script.

3. Create documents with the new class.
   To create a text window with the new class, run :

   ```
   make new text window with properties
   {class script:"MyCl"}
   ```

   To turn a standard text window (`theWind`, here) into a window of the new class, run:

   ```
   set class script of theWind to
   "iTxt"
   ```

   When you save it, the document keeps its custom `class script` property.

4. Create a class when Smile launches, so that next time Smile is launched it recognizes automatically documents belonging to the new class. Save the one-liner script that creates the new class:

   ```
   make new class script with
   properties {class script:"MyCl",
   parent:text window, path
   name:"MyClass"}
   ```

   as a script document in the *Initialization* folder of the *More stuff* folder.

Chapter 18

About Smile's libraries

18.1 Overview

Several of the subsequent chapters present the extensions to the AppleScript language that Smile offers. This chapter enlightens where those different extensions really "live". Together, these extensions sum to what is called below Smile's "libraries".

Two categories of extensions are available to your scripts. First, Smile offers the commands that can be found in its dictionary; more commands can be found in the dictionary of the Satimage osax, Smile's companion Scripting Addition.
The second category gathers handlers which are available to Smile's context. By different means that are described below Smile defines a number of handlers at launch time. Since Smile's context is persistent, your scripts can use these handlers. Yet, they are not declared in an AppleScript dictionary.

More precisely, Smile's libraries are made of the following resources:

Smile's dictionary

Satimage osax' dictionary

Smile's routines: routines that are included in the `Context` or the `Application` scripts, that Smile compiles into its context at launch time

Context additions: routines that are defined in the files located in the *Class Scripts/Context additions* directory, and that Smile compiles also into its context at launch time.
Thus, some of the routines may require a specific installation.

18.2 Documentation about Smile's libraries

The chapters below give a thematic presentation of Smile's libraries. Under each theme, we give:

- the concerned commands and handlers
- for each of those, a summarized description
- a hypertext link to the entry in the corresponding Appendix
- additional detailed information when useful

At the end of the Manual, separated Appendices provide the lists of the commands and

handlers that can be found in Smile's dictionary, in the Satimage osax, in Smile's routines, and in the Context additions. These Appendices are where you find the syntax of each term: the syntax of the commands and handlers is not fully provided in the subsequent chapters, furthermore the latter are not exhaustive.

Chapter 19

General purpose library

19.1 Strings

Short description

- `find text`: find text literally or using regular expression syntax.
- `change`: replace all occurrences of a substring
- `re_compile`: compile a regular expression
- `extract string`: extract a substring out of a string.
- `uppercase`: move to uppercase.
- `lowercase`: move to lowercase.
- `converttext`: convert between encodings that you specify as integers or as strings
- `textencodings`: provides the encodings available as integers or as strings
- `convert to Windows`: converts a Mac string into a Windows string
- `convert to Mac`: converts a Windows string into a Mac string
- `make new name`: supplies a unique name in the form `YYMMDD_HHMMSS`
- `format`: format a real number using a specification string. Ex: format pi into "##.##"->"3.14". "0" instead of "#" forces trailing zeros. "^" adds a space. "+f1;-f2;f3" provides formats for numbers >0, <0, =0. Encapsulate custom strings with "".
- `special concat`: append a new column to an array given in text format
- `extractcolumn`: extract columns from an array

Comments

- `find text` and `change`: if called from Smile, `find text` and `change` support as their `in` parameter — in addition to strings — a reference to a window of Smile, or a reference to any range of text in a window of Smile.

  Example 48
  ```
  change ":" into "/" in first
  paragraph of selection of window
  1
  ```

  Regarding how to describe a range of text of a window, see Chapter 13 about

Smile's Text Suite.

`find text` and `change` also support a file as their `in` parameter. If `change` is called with a file as its `in` parameter, the file remains unchanged and `change` returns the new string.

- `special concat`: `special concat` appends a new column to an array given in text format. The direct parameter should be a string representing an array with tab-delimited columns and return-delimited rows. The `with` parameter is the column to append, formatted into a return-delimited string. `special concat` will return the array with the new column appended, as a string.

- `extractcolumn`: if the `in` parameter is a table given as a string delimited with `tab` and `return`, `extractcolumn` will return the required column(s) as text.

  You specify a range of columns with the direct parameter and the `to` parameter:

  Example 49
  ```
  extractcolumn 2 to 4 in theTable
  ```

  You specify a set of non necessarily contiguous columns by supplying a list as the direct parameter:

  Example 50
  ```
  extractcolumn {2, 4} in theTable
  ```

  If you specify a non-existing column index, `extractcolumn` returns a column of empty strings.

  Requesting the result `as list` provides additional possibilities, as described now with an example.

  Let theTable be the following string:
  ```
  Year    Inc.    %
  2018    708     .3
  2019    712     .6
  ```

  - To get the list of the items as strings, request the result `as list`:

    Example 51
    ```
    extractcolumn {1, 2} in theTable
    as list
    -- {"Year", "Inc.", "2018",
    "708", "2019", "712"}
    ```

  - To get the list of the columns as lists of strings, provide the column indices as one list of lists:

    Example 52
    ```
    extractcolumn {{1}, {2}} in
    theTable as list
    -- {{"Year", "2018", "2019"},
    {"Inc.", "708", "712"}}
    ```

  - To get the list of the rows as lists of strings, provide the column indices as one list of one list:

    Example 53
    ```
    extractcolumn {{1, 3}} in
    theTable as list
    -- {{"Year", "%"}, {"2018",
    ".3"}, {"2019", ".2"}}
    ```

Instead of providing a tabulated string as the `in` parameter, you can provide the table as the list of the rows as lists of strings:

Example 54

```
set theTable to {{"Year", "Inc.",
"%"}, {"2018", "708", ".3"},
{"2019", "712", ".6"}}
```

in which case the default behavior is `as list`: you must specify `as string` if you want the result as one tabulated string.

See also the use of `extractcolumn` to generate arrays of numbers in the chapter about Mathematical libraries.

- `make new name`: supplies a unique name, based on the current time and date, under the form YYMMDD_HHMMSS. If several calls are issued in the same second, `make new name` still provides different names.
  Files which are named with `make new name` assume the chronological order of their creation dates when listed alphabetically.

- `format`: the formatting string for the `format` command consists of the following metacharacters:

  `#`: stands for an optional digit

  `0`: stands for a required digit

  `^`: stands for a required digit, but `format` prints spaces ␣ instead of the leading and trailing zeros

  `.` (period): indicates the location of the (optional) decimal separator

  Examples 55

  ```
  format 3.14 into "###.000"
  -- "3.140"
  format 3.14 into "000.#"
  -- "003.1"
  format 3.14 into "^^^.#"
  -- "␣␣3.1"
  ```

  `+`, `-`, `(` and `)` : stand literaly for themselves

  Examples 56

  ```
  format 3.14 into "(+###.000)"
  -- "(+3.140)"
  format -3.14 into "(+###.000)"
  -- "-(+3.140)"
  ```

  `+` and `-` really make sense when used with the following metacharacter:

  `;` (semi-colon): as you can observe on the latest example, `format` just prepends a `-` minus sign for negative numbers. Alternately you can provide a different formatting string for negative numbers using the `;` metacharacter. To do so, specify a formatting string in the form `f1;f2;f3`: the substrings `f1`, `f2` and `f3` being the formatting strings for the positive, negative and null numbers respectively.
  When you specify such a kind of formatting string it becomes your responsibility to display the sign or not: use `+` and `-`.

  Examples 57

  ```
  format 3.14 into
  "+00;(-#0.00);#"
  -- "+03"
  format -3.14 into
  "+00;(-#0.00);#"
  -- "(-3.14)"
  format 0 into "+00;(-#0.00);#"
  ```

```
-- "0"
```

To customize further the formatting string, use the following metacharacters:

' (non-smart single quote): encapsulates any string

Examples 58

```
format 3.14 into "+00;'ALERT !
LEVEL '-00.00;'(empty)'"
-- "+03"
format -3.14 into "+00;'ALERT !
LEVEL '-00.00;'(empty)'"
-- "ALERT ! LEVEL -03.14"
format 0.0 into "+00;'ALERT !
LEVEL '-00.00;'(empty)'"
-- "(empty)"
```

%: displays the number as a percentage.

Example 59

```
format 0.14 into "###.0' '%"
-- "14.0 %"
```

19.2 Lists and records

Short description

- `special concat`: concatenate {a_ppty: X, ...} and {a_ppty: Y, ...} into {a_ppty: Z, ...} where Z is the union X & Y if X and Y are lists, and where Z is the sum X + Y if X and Y are numbers.
- `suppress item`: delete an item from a list or a record.
- `extractcolumn`: can make tabulated strings into lists, and it can handle tables provided as lists: see the entry for `extractcolumn` in the section about string manipulation.
- `sort`: recursive sort.
- `heapsort`: non-recursive sort.

Comments

- `suppress item`: if the `from` parameter is a list then the direct object of `suppress item` has to be an integer, the index of the item that you want to suppress. If the `from` parameter is a record then you can specify as the direct object a quantity of the following classes:
  - an integer, the index of the item that you want to suppress.

    Example 60

    ```
    suppress item 1 from
    {pencils:0, erasers:3}
    -- {erasers:3}
    ```
  - a string, to suppress an item provided with a user-defined label.

    Example 61

    ```
    suppress item "pencils" from
    {pencils:0, erasers:3}
    -- {erasers:3}
    ```
  - a four-character string, to suppress an item provided with a raw-code label.

    Example 62

    ```
    suppress item "penc" from
    ```

```
{«class penc»:0, «class
eras»:3}
-- {«class eras»:3}
```

- a keyword — with no quotes — to suppress an item provided with that keycode as its label.

  Example 63

  ```
  suppress item menu from
  {menu:"Foie gras frais",
  price_USD:25}
  -- {price_USD:25}
  ```

- `heapsort` and `sort` differ on the following points:
  - `sort` is recursive, `heapsort` is not: you will not hit any `Stack overflow` issue with `heapsort`, while you may hit one if you use `sort` on very large lists.
  - `heapsort` is slightly slower on the average than `sort`, but its execution time is essentially independent on the order of the list in input. On quasi-ordered lists (resp. on particularly randomized lists) `sort` is faster (resp. much slower) than `heapsort`.
  - `heapsort` can easily be changed to perform a partial (and faster) ordering: if you replace the `n` variable in the first `repeat` by a smaller value `m`, the list returned by `heapsort` will have the `m` lowest values, properly sorted, as its `m` first values, the rest of the list will contain the larger values, not sorted.

  You can customize both `heapsort` and `sort` e.g. if you need to implement a custom ordering rule. The source of those handlers is in the *Class scripts/Applications* script file. If you copy then customize and rename the `sort` handler, it is important that you change also the calls to `sort` which are in `sort`'s source.

19.3 Files and resources

Short description

- `remote info for`: locate an alias on the network
- `list files`: the list of the files contained in the folder
- `backup`: synchronizes 2 folders.
- `read binary`: read a file of real or small real
- `write binary`: write the data into a binary file of small real (4 bytes per number)
- `load resource`: get the resource of the given type and id from the specified file
- `list resources`: return the list of the ids of the resources of the specified type stored in the specified file
- `get resource name`: return the name of the resource of the specified type and id from the specified file
- `put resource`: write the given resource to the specified file with specified type and id

Comments

- `backup`: `backup` resolves the aliases located at the first level of the source folder and of the destination folder, and it does not resolve the aliases located deeper. Thus, you may fill the source folder with aliases to the original folders that you want to backup. In the destination folder, you will put aliases to the copies, that need to be synchronized, supplying to each alias the same name as the corresponding alias to an original folder.

- `put resource`: `put resource` stores the data that you provide, not their class. In order to specify the class of the data, provide a name to the new resource with the `with name` parameter. That name has to be the 4-characters code for the class. Later when you load the resource into an AppleScript variable using `load resource`, Smile will attempt to coerce the data contained in the resource into the class whose 4-characters code can be found as the resource's name.
  A trick to find out the 4-characters code of a given class is to apply `display dialog` to it, e.g. `display dialog integer`.

19.4 Scripts

Short description

- `execute`: run the script of a script window
- `check syntax`: check syntax of a script window
- `do script`: execute a script
- `display`: return the direct object as a string
- `find definition for`: retrieves the dictionary that contains a given term.

Comments

- `do script`: performs essentially the same action as the Standard Additions' `run script` command, but unlike the latter you can request the result of `do script as text`.

- `display`: `display` will return the direct parameter as a string, even if applied to a quantity that cannot be coerced into a string. For instance you may use `display` to parse records when you do not know what labels it may contain.
  `do script` and `display` are somehow inverse operators of each other.
  Example 64
  ```
  display {age: 33}
  -- "{age: 33}" -- a string
  do script "{age: 33}"
  -- {age: 33} -- a record
  ```

19.5 User interaction

Short description

- `navchoose file`: choose file with Navigation Services
- `navchoose folder`: choose folder with Navigation Services
- `navchoose object`: choose file or folder with Navigation Services
- `navchoose volume`: choose volume with Navigation Services
- `navask save`: prompt for save

- `navchoose file name`: get a new file specification from the user, without creating the file. Uses Navigation Services
- `navnew folder`: get a new folder specification from the user. Uses Navigation Services
- `choose color`: choose a color with the color picker
- `smilepause`: pauses a script without hanging the application
- `chrono`: returns the time elapsed since the last call to `chrono` (seconds)
- `modifiers`: returns the list of the modifiers keys which are being pressed
- `mouse location`: returns the mouse location as the list of the two coordinates. The origin is the upper left corner.
- `mouse button`: returns the state of the mouse button
- `menu` and `menu item`: two classes intended for customizing Smile's menu bar
- `FatalAlert`: display an alert box with the Stop icon and one OK button
- `dd`: display an alert box with the Note icon and one OK button
- `AskUser`: request an input from the user
- `quietmsg`: prints to Smile's Console
- `msg`: prints to Smile's Console and brings the Console into view
- `DoOpen`: open any file or reveal its window if the file is already open

Comments

- `smilepause`: `smilepause x` suspends the execution of the script for `x` seconds. During that pause, Smile is fully responsive: you can use your computer normally.

  By default, two keys have a special action while the script is paused with `smilepause`: the *esc* key triggers a `User canceled` error (error number -128), while the → (right arrow) key just exits smilepause, resuming execution of the script.
  If you set the `«class unti»` parameter (not presented in the dictionary) to `false`, then those keys have no special effect, and there is no way of exiting the pause before it is elapsed: `smilepause 3 without «class unti»`

  Typical uses of `smilepause` include the following:

  - inserting `smilepause 0` in a loop brings three advantages: 1. the interface remains responsive while the loop is executing, 2. more graphical update can take place (e.g., update of the progress bars and of the chasing arrows), 3. you can interrupt the loop by typing the `esc` key.
    Note however that while the loop is executing, the → (right arrow) key does not work as usual since `smilepause` intercepts the right arrow keystrokes.
  - if you use Smile to present a slide show (for instance, SmileLab plots) you can use long `smilepause`'s to have the show run alone. To switch to the next slide press the → (right arrow) key. To

terminate the show press the *Esc* key: this will trigger a `User canceled` error (that your script may handle).
  - you may want to display a message for a few seconds only: use `smilepause without «class unti»`. While the message is being displayed, the computer is fully responsive and all keys work as usual.

While some script is paused with `smilepause`, you can do any action, such as selecting a menu or typing.
In particular, you can launch another script. The execution of that other script will take place "inside" the pause: the pause itself is suspended until the second script returns. Therefore, the second script will return before the first one.

- `chrono`, `modifiers`, `mouse location` and `mouse button`: these quantities are properties of Smile, they are documented in the entry about the `application` class in the dictionary of Smile.

- `menu` and `menu item`: you change the keyboard shortcut associated with a menu item by editing two properties of the `menu item` class: `shortcut` and `modifiers`.
By default, `shortcut` is the key which, in conjunction with the *apple* key, will activate the menu item.
When you set the `modifiers` property of some `menu item`, the *apple* key is assumed even if you don't specify it in the list.
Though, it you set the `modifiers` property of some `menu item` to a list that contains "no command" as one of its items, then the keyboard combo will no longer include the *apple* key. This way, you can map directly one key of the keyboard to one menu item — use this feature with caution. For instance if you execute:

Example 65
```
set modifiers of menu item "Save" of
menu "File" to {"no command"}
```

then each time the user types *S* the front window will get saved.

Usually, if you customize the `menus` and the `menu items`, you will store a script which does so in the *More stuff/Initialization* folder, so that it runs when Smile opens.

- `quietmsg` and `msg`: insert `quietmsg` in your scripts when you want to print any useful information such as intermediate results, and use `msg` when some information may require the user's attention.

Chapter 20

Mathematical library

20.1 Functions

Short description When it makes sense, the mathematical functions support as their direct parameter (and return) a list of numbers or an array of real (a class defined in the Satimage osax). Angles are in radian.

- `abs`
- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atan2`: `atan2 (y, x)`
- `atanh`
- `cosh`
- `cos`
- `erf`
- `erfc`
- `exp`
- `gamma`
- `hypot`
- `lgamma`
- `ln`
- `log10`
- `sin`
- `sinh`
- `sqr`
- `sqrt`
- `tan`
- `tanh`

Comments Keep in mind that AppleScript really defines commands with direct parameters, not functions. So a correct syntax is:

Example 66

```
sin pi
```

Here, adding parentheses may be misleading:

Example 67

```
sin (pi) / 4
```

will really first compute the ratio `pi / 4`, then apply the sinus, exactly like:

Example 68
```
sin (pi / 4)
```

Thus, parentheses may be required around the command itself:

Example 69
```
(sin pi) / 4
```

20.2 Lists and arrays of numbers

Short description

- `array of real`: a packed list of small real
- `multlist`: performs the product of the parameters. Each parameter may be a list of numbers, an `array of real`, or a single number e.g. `multlist thePolygon with theScaling`
- `divlist`: quotient
- `addlist`: sum
- `sublist`: subtraction
- `statlist`: returns the min, max, min's index, max' index, mean, standard deviation.
- `reversearray`: reverse
- `replacemissingvalue`: replace the `missing value`'s and the NAN's
- `read binary`: read a file of real or small real
- `write binary`: write the data into a binary file of small real (4 bytes per number)
- `extractitem`: extracts a sub-array
- `creatematrix`: create an `array of real` of size `ncols*nrows`
- `extractcolumn`: extract columns out of a 2D array provided as a string

Comments

- `array of real`: The Satimage osax defines one class of data, `array of real`, and a set of operators which work with that class. An `array of real` is logically equivalent to a list of real numbers but it is more reliable and faster: you should use `arrays of real` to compute with large lists or when speed is an issue.
  The operators for lists work both on regular lists of numbers and on `arrays of real`.
  To coerce an `array of real` into a regular AppleScript list of real numbers, request it `as list of real`. To make a list of real numbers into an `array of real`, use `as array of real`.
  Those coercions work also between one list of lists of numbers and one list of `arrays of real`.
- `multlist`, `divlist`, `addlist`, `sublist` and `statlist`: In addition to lists of real numbers and on `arrays of real`, those operators accept lists of strings — provided the strings represent numbers. You can use those operators to convert a list of strings

into a list of real:

Example 70

```
addlist {"1", "2"} with 0
-- {1.0, 2.0}
```

- `extractcolumn`: `extractcolumn`'s basics are described in the section about string manipulations.
  You can request the result of `extractcolumn as real` or `as array of real`. The features described with `as list` extend to real numbers:
  - if you use the `i to j` construct or the `{i, j, etc.}` construct to specify the columns, `extractcolumn` will return one list of numbers or the list of the columns as `arrays of real`.
  - if you specify the columns in the form `{{i}, {j}, etc.}`, `extractcolumn` will return the list of the columns as lists of numbers or as `arrays of real`.
  - if you specify the columns in the form `{{i, j, etc.}}`, `extractcolumn` will return the list of the rows as lists of numbers or as `arrays of real`.

  The non-numbers result in `missing value` items (the `array of real` class supports a `missing value`).

Chapter 21

RS232 library

21.1 Overview

Smile's RS232 library allows you to control by script the Keyspan USB/RS232 adaptors. By script, you configure the RS232 serial links and you receive and you send characters. To control the serial link, you will create one or several instances of the `RS232` class.

21.2 Instructions of use

Here is how you would use the RS232.

1. Install the driver software that ships with the adaptor

2. Plug the USB connector of the adaptor into your machine

3. Launch Smile

4. From any text window, run `serial ports` (`serial ports` is given in the dictionary as a property of the application). This will return a list of lists of three items. Each of the lists provides information regarding one of the serial ports available on your machine. The three items are: a number that identifies the kind of the port, a UNIX path name, and the name of the port. One of thoses lists with 9 as their identification number refers to the Keyspan adaptor: select a list whose name includes neither "modem" nor "IrDA". You will use information from that list that you have selected.

5. Make a new instance of the `RS232` class and provide it with the UNIX path you got with `serial ports` as its `configname` property:

   Example 71

   ```
   set theRS to make new RS232
   with properties {configname:
   theUNIXPath}
   ```

6. Activate the RS232 by setting its `enabled` property to `true`:

   Example 72

   ```
   set enabled of theRS to true
   ```

7. Configure the RS232 by setting its `RSOptions` property. Provide a record as described in the dictionary at the entry

about `Class RSOptions`, for instance:

Example 73

```
set RSOptions of theRS to {bauds:
9600, databits: 8, stopbits: 1}
```

8. To send characters `set` the `contained data` property of the RS232:

   Example 74

   ```
   set contained data of theRS to
   "hello world"
   ```

9. To receive characters `get` the `contained data` property of the RS232:

   Example 75

   ```
   try
       set theBytes to contained data
   of theRS
   on error -- nothing was received
       set theBytes to ""
   end try
   ```

   This reads and resets the RS232's receiving buffer.

Chapter 22

Digital I/O library

22.1 Overview

Smile's Digital I/O library allows you to control by script the Delcom USB/Digital I/O board. By script, you write to and you read from digital (TTL) outputs and inputs via the board. No additional software is needed: Smile includes the USB driver for the Delcom board. Each of the two ports (8 lines each) can work as an output port or an input word. You can plug several boards to get more lines.
Typically you connect the input lines to opto-couplers and the output lines to electrical relays.

22.2 Instructions of use

1. Plug in the USB connector of the Delcom board into your machine

2. Create a new instance of the `Delcom USB Board` class

3. To write to output lines, `set` the `contained data` property of a given port (you set 8 bits in one command) or the `contained data` property of a given bit:

   Example 76

   ```
   set theIO to make new Delcom USB
   Board
   set contained data of bit 8 of
   digital port 2 of theIO to 1
   ```

4. To read inputs, `get` the `contained data` property of a given port (you get 8 bits in one command) or the `contained data` property of a given bit:

   Example 77

   ```
   get contained data of digital port 2
   of theIO
   ```

Chapter 23

PDF library — The Graphic Kernel

23.1 Overview

The PDF generation library allows you to make PDF records (strings, actually) that represent vectorial drawings. PDF records can be saved as PDF files, they can be displayed in "Graphic windows" — which can be saved as PDF files — and also in Custom dialogs.

The PDF generation library is expandable. You can write your own high-level graphic libraries and make them available to your scripts. One example of such a library (GeomLib) is included in the standard distribution of Smile.

23.2 Producing a graphic in a window

23.2.1 The basics

When you write a script in order to produce a graphic in a window, your script should do the following:

- create a window — this is optional, Smile can do it for you
- reset it for drawing, by calling `BeginFigure`
- define the drawing, using the graphic functions described below in this chapter
- instruct the window to realize the drawing, by calling `EndFigure`

Example 78

```
BeginFigure(0)
CirclePath({250, 250}, 2)
DrawPath(1)
EndFigure()
```

The distribution of Smile includes more examples. The examples are scripts provided as text files. If you are not familiar with executing scripts from text windows you may want to read chapter 4.

- To move the image drag it while holding the option (alt) key down.
- To resize the image's frame (not the window's frame) drag its right bottom corner. When you export or save the image as a PDF, the PDF will assume that frame as its size.

- To save the image as a PDF file, use the Save item of the File menu.
- To save the image as a TIFF file, use the Save item of the File menu, and provide the *.tiff* extension to the new file's name.

23.2.2 The graphic window

Since drawing occurs in a graphic window, one has to make a new graphic window.
If one graphic window is enough for your purposes, if you want to perform a drawing on the fly, just call `BeginFigure` with the `0` parameter. Smile will create automatically the graphic window if needed. It will then always use that same window upon new `BeginFigure (0)` calls, not clogging your screen space as you perform successive trials.

If you want to make multiple windows, or if your script requires a reference to the graphic window, create it yourself, and then pass its reference to `BeginFigure`:

Example 79

```
set theWind to make new graphic window
with properties {name: "Tiny dot"}
BeginFigure(theWind)
CirclePath({250, 250}, 2)
DrawPath(1)
EndFigure()
```

23.2.3 The graphical objects

The instructions given above do realize one static drawing in a graphic window. The drawing is really a property of the graphic window, it is not an object. For a more dynamical behavior, you will define graphical objects.

Smile defines a generic graphical object, the `picture view`, an element of the graphic window. The `picture view` provides a way for performing basic animations. Below we describe how to have a `picture view` display some graphics and how to use it to perform animated graphics.

The SmileLab section of Smile's dictionary defines several graphical objects that are specialized in representing numerical data. SmileLab's features are documented in a separate documentation that ships with the *SmileLabSet* package, a separate download.

23.3 The basics of the PDF language

PDF handles "graphic states", "paths" and text. PDF draws the paths and draws the text, according to the current graphic state.

23.3.1 The paths

A path is a sequence of drawing commands which may contain lines, moves, arcs, beziers etc... In order to make the drawing effective you must call the `DrawPath` handler (see G.3.1).

Example 80

```
Moveto({0, 0})
LineTo({100, 150.1})
DrawPath(2) -- 2 = stroke
```

draws a line with the current pencolor, penwidth and dashpattern.
The coordinates are real numbers, and by

default the unit is one point = 1/72 inch. The default origin is the bottom left corner.

23.3.2 The Graphic State

The graphic state includes the current properties (penwidth, colors, transformations, text font & size, ...) which will be used when you invoke the `DrawPath` or `DrawText` functions. If a block of lines changes the graphic state, you may want to bracket it between `SaveState()` and `RestoreState()`.

Example 81

```
SaveState()
Moveto({0, 0})
LineTo({100, 150.1})
SetPenGray(0.5)
SetDashPattern({0, 2, 4})
SetPenWidth(0.8)
DrawPath(2)
RestoreState()
```

draws a gray dashed line without corrupting the current graphic state.

Example 82

```
SaveState()
SetTransformation({20, 10} & {30, -60} &
{100, 100})
CirclePath({0, 0}, 1)
RestoreState()
DrawPath(2)
```

draws an ellipsis centered at `{100,100}` with axis `{20,10}` and `{30,-60}`. Here we need to call `SaveState-RestoreState` in order to restrict the transformation to `CirclePath`. Calling `DrawPath` before `RestoreState` would result in the penwidth undergoing the transformation, whence a huge stroke.

23.4 The graphic commands

Graphic commands, examples of which were given above, are the commands which do define the drawing. They are documented in a separate Appendix G. The Graphic Kernel Quick Reference item of the Help menu provides a short and convenient reminder of the graphical commands of Smile's PDF library.

23.5 Producing PDF data

23.5.1 The basics

You can produce PDF data (a "PDF record") without creating first a graphic window. Here is what you can do with a PDF record:

- you can save it as a PDF file
- you can append it to an existing PDF file
- you can give it to a graphic window as its background picture or as its foreground picture. This is particularly intended for finalizing graphics produced with SmileLab
- you can display an animation in a graphic window
- you can display it in a custom dialog

To produce a PDF record, your script should do the following:

- specify the size of the PDF and initialiaze the new PDF by calling `BeginPDF(theRect)`

- define the drawing, using the graphic functions described in this chapter
- realize the PDF record by calling `EndPDF` which returns the PDF record, a string

When you call `BeginPDF(theRect)`, `theRect` specifies the coordinates `x` and `y` of the bottom left corner and the width and height of the PDF's frame:
`{x, y, theWidth, theHeight}`.

Example 83

```
BeginPDF({0, 0, 100, 100})
CirclePath({50, 50}, 2)
DrawPath(1)
set thePDF to EndPDF()
```

The string returned by `EndPDF()` is a PDF record: it begins with `%PDF`.

23.5.2 Producing a PDF file

Usually, to make a PDF file, you make a drawing in a graphic window as described above, then you save the window to disk. Alternately, you can make a PDF file without using a graphic window:

- make the additional PDF record as described just above
- write the PDF record to a new file using the standard `write` command
- set the type of the file to `PDF␣` using the Finder's dictionary

23.5.3 Appending PDF to a PDF file

To add some graphics to an existing PDF document proceed as follows:

- using the Open item of the File menu, open the PDF file. This will open a new graphic window.
- make the PDF record as described just above
- set the `front pdf` property of the graphic window to the PDF record
- force the display of the new data with `draw front pdf of theWind`, where `theWind` contains a reference to the graphic window
- save the graphic window

You can build the PDF by successive pieces: by default, setting the `front pdf` property (or the `back pdf` property) appends the new data to the existing data rather than replacing the existing data. To erase the previous data you must explicitly reset the property, e.g.:

```
set front pdf of theGraphicWindow to ""
set front pdf of theGraphicWindow to
thePDF
```

23.5.4 Setting the background or the foreground picture of a SmileLab plot

SmileLab is a library intended for making 2D and 3D representations of numerical data. Such SmileLab plots occur in graphic windows. To add some graphics into the background or the foreground picture of a graphic window proceed as follows:

- make the PDF record as described above in this section
- set the `front pdf` property (or the `back pdf` property) of the graphic window to the PDF record

- force the display of the new data with `draw front pdf of theWind` (or `draw back pdf of theWind`) where `theWind` contains a reference to the graphic window

- you may then save the graphic window

23.5.5 Displaying an animation in a graphic window

Your script can use a loop to generate successive PDF records as described above, with graphic commands bracketed between `BeginPDF(theRec)` and `EndPDF()`. To display successive PDF records as an animation, proceed as follows:

- before entering the loop, create a new graphic window

- in the loop, provide the PDF record to the graphic window as its `back pdf` property (you can use the `front pdf` as well). This will add the new drawing to the window without suppressing the previous one: do so, e.g., to draw a trajectory. To suppress the previous drawing before displaying the new one, reset the `back pdf` property to the empty string `""` before setting it to the new PDF record.

- in the loop, force the display of the new drawing with `draw back pdf of theWind`, where `theWind` contains a reference to the graphic window

`picture views`, described below, are often a better candidate to perform animations.

23.5.6 Displaying graphics in a picture view

The `picture view` is an element of the graphic window. You can have a `picture view` display a PDF record that was generated as described above, with graphic commands bracketed between `BeginPDF(theRec)` and `EndPDF()`. To have a `picture view` display a PDF record, proceed as follows:

- create a new graphic window
- create a new `picture view` in the graphic window
- set the `picture view`'s `frame` property, a list of four numbers. The two first numbers are the location of the bottom left corner with respect to the graphic window's `frame`. The two last items are the size of the displayed graphics. The PDF will be rescaled to fit that size.
- set the `picture view`'s `contained data` property to the PDF record
- refresh the display by calling `draw theWind`, where `theWind` contains a reference to the graphic window

23.5.7 Displaying animated graphics with picture views

The `picture view` can handle the following effects:

resize : to resize a `picture view` change the two last items of its `frame` property. The PDF will be scaled to fit the new size.

move : to move a `picture view` change the two first items of its `frame` property.

hide/show : change its `visible` property

To refresh the display use the `draw` verb. Usually you apply it to the graphic window, so as to refresh the full graphic. For specific purposes you can apply `draw` to the `picture view`: this will draw the new state of the `picture view` but will not erase its previous state.

Example 84

```
set theRect to {0, 0, 20, 20}
BeginPDF(theRect)
SetFillColor({0, 0, 0.75, 1})
RectPath(theRect)
DrawPath(0)
set thePDF to EndPDF()

set theGW to make new graphic window
set thePV to make new picture view at
end of theGW
set thePV's frame to theRect
set thePV's contained data to EndPDF()

set {x0, y0, x1, y1} to theGW's frame
set x to x0 + (random number of x1)
set {y, dy} to {y0 + y1, -1}
repeat while y > y0
    set thePV's frame to {x, y, 2, 10}
    draw theGW
    set {y, dy} to {y + dy, dy - 4}
end repeat
set thePV's frame to {x - 6, y0, 14, 3}
draw theGW -- splatz
```

23.5.8 Displaying graphics in a custom dialog

To display graphics in a custom dialog, use the PDF Holder dialog item provided in the Palette. The operation is described with more details in the chapter about custom dialogs (Chapter 15). Basically, proceed as follows:

- to have the PDF Holder display an existing PDF document, use the contextual menu on the PDF Holder in edit mode
- to have the PDF Holder display a PDF record which was produced as described in the present section, make a PDF record then set the «`class PDF␣`» property of the PDF Holder to the PDF record
- force the display of the new data with `draw theDialog` where `theDialog` contains a reference to the dialog that contains the PDF Holder.

23.6 Additional information and examples

23.6.1 How Smile's PDF engine really works

Smile's PDF engine is transparent and versatile. You can customize it in order to fit specific needs, provided you know how it works:

- the commands that you call belong to the *Graphic Kernel*, a library located in the *Class scripts/Context additions* folder. If you know what you are doing, you can make changes to that library and recompile it in a text window. Or, you can add your own libraries in the *Class scripts/Context additions* folder.

- when your script calls `BeginFigure` or `BeginPDF`, Smile essentially resets the global string stored in `my gstr` to the empty string `""`. These commands also reset some default settings, namely the shapes of the arrow and of the cross.

- each time your script calls a graphic command, Smile appends new lines to the global string `my gstr`. The contents of `my gstr` is really a source program for generating PDF, written in such a language as to support dynamical addition of commands — which is not the case for the regular PDF format.

- when your script finally calls `EndFigure` or `EndPDF`, Smile compiles the source program contained in `my gstr` into a PDF record: `EndFigure` will display it in the graphic window while `EndPDF` will simply return the PDF record (a string). To compile the source PDF program, those handlers call `makePDF`, a verb that belongs to Smile's dictionary.

- it was stated above that you can provide a PDF record as the `back pdf` property (or as the `front pdf` property) of a graphic window. Actually those properties also accept uncompiled source PDF programs, in other terms the string stored in `my gstr`.

- when you use the `back pdf` and `front pdf` properties of graphic windows, keep in mind that the graphics is really realized in the window only once you have called `draw` with those properties as its argument e.g. `draw back pdf of theWind`.

- it was described above how to save a PDF record as a file. Alternately, you can use `makePDF` to make directly an uncompiled source PDF program (not a PDF record) into a PDF file. With `makePDF` you can make a PDF file without creating any `graphic window`.

- unlike the `back pdf` and `front pdf` properties of graphic windows, the `«class PDF␣»` property of custom dialogs accepts only PDF records — that is, not uncompiled PDF source.

23.6.2 Additional resources

For more information, you should explore the source files:

- the basic functions are defined in the file *Smile folder:Class Scripts:Context additions:Graphic Kernel*. This file defines an interface to most of PDF graphic features.

- the *Smile folder/Class Scripts/Context additions/GeomLib* file provides an example of a 2D library for drawing 2D geometrical figures as can be found in school books. This file includes useful handlers intended for creating mathematical drawings.

Class Script/Context additions is a special folder for Smile. Any text or script file located inside this folder is loaded at startup, making the handlers that it contains available for scripting. Adding your own libraries into the *Class Script/Context additions* folder is how you customize your scripting environment.

Changes made to a text file located in the *Class Script/Context additions* folder will be effective immediately - provided you compile it once changed, while changes made to a script file located in the *Class Script/Context additions*

folder are effective when Smile is relaunched. Thus, if you want to develop a library, you should work on text files. You will compile them into script files when your project is final.

Chapter 24

Smile's folders

24.1 The roles of Smile's folders

To run properly, Smile requires to locate at launch time two special folders: *Class scripts* and *More stuff*. Those special folders contain themselves other special folders. Here is briefly what each special folder is for:

- *Class scripts*
  contains the class scripts of the objects of Smile, including the script of the application itself. The class scripts are described in section 17.3.

- *Class scripts/Context additions*
  script or text files stored in that directory are compiled into Smile's global context when Smile is launched. Storing a library as a text file in the *Context additions* folder is how you make it available to any script running from Smile.

- *More stuff*
  contains all the auxiliary files needed by Smile at one moment or another, such as for instance the Find dialog or the stationery applet.

- *More stuff/Initialization*
  script or text files stored in that directory are executed when Smile is launched. Storing a script in the *Initialization* folder is how you have a script execute at launch time.

- *More stuff/Documentation*
  text files stored in that directory are displayed in the Help menu of Smile — in addition to the Smile Help item — and can be opened via that menu.

- *More stuff/SmileLabTemplates*
  contains the default settings of the objects of SmileLab, stored as p-lists. Although those settings are not yet documented, you can edit most of them, since the p-list format includes labels: use a text editor such as Smile or use *Property List Editor*, an application included in Apple's *Developer Tools* package.

In addition to the *Class scripts* and the *More stuff* folders, if Smile locates a *User scripts* folder it will create an additional menu with the icon of a script, named the Scripts menu, which will display the contents of the *User scripts*. The Scripts menu can launch scripts, open text documents, application dictionaries and Smile dialogs. How

the Scripts menu works is described in Chapter 9.

24.2 Where Smile locates its folders

Smile recognizes three locations for its special folders:

- the shared Smile folder. Smile's shared folder is the folder that contains the double-clickable Smile application itself. As for Smile 2.5.2 Smile ships as a folder which contains Smile itself and the three special folders *Class scripts*, *More stuff* and *User scripts*. By default those are the special folders that Smile will consider.

- the user Smile folder. Smile creates a *Smile* folder in the *Library/Application Support/* user's directory: this folder is named the user Smile folder. The user Smile folder is a partial replica of the (shared) Smile folder which is located in the *Applications* folder. We qualify with "user" (or "shared") the folders located in the user (or the shared) Smile folder.

  - if you use the Worksheet, a text window that Smile saves automatically, Smile saves it into the user Smile folder.
  - when you quit Smile, Smile saves your personal settings and the permanent variables into a *Globals* file in the user *Class scripts* folder that gets created in the user Smile folder. (Regarding permanent variables, see section 11.7).
  - if a user *Class scripts* folder exists, Smile will consider the union of that folder with the shared *Class scripts* folder: it will load the contents of both folders. If a given file exists in both locations, only the file located in the user *Class scripts* is loaded.
  - if a user *Class scripts/Context additions* folder exists, Smile will load the files that it contains in addition to the files located in the shared *Class scripts/Context additions* folder. The shared *Class scripts/Context additions* folder is loaded first, so if a handler is found in both locations, the version located in the user directory hides the shared one.
  - if a user *More stuff* folder exists, then Smile will consider that folder instead of the shared *More stuff* folder. Smile considers only one *More stuff* folder, whose path is published as the `my gMoreStuffFolder` global variable.
  - if a user *More stuff/Initialization* exists, then Smile will execute the files of that folder after having executed the files of the shared *More stuff/Initialization* folder.
  - if a user *User scripts* folder exists, Smile uses that folder instead of the shared *User scripts* folder. Smile considers only one *User scripts* folder, whose path is published as the `user folder` application's property. Having a user *User scripts* folder allows the user to edit a user script and to make new user scripts particular to their user account. (See Chapter 9 about how user scripts work). You may want to duplicate the shared *User scripts* folder into the user Smile folder.

- Smile's package. You can move the *Class scripts* and *More stuff* folders into the Smile application's package. You open Smile's package by choosing Show package contents in the contextual menu of Smile in Finder. Smile expects to find the *Class scripts* and *More stuff* folders in the *Contents/Resources/* directory. As for Smile 2.5.2, Smile does not recognize a *User scripts* folder inside its package.

Chapter 25

If you are curious about Smile

25.1 The history of Smile

The history of Smile begins in '93 with SMI, the vision engine of Satimage, a French machine vision company. In order to be able to develop rapidly custom machine vision systems for various clients, Satimage develops an automation framework based on AppleScript. By '95 the framework starts to make sense in itself, and it is made publicly available for free: Smile (at first, "SMILE" for SMI Limited Edition) is born.

Today, Satimage-software, a department of Satimage, maintains Smile publicly available for free, publishes free additions for the advanced use of Smile, and makes additional Smile-based software for dedicated purposes: SMI (machine vision and industrial automation) for Satimage, soon SmileLab (data processing and graphics publishing) for the scientific users of Mac OS X, and more products in the future.

25.2 The philosophy of Smile

The core founders of Smile are physicists and they are familiar with the UNIX machines. When Apple first introduced AppleScript, they imagined how it could be if AppleScript was used from a shell window. AppleScript is such an advanced language and an open framework: an AppleScript owns a context, with persistent values, and AppleScript can send commands to software while they run, whence providing a much more interactive experience.

In 2003, the Smile platform is stable, powerful and polished. It is ready to become the environment of users concerned with productivity, to be the Swiss-army knife of the scientific OS X community, and to help more users switch to the Mac OS X.

25.3 Why Smile is free

Satimage-software makes a deal with the users of Smile as explicitly described at URL:
http://www.satimage-software.com/en/licensefree.html

- Satimage-software makes Smile available for free
- Satimage-software makes its best to improve frequently Smile and to suppress bugs in the shortest delays
- Satimage-software provides a (now famous)

Figure 25.1: Satimage-software welcomes feedback from all the users of Smile.

fast and professional technical support to Smile users

- the users of Smile report to Satimage-software their suggestions of improvement, and the bugs that they experience
- the users of Smile are not angry after Satimage-software when some new feature is not yet fully or not yet fully safely implemented

Part III

Appendices

Appendix A

Satimage regular expressions

A.1 Overview

Regular expressions define a syntax designed to perform complex search operations on text, such as searching for a class of characters instead of a specific character — e.g., digits. Introductions to the regular expressions can be found in various places on the World Wide Web. You will find a good introduction on *Script Meridian's site*.

Regular Expressions are also documented in the manual for the `grep` UNIX command: type `man grep` in a Terminal window, then read the section entitled "Regular expressions".

Alternately, use the Getman ... Smile's user script, which displays the manual in a standard text window.

Most often, the Regular expressions are for performing complex text searches. Various UNIX tools make use of Regular expressions. In Smile you can also use Regular expressions for performing text replacements.
Within Smile, you can use regular expressions, either in the Find (or Enhanced Find) panel, or by script, by using the `find text`, `change` and `re_compile` commands of the Satimage osax. Those commands are commented in the Chapter about Smile's general purpose library.

When you intend to use a regular expression from a script, use the Smile's Find (or Enhanced Find) panel to test and debug it.

A.2 Defining a search pattern

Using regular expressions consists in the first place in passing a special string as the search string. That special string, instead of its literal meaning, defines the pattern that will be searched for.

When used in a regular expression search pattern, most characters assume their literal meaning, and several characters take a special role: the metacharacters, described below in this section.

When you use the regular expressions in a script, you have to pass strings as AppleScript strings.
AppleScript strings imply to encapsulate the string between double quotes `"` and to "escape" two special characters, namely the double-quote

" and the backslash \. Escaping a character consists in prefixing it with backslash \. Thus: "\"" is the AppleScript string for double-quote, and "\\" is the AppleScript string for backslash. You may want to use the Make an AppleScript string user script (see section 12.3.1) when you have to make a string into an AppleScript string.

A.2.1 Metacharacters and "escape" character

To have a metacharacter (for instance, the bracket [) recover its literal meaning, you prefix it with backslash \. For instance, `[a-z]\[[0-9]\]` may match "c[8]".
In other cases, as you will see below, the character with the backslash is the metacharacter, while the character alone keeps its literal meaning.
Note also that those metacharacters which do not make sense inside brackets (the brackets define characters class, e.g. `[a-z]`, see below) recover their literal meaning inside the brackets. For instance, `[.]` and `[\]` stand for the period and for the backslash, respectively.

A.2.2 Anchors

You can use the characters below as tags which will stand for some specific kind of location in the text.

^ (hat): beginning of a line, or the beginning of the selection in a window, or the beginning of the text stored in a variable

$: end of a line, or the end of the selection in a window, or the end of the text stored in a variable.

Examples 85

```
change "^to be" into "" in "to be or
not to be" with regexp
-- " or not to be"
change "to be$" into "" in "to be or
not to be" with regexp
-- "to be or not "
```

This is the default behavior, see also section A.2.5 about flags which change the meaning of the tags above.

`\<` beginning of a word

`\>` end of a word

`\b` beginning or end of a word

`\B` strictly within a word

Example 86

```
find text "\\bbe" in "tobe or not to be"
with regexp
```

will match the last word.

A.2.3 Character classes

. stands for any character except CR (ASCII 13).

Example 87

```
find text "n(.*)" in "to be or not
to be" with regexp
```

will match the end of the line from "not".

This is the default behavior, see also section A.2.5 about flags which allow . to match CR (ASCII 13).

`[]` the brackets encapsulate the definition for a class of characters. For instance, `[0-9]` matches any digit.

`-` defines the range of characters which are within (considering the ASCII ordering) the characters on each side of the hyphen, for instance `[a-zA-Z]` matches any of the 52 uppercase and lowercase Roman letters

`^` defines a class by excluding the characters which follow the hat character.

Example 88

```
find text "[^@]*" in "homer@lol.com"
with regexp
```

(the meaning of the star `*` is explained below in section A.2.4)

`\w` any of the characters which are allowed in words

`\W` any of the characters which are allowed as word separators

`\r` CR, *carriage return*, ASCII 13

`\n` LF, *line feed*, ASCII 10

`\t` *tab*, ASCII 9

`[:alnum:]` pre-defined set, the Roman letters and the digits. The pre-defined sets work only when encapsulated within brackets. For instance, `^[[:alnum:]]{5}@` will match a set of exactly 5 alpha-numeric characters located at the beginning of a line and followed by "@".

`[:alpha:]` the Roman letters

`[:lower:]` the lowercase Roman letters

`[:upper:]` the uppercase Roman letters

`[:digit:]` the digits

`[:xdigit:]` the hexadecimal digits (lowercase and uppercase)

`[:blank:]` space or tab

`[:space:]` space, tab, CR, LF or FF

`[:cntrl:]` the set of the characters with an ASCII code < 32 or $= 127$

`[:punct:]` neither a control character nor alphanumeric

Examples 89

```
change "^" into "--" in selection of
window 1 with regexp
change "^[[:space:]]*--" into "" in
selection of window 1 with regexp
```

would comment out and uncomment, respectively, the block of text selected in the active window.

To include a literal `]` in a `[]` range place it first in the list.
To include a literal `^` place it anywhere but first.
To include a literal `-` place it last.
`\w` and `\W` are considered metacharacters only outside brackets `[]`.
`\r`, `\n` and `\t` are considered metacharacters inside and outside brackets, except when they just follow a backslash.
Thus, to match a literal backslash followed by an r `\r` (or to search for the sequence `\n` or `\t`) insert an additional backslash: search for `\\r`.

A.2.4 Operators

`*` zero or more occurrences of the preceding group, e.g. `^[[:space:]]*` will match any

combination of spaces and tabs at the beginning of a line

`+` one or more occurrences

`?` zero or one occurrence

`{i, j}` i to j occurrences, for instance `[0-9]{2,4}` will match a group of 2, 3 or 4 digits

`{i,}` i occurrences exactly or more

`{i}` i occurrences exactly

`|` or, e.g. `begin|end` will match either "begin" or "end"

`()` groups characters, e.g. `([0-9]{3},)+` may match "123,234,345,". You also use groups when you want to be able later to reference them:

`\1`, `\2` ... `\9` are references to the successive groups of the pattern. Those references can be used, either in the search string itself, or in the `using` parameter of `find text`, or in the `into` parameter of `change`: `^(.*)\r(.*\r)*\1$` — which, once written as an AppleScript string, reads `"^(.*)\\r(.*\\r)*\\1$"` — will match a block of text bracketed between two identical lines. While:

Example 90

```
find text "^(.*)\\r(.*\\r)*\\1$" in
someText with regexp using "\\1"
```

will match the same pattern, but will return only the duplicate line.

References may be very helpful with the `change` verb.

Example 91

```
change "([0-9]{2})/([0-9]{2})" into
"\\2/\\1" in someText with regexp
```

will change, e.g. "25/12" into "12/25".

The order of the groups is the order of the opening parentheses. If some group is repeated in the pattern, it finally stands for the last occurrence.

Example 92

```
find text "(^|[^0-9])(([0-9]{1,3}\\.)
{3}[0-9]{1,3})" in theText using
"\\2" with regexp, all occurrences
and string result
```

will return (as strings) the list of all dotted numeric IP addresses found in `theText`.

Example 93

```
find text "(^|[^0-9])(([0-9]{1,3}\\.)
{3}[0-9]{1,3})" in theText using
"\\3" with regexp, all occurrences
and string result
```

will return (as strings followed by a dot) the list of the third bytes of the dotted numeric IP addresses found in `theText`.

A.2.5 Flags

The dictionary of the Satimage osax states that the regular expressions support three flags: `"EXTENDED"`, `"NEWLINE"` and `"ICASE"`. The flags can only be used from a script. By default (thus, in the Find panel) the following flags are set: `{"EXTENDED", "NEWLINE"}`.

`"EXTENDED"` Set by default. Instructs to use the extended POSIX metacharacters set instead of the basic one. The basic metacharacters

set does not include `()`, `|`, `+` nor `?`. You may want to disable the `"EXTENDED"` flag if you are using a Regular expression which includes a variable: this way, the variable may contain those characters without confusing the Regular expression.

`"NEWLINE"` Set by default. Instructs to consider each line as a separate record. Namely, the period `.` does not match CR, and the anchors `^` and `$` match beginnings and ends of lines. If the `"NEWLINE"` flag is not set, the period `.` may match CR, and the anchors `^` and `$` match the beginning and the end of the text.
Searches performed with the `"NEWLINE"` flag disabled may require exponential amounts of memory and thus may be more prone to failures than searches performed with that flag set.

`"ICASE"` Not set by default. When set, uppercase and lowercase characters are considered the same. When you supply the `regexpflag` parameter, the setting for `"ICASE"` that you provide (implicitly or not) overrides the `case sensitive` parameter of Satimage osax' regular expression Suite.

When you use the `regexpflag` parameter for the `find text`, `change` or `re_compile` commands, you provide a list of flags, or an empty list. Those flags that belong to the list will be set, while those flags that the list does not contain will be implicitly unset.

Example 94

```
find text "end$" in "end" & return
& "end" regexpflag {"EXTENDED"} with
regexp
```

will match only the second word, since the `"NEWLINE"` flag is not set.

A.3 Defining a replace pattern

The replace string, such as entered in the Find panel or as the `into` argument of the `change` verb, can include the following metacharacters: `\1`, `\2` ... `\9` (the references to the groups which where defined in the search pattern) and the special characters `\r`, `\n` and `\t`.
The same characters are valid in the string passed as the `using` argument of `find text`. The `using` argument of `find text` supports strings and lists of strings — all strings recognize the metacharacters listed just above.

Example 95

```
find text "(.+) (.+)" in "Mickey Mouse"
using {"Dest:  Mr \\2", "Dear \\1,"}
with regexp and string result
-- {"Dest:  Mr Mouse", "Dear Mickey,"}
```

Appendix B

Portability and raw codes

B.1 What portability is about

Portability is a concern for scripts which involve some applications. Portability does not makes sense for scripts which involve only pure AppleScript commands, including the scripts which use terminology of the Scripting Additions.

Suppose your script runs perfectly on your machine, where it uses a scriptable application named “SurfWriter 4.7v5US”. Most probably it will include lines such as:

Example 96

```
tell application "SurfWriter 4.7v5US"
    -- your script here
end tell
```

When your script runs on a machine with a different version of the software, say, “SurfWriter 4.7v6”, or the same version with a different file name, say, “SurfWriter”, the script will probably — if you take no special precaution — request that the user specify, through a standard Open dialog, “Where is” the application it must use. This can happen at the first launch of the script on a new machine, or at each launch, depending on how the script is written. A script is considered portable if it never displays that dialog.

Also on your own machine, you may get the dreadful dialog if for some reason AppleScript is unable to retrieve “SurfWriter 4.7v5US”, which may happen if the application was not launched since long.

B.2 Referring to applications by creator code

A solution to the problem is to store in the script, not the application’s name, but its signature, which is common to all versions. The signature of an application is the same four-characters code as the creator of the files that it creates (“VIZF” for Smile).

The script will use Finder to retrieve the application being given its signature, and to launch it if necessary. Here is a short sample script which does so. It will be your responsibility to handle the error (number -1728) if Finder does not find any application with the requested signature, or if it cannot open it.

Example 97

```
set theCreator to "xxxx" -- the
signature of the application here
tell application "Finder"
    set theAppName to name of
application file id theCreator
    if theAppName is not in name of
every process then
        open application file id
theCreator
    end if
end tell
```

The launcher script above must run once.
If you do not know what the signature of the application is, launch it normally, then run:

Example 98

```
tell application "Finder" to get
creator type of process theName
```

where you will have replaced `theName` with the name of the application as it displays in the Dock.

The launcher script stores the application name into the variable `theAppName` (supposedly a global variable of your script). Subsequent calls to the scriptable application should look like:

Example 99

```
tell application theAppName
    -- script here
end tell
```

In some circumstances you may face conflicts due to the presence on your disk of a classic version of the application. In such a case, store the application file's path, not the application's name, in `theAppName`.

Example 100

```
tell application "Finder"
    set theAppName to (application file
id theCreator) as text
end tell
```

`tell application` accepts file paths as well as application names.

B.3 Why to use raw codes

Encapsulated in such a "virtual" wrapper (`theAppName` is not resolved when the script compiles), the script cannot use the keywords of the target application — so the script will not compile if it does include keywords that are specific to the target application.
The solution here is to use "raw codes". Raw codes are the canonical description of AppleScript's keywords (properties, classes, events, constants, etc.). Usually, the user sees only the 'English-like' terminology, and AppleScript uses the dictionary to translate the English-like terminology into raw codes.

For example, the canonical description of the `startup disk` property of the Finder is `«class sdsk»`.

B.4 How to get the raw codes

To get the raw codes, write (or copy) the script (using the usual keywords of the application) in a window connected to the target application of the script (see 10 about connecting a window to an application). Preferably, use a text window.
Then, select the text of the script in this

window. Select Copy translate (Scripting menu), then Paste inside the “virtual” wrapper of your script. The script will be pasted as raw codes. You can go back and forth between the raw codes in your script and the keywords in the window connected to the application.

An alternate way (proposed by jj, a Smile user) is to write and then save the script in a window connected to the target application. When you re-open the script, it will display the raw codes.

Appendix C

The components for custom dialogs

Below we present the different kinds of controls (named `dialog items` in Smile's dictionary) available in the Dialog components palette.
How to use those controls and build custom dialogs is described in the chapter devoted to Smile dialogs, Chapter 15.

For each kind of item we supply the following information:

- the value of the `control kind` property for that kind of item

- what user's action(s) will result in a notification to a script: the events that the item "detects"

- information specific to each kind of item

Most user's actions result in `click in` events.

The `click in` event is sent, not to the control that received the user's action, but to its container: the dialog window itself or possibly a container such as a Group Box.
The other events are sent to the control itself.

C.1 Push Button

`control kind` value: 368
Event(s): clicking sends `click in` to the control's container.
The OK and Cancel buttons of the Dialog components palette are special because of the special value of their `tag` property, "dflt" and "canc" respectively. They can be activated, by the *Enter* or *Carriage Return* key and by the *esc* key respectively.

C.2 Static Text Box

`control kind` value: 288
Event(s): clicking sends `click in` to the control's container.
The string that a Static Text Box displays is its `contained data` property.
The settings dialog's input field edits the string that the Static Text Box will display in the dialog window. The contextual menu offers two settings:

format: the formatting instructions for displaying real numbers.
Static text items support numbers as their `contained data`. You can specify the

format used to display numbers by supplying a specification string. The formatting string uses `#` for an optional digit, `0` for a required digit, `.` for the decimal point.

Example 101

```
###.000
```

will force 3 decimal digits, e.g. `2.500`. If the Static Text cannot display the number in the format specified (for instance the example above cannot be applied to numbers above `1000`) then the Static Text uses its standard display format, which is scientific notation.

The formatting string supports more advanced options, which are described in the entry about the `format` command, whose syntax is identical.

Justify the justification of the text, left, center or right

As described in section 15.7, the Static Text Box supports text coloring.

Example 102

```
set font of theStatic to font:-1,
color: 65535, 0, 0
```

C.3 Editable Text Box

`control kind` value: 272
Event(s): keystroke while focused sends `click in` to the control's container.
Like the Static Text Box, the contextual menu for the Editable Text Box allows you to set the format for real numbers using a specification string.

The control's `contained data` property contains the text displayed in the Editable Text Box.
By default, the control's `contained data` property returns Unicode text. To get the text as a regular AppleScript string, specify `as text` while getting the control's `contained data`.
By default, each keystroke in an Editable Text Box sends a `click in` event. There are cases when you would prefer to handle the user's entry only when completed. For instance, you do not want to send a "Wrong password" message 7 times.
If you want the Editable Text Boxes of a dialog window to send `click in` only when the user's text entry is completed, switch the dialog's «`class VaOE`» property to `true` (by default it is set to `false`).
When the «`class VaOE`» property of a dialog is set to `true`, the `click in` event is sent only when the user types *Enter* or *Tab* or when the user leaves the Editable Text Box, by clicking in another field for instance. The *Tab* key switches focus between the Editable Text Boxes, in the order of their increasing `index` values.

Like the Static Text Box, the Editable Text Box supports text coloring.

C.4 Password Text Box

`control kind` value: 274
Event(s): keystroke while focused sends `click in` to the control's container.
Works like the Editable Text Box, except that it displays only bullets. Its `contained data` property also contains bullets. The string as entered by the user is stored in the «`class pass`» property of the control.

C.5 Popup Menu Button

`control kind` value: 400
Event(s): menu selection sends `click in` to the control's container.
The list of the menu items, a list of strings that you can edit with the contextual menu, is the control's `menu` property. The index of the item selected is the control's `contained data` property.

C.6 Slider

`control kind` value: 51
Event(s): dragging the slider and releasing the mouse button send `click in` to the control's container.
The script is called during the mouse drag every time the slider is moved. If Live tracking is disabled in the contextual menu, the control detects only mouse up events: the script is called only once, when the user is finished moving the slider.
In the contextual menu you set the range of values assumed by the control's `contained data` property when the slider is moved from left to right, and the number of ticks displayed.

C.7 Little Arrows

`control kind` value: 96
Event(s): clicking sends `click in` to the control's container.
The control's `contained data` property reflects which part of the control was just clicked: `0` and `1` represent the upper and lower arrows respectively.

C.8 Radio Button

`control kind` value: 370
Event(s): clicking sends `click in` to the control's container.
A Radio Button must belong to a buttons family. A button family consists of a set of Radio Buttons with consecutive `index` values.
When the user clicks a radio button, Smile automatically blanks the other radio buttons in the same family. The `contained data` property of a Radio Button is an integer, `0` or `1`. You can also `get` it `as boolean`.

C.9 Check Box

`control kind` value: 369
Event(s): clicking sends `click in` to the control's container.
The `contained data` of a Check Box is an integer, `0` or `1`. You can also `get` it `as boolean`.

C.10 Time Clock

`control kind` value: 241
Event(s): clicking in the arrows or in the digits or typing in the digits while focused send `click in` to the control's container.
In the contextual menu, you can toggle two options on and off:

Display only suppresses the arrows and makes the control read-only.

Live makes the control continuously display the current time.

Changes to these options are effective the next time the object is initialized. Cutting then pasting the control, dragging it to a different container, or closing then re-opening the dialog are

ways re-initializing a control.
The time displayed by the control is the control's ≪class date≫ property. Only the time of the ≪class date≫ is relevant, the date of the day may be any date.
You may observe cosmetic glitches with the Time Clock dialog item, such as its display changing when you click a different dialog item. One solution is to explicitly set the text font and the text size of all concerned dialog items, and of the dialog itself, to the same value. For the dialog, set its `text font` and `text size` properties. For a dialog item, set its `font` property to a record with `font` and `text size` properties.

C.11 Date Clock

`control kind` value: 242
Event(s): clicking in the arrows or in the digits or typing in the digits while focused send `click in` to the control's container.
Works like the Time Clock, but only the date of the day of the ≪class date≫ property is relevant.

C.12 Progress Indicator

`control kind` value: 80
Event(s): none.
You set the Progress Indicator's limit values with the contextual menu, then your script sets its `contained data` in order to display the progress.
Changing the `contained data` of the Progress Indicator does not refresh the display until the script has finished executing. To really display a live progress bar, call `smilepause`, e.g. `smilepause 0`, from your script each time you change the Progress Indicator's state.

C.13 Chasing Arrows

`control kind` value: 112
Event(s): none.
Like the Progress Indicator, the Chasing Arrows dialog item requires calls to `smilepause` to be refreshed while a script is executing.

C.14 Visual Separator

`control kind` value: 144
Event(s): none.
When you resize the Visual Separator, you can make it an horizontal or a vertical line, depending on the shape you give to its boundary.

C.15 Disclosure Triangle

`control kind` value: 66
Event(s): clicking sends `click in` to the control's container.
In its default state, the triangle points to the right and the control's `contained data` is `0`. Once the user clicks it, the triangle points down, the control's `contained data` is `1`, and the window that contains the control gets enlarged downwards, by a quantity equal (in pixels) to the value of the control's ≪class MAXC≫ property. (This setting should have been available in the contextual menu, but it is not).

C.16 PDF Holder

`control kind` value: 256
Event(s): clicking sends `click in` to the control's container.
To have the control display a PDF document, use the contextual menu. The control displays

only the first page of PDF documents. In order to adjust the control's bounds to the PDF's bounds, you may want to use the Get pdf's size user script.
Instead of an existing PDF document, you can have the control display a PDF graphic that you create programatically, on the fly or once for all when you create the dialog.
Chapter 23 describes how to generate a PDF record by script and how to imbed it in the PDF Holder dialog item. Basically, you open the Graphic Kernel Quick Reference help file (Help menu), then you write a program using the handlers provided in that file, encapsulating the graphic commands with `BeginPDF ... EndPDF`. `EndPDF()` is the command that returns the PDF data. Finally you set the «`class PDF␣`» property of the PDF holder to the returned PDF.
For a sample, open the script of the PDF Holder provided in the Dialog components palette. The script contains routines which generates a (random) colored pattern on the fly.

C.17 Icon Control

`control kind` value: 323
Event(s): none.
You use the Icon Control to display an icon. The contextual menu offers a list of icons which should be available to Smile.
Displaying an icon consists in setting the value of the control's «`class ICON`» property to the resource index of a resource of type `"ICN#"` which is available to the program.
Smile includes commands to handle resources, see section 19.3.

C.18 Image Well

`control kind` value: 176
Event(s): clicking sends `click in` to the control's container, dragging from the control sends `export` to the control, dropping on the control sends `drop` to the control.
The Image Well displays icons. The mechanism is the same as for the Icon Control, except that the control's property which should contain a `"ICN#"` resource's number is (more consistently) its «`class ICN#`» property.
The user can drop things on the Image Well, provided you allow so. In the contextual menu, you can set the control to accept text (strings), files and/or Smile's objects. When the control is set to accept some kind of data and that kind of data is dragged over the Image Well the Image Well inverts its display to indicate that it can accept the data. If the user releases the mouse button, a `drop` event is sent to the Image Well's script:

Example 103

```
drop theThing onto theControl at
theLocation
```

`theThing` contains a reference to the object that was dropped `theControl` contains a reference to the control `theLocation` is meaningless.

You can set the control to accept other kinds of dropped data by setting its «`class flav`» property. The control's «`class flav`» property, a list of 4-character strings, specifies what kind of objects (what “flavors”) the control accepts. Usual flavors include:

- `"hfs␣"`: a file reference (for exemple, an icon from Finder)

- "long": an integer
- "doub": a real number
- "alis": an alias
- "reco": a record
- "TEXT": a string
- "obj␣": a reference to an object of Smile

The user can also perform drag-and-drop from the Image Well. When the user starts dragging the control, the control's script receives an `export theControl` event, provided the control was set to accept at least one kind of thing as described just above. If the control's script includes an `export` handler, and if the class of the result that the `export` handler returns matches one of the items of the control's «`class flav`» property, then the drag-and-drop will carry that quantity to the drop target. The drop target may belong to Smile, or to any another application (exporting file references is not fully implemented).

C.19 Bevel Button

`control kind` value: 32
Event(s): clicking sends `click in` to the control's container, dropping on the control sends `drop` to the control.
The Bevel Button works like the Image Well (above) except that you cannot drag from the Bevel Button.

C.20 List Box

`control kind` value: 352
Event(s): double-clicking sends `click in` to the control's container, dragging from the control sends `export` to the control, dropping on the control sends `drop` to the control. Drag-and-drop from the list into itself sends only `export`.

To change the contents of the list, use the contextual menu. The List Box displays the contents of its `contained data` property, which should be a list of strings. You can re-arrange the lines of the list by drag-and-drop.
The List Box control's container receives the `click in` event when the user double-clicks on some item of the List Box.
The user can export from the List Box and import into the List Box by drag-and-drop: the mechanism is the same as described above for the Image Well.

C.21 Menu Group Box

`control kind` value: 162
Event(s): a menu selection sends `click in` to the control's container Works like the Popup Menu Button, except that the Group Box may contain other dialog items.

C.22 Group Box

`control kind` value: 160
Event(s): none.
May contain other dialog items.

C.23 Tabs Holder

`control kind` value: 128
Event(s): clicking in one of the tabs sends `click in` to the Tabs Holder. The Tabs Holder is really

a container for tab items: for instance the Preferences dialog contains one Tabs Holder at the first level: that Tabs Holder contains three tab items named General AppleScript and Windows.
Smile creates and names the tab items of a Tabs Holder when it creates the Tabs Holder, after the value of the control's «`class tab#`» property. The «`class tab#`» property specifies how many tab items will be created, and what their names are.
To create, remove or rename a tab item, edit the Tab Holder's «`class tab#`» property, then force it to be created again, for instance Cut then Paste.
Each tab item works like a regular Group Box dialog item. To edit a tab item, the dialog must be in edit mode. To select a different tab item, toggle the dialog into normal mode (use the Edit mode item of the Edit menu) then click the desired tab item, then toggle back into edit mode.

Appendix D

The dictionary of Smile

D.1 Smile

`prepare`
reference – the newly created object

event sent by the application when an object is created

`do menu`
small integer – the integer code of the command
`to` reference – the target of the command

event sent by the application when a menu item is selected

`click in`
reference – the object
`item number` small integer

event generated by a click in an object

`drop`
anything – the dropped object
`onto` reference – the destination object
[`at`] point – the drop coordinates

event generated by dropping

`export`
reference – the object

Result
anything

return a description for the object to export

`store`
reference – the object being saved

sent by the application just before saving an object

`draw`
reference

draw an object

`execute`
reference – the script window
[`as`] type class – default : return the raw result

Result
string – result of the script

run the script of a script window

`check syntax`
reference – the script window

check syntax of a script window

`postit`
string

`localize`
string
[`encoding`] small integer

Result
string

(advanced) returns the localized string as found in the "Smile.localized.strings" file of the current ".lproj" folder

`notify`
reference – the recipient object
[`from`] reference – the sender
`with data` anything – the message
[`with delay`] small real – seconds

a general mechanism intended for sending messages from an object to another.

`Class basic object`

Plural form
`basic objects`

Properties
`class` type class – [read only]
`name` string
`id` integer – the unique id number [read only]
`container` reference – the object it belongs to [read only]
`named reference` reference – a reference by name [read only]
`bounds` bounding rectangle
`path name` file specification
`visible` boolean
`drawing` boolean – does the object draw its result ?
`class script` script – the script shared by all objects of the same class
`script` script – the personal script of the object
`extras` anything – any user data
`extension file` string – the name of the file containing the external code for the object
`current dialog` integer – the id number of the settings dialog of the object
`want idle` boolean – does its script receive an 'idle' callback on idle events ?
`idle delay` small real – delay between idle events in seconds
`properties` record – the properties of the object
`whole` record – the properties and elements of the object [read only]

generic class which has the properties owned by each object

`Class application` (inherits from basic object)

Properties
`creator type` type class – [read only]
`cursor` arrow/watch/busy

`screen bounds` bounding rectangle – [read only]
`user folder` file specification – the folder related to the Scripts menu [read only]
`user script file` file specification – the currently running script file [read only]
`context` script – the class script of the basic object class [read only]
`globals` script – the script of the permanent global variables [read only]
`dictionary` string – the dictionary of the application [read only]
`modifiers` list of command down/option down/control down/shift down/caps lock downs – [read only]
`clipboard` anything – can contain text, references etc.
`recording` boolean – toggled to record scripts
`console` reference – the text window for recording
`chrono` small real – the time elapsed (in seconds) since the last "chrono" call
`mouse location` point
`mouse button` boolean
`background` boolean
`serial ports` list of strings – A list of info for each serial device. This info is a list {kind, UNIX path, name}. kind=9 means RS232. [read only]

Elements
`window` by numeric index, name, id
`text window` by numeric index, name, id
`script window` by numeric index, name, id
`graphic window` by numeric index, name, id
`dialog` by numeric index, name, id
`IO device` by numeric index, name, id
`menu` by numeric index, name
`menu command` by numeric index

the application program

`Class window` (inherits from basic object)

Properties
`text font` string – the name of the font or its id number
`text size` small integer
`width` small integer – the width
`height` small integer – the height
`resource id` small integer – the id number of the resource containing the definition of the window
`message height` small integer – the height of the button bar in a text window or of the message bar in a video window
`collapsed` boolean
`closeable` boolean – Does the window have a close box?
`resizable` boolean – Is the window resizable?
`zoomable` boolean – Is the window zoomable?
`modified` boolean – Is the window modified
`message bar` message bar – the text field in a video window

Elements
`agent` by numeric index, name, id

generic window

`Class agent` (inherits from basic object)

Plural form
`agents`

Properties
`active` boolean
`permanent draw` boolean – does it still draw

if not active ?
`call script` boolean – does it send a 'post process' callback to its script once its job is over ?

provides a specific functionality to a window

`Class text window` (inherits from window)

Plural form
`text windows`

Properties
`selection` list of integers – the selection range, or the selected text (as text)
`line width` small integer
`fit to window` boolean – adjust text to window width
`tab width` small integer
`scripting language` string – the default scripting language
`console` reference – the text window for output (default : the same window)
`store undo` boolean – true-false to encapsulate complex operations avoiding ridiculous undo's
`update screen` boolean – false-true to encapsulate complex operations avoiding lengthy text calculations

Elements
`character` by numeric index, relative position, range, test
`word` by numeric index, relative position, range, test
`paragraph` by numeric index, relative position, range, test
`text` by numeric index, relative position, range, test
`run info` by numeric index, relative position, range, test

`Class script window` (inherits from text window)

Plural form
`script windows`

`Class text`

Properties
`text size` small integer
`text font` string – the font name or index
`text color` RGB color – a list, e.g. {0,0,0} for black
`style` text style info
`length` integer
`index` integer – the index of the first character of the text in its window
`boundaries` list of integers – the text range as a list of 2 integers
`paragraph index` integer
`word index` integer

`Class dialog` (inherits from window)

Plural form
`dialogs`

Properties
`contained data` record – the contents of the dialog items, by keyword
`modal` boolean – does the dialog have to be closed before any new user action ?
`focus` reference – the active item
`mode` boolean – is the dialog in edit mode?
`owner` reference – (advanced)

Elements
`dialog item` by numeric index, name

`Class dialog item` (inherits from basic object)

Plural form
`dialog items`

Properties
- `enabled` boolean
- `contained data` anything – contents of the item
- `control kind` small integer – the control type as in appearance manager
- `call script` boolean – does it trigger a "click in" call to the script of the dialog?

`Class dialog list item` (inherits from dialog item)

Properties
- `selection` list of small integers – the indices of the selected items

an item of a dialog which displays a list

`Class menu`

Plural form
- `menus`

Properties
- `name` string
- `enabled` boolean

Elements
- `menu item` by numeric index, name

`Class menu item`

Plural form
- `menu items`

Properties
- `name` string
- `enabled` boolean
- `checked` boolean
- `modifiers` list of command down/option down/control down/shift down/caps lock downs
- `shortcut` string

`Class menu command` same as menu item, but access is by command id

`Class IO device` (inherits from basic object) I/O peripheral

`Class RS232` (inherits from IO device)

Properties
- `configname` string – UNIX path to the serial port (as provided in the serial ports Smile's property)
- `RSOptions` RSOptions
- `enabled` boolean
- `contained data` string – data to send or data received

RS232 device

`Class RSOptions`

Properties
- `bauds` small integer
- `databits` list of small integers – data bits count (5, 6, 7 or 8)
- `stopbits` small integer – 1: send one stop bit, 2: send two stop bits
- `parity` small integer – 0: disabled, 1: enabled, 2: odd parity
- `flowcontrol` small integer – 0: none, 1: outbounds CTS, 2: inbounds DTR, 3: enable input and output flow control

Options for RS232

D.2 Misc

Miscellaneous Events

`do script`
string – the script
[`as`] type class – wanted type for the result

Result
anything

execute a script

`cut`
reference

`copy`
reference

`paste`
reference

`undo`
reference

`reveal`
reference

Bring the specified object(s) into view

D.3 Satimage utilities

`display`
anything

Result
string

return the direct object as a string

`smilepause`
real – the timeout in seconds

`remote info for`
alias – the file

Result
a list of string – {the appletalk zone,the server name,the volume}

locate an alias on the network

`extractcolumn`
small integer – the column index
[`thru`] small integer – the last column
`in` anything – a file, a string or an array of real
[`as`] type class – requested type for the result
[`skipping`] small integer – number of lines to skip

Result
string – the column

`throwerror`
string – the error string
[`number`] small integer – the error number
[`partial result`] anything
[`from`] reference – the offending object

Result
anything

same as error, but faster

`find definition for`
string
[`in`] alias – (list of) file or folder, default : scripting additions folder
[`as`] type class – default : return the definition as styled text

Result
anything

`converttext`
  string – the string to convert
  `from` string – the initial encoding
  `to` string – the requested encoding

Result
  string – the converted string

`textencodings`
  [`as`] type class – string or integer, default : string

Result
  a list of string – the available text encodings

D.4 Smile drawings Suite

`makePDF`
  string – the pdf description provided by Graphic Kernel
  [`in`] anything – write directly into this file
  `media box` bounding rectangle

Result
  string – the PDF data

`addPDF`
  string – the pdf data or file
  `in` reference – a reference to a window
  `at` anything – a point or a rect

`Class graphic window` (inherits from window)

Plural form
  `graphic windows`

Properties
  `frame` list of small reals – {x origin,y origin,width,height}, the page frame. Values are real numbers. Unit = 1/72 inch (1 pixel). Prefer pageheight and pagewidth
  `pageheight` small real – Unit = 1/72 inch. Can be set in inches or centimeters
  `pagewidth` small real – Unit = 1/72 inch. Can be set in inches or centimeters
  `grid` list of small integers – a list of 2 integers, default is {1,1}. These nummbers are used to provide default frames to the graphic views. The first (resp. second) number is the number of expected views horizontaly (resp. verticaly)
  `back pdf` list of strings – The pdf data for the background of the window. Can be set to a file, to some Graphic Kernel output or to raw pdf data as string.
  `front pdf` list of strings – The pdf data drawn after the background and the graphic views of the window. Can be set to a file, to some Graphic Kernel output or to raw pdf data as string.
  `title offset` list of small reals – vertical offset for view's titles

Elements
  `graphic view` by numeric index, name

a window where you can draw pictures of various kinds by script, and that you can save as a pdf file or as a tiff file.

`Class graphic view` (inherits from basic object)

Properties
  `frame` list of small reals – {x origin,y origin,width,height}. Defines the rectangular region which will be erased when the graphic object is redrawn. The rectangle is relative to the origin of the graphic window. Values are real numbers. Unit = 1/72 inch (1 pixel)

a virtual class, the common ancestor for all the classes of objects that you may create in a

graphic window.

D.5 SmileLab Suite

Graphic presentation of numerical data. Unless otherwise stated, lengths are real numbers, and the length unit is 1/72 inch (1 pixel)

`HSV2RGB`
list of small real – {hue,saturation,value}

Result
a list of small real – {red,green,blue}

color translation

`choose color`
list of small real – {red=0..1,green,blue}

Result
a list of small real – {red=0..1,green,blue}

choose a color with the colr picker

`Class picture view` (inherits from graphic view)

Properties
`contained data` string – Quartz data. See the documentation of Smile's graphic engine for more information.

`Class plot` (inherits from graphic view)

Plural form
`plots`

Properties
`plot frame` list of small reals – {x origin,y origin,width,height}, the rectangle enclosing the curves. Values are real numbers. Unit = 1/72 inch (1 pixel)
`limits` list of small reals – {xmin,xmax,ymin,ymax}, the limit values for the x and y axis
`text font` string – the name of the font
`text size` small real
`pen color` list of small reals – {red=0..1,green,blue,alpha=0..1}, alpha=1 (opaque drawing) as of MacOS 10.1
`fill color` list of small reals – {red=0..1,green,blue,alpha=0..1}, alpha=1 (opaque drawing) as of MacOS 10.1
`grid color` list of small reals – {red=0..1,green,blue,alpha=0..1}, alpha=1 (opaque drawing) as of MacOS 10.1
`grid dash` list of small reals – {Lstart,Lstr1,Lsp1,..,Lstrn,Lspn}. Dash starts at Lstart and draws n sequences of stroke (Lstr) + space (Lsp). For instance use Lstart = Lstr1 to have dash start at beginning of first space.
`grid pen width` small real
`major tick length` small real
`minor tick length` small real – (enter a negative value to have the ticks point outwards)
`log xaxis` boolean – Is the x axis logarithmic? Default false.
`log yaxis` boolean – Is the y axis logarithmic? Default false.
`grid` reference – use "grid" only with the "draw" verb to have the grid redraw before drawing the set of curves
`xlabel` string – text of label for x axis. Texts of labels support TeX conventions. For instance "\\p" will display the greek pi

letter, "a^n" (resp. "a_n") will display n as a superscript (resp. subscript).
`xlabel offset` small real – vertical offset of label for x axis
`ylabel` string – text of label for y axis. Texts of labels support TeX conventions. For instance "\\p" will display the greek pi letter, "a^n" (resp. "a_n") will display n as a superscript (resp. subscript).
`ylabel offset` small real – horizontal offset of label for y axis
`label text font` string – the name of the font. If the specified font is not available, the default font is used instead.
`label text size` small real
`legend frame` list of small reals – {x origin,y origin,width,height}. Values are real numbers. Unit = 1/72 inch (1 pixel)
`legend text font` string
`legend text size` small real
`legend pen width` small real – pen width for the legend frame
`legend fill color` list of small reals – {red=0..1,green,blue,alpha=0..1}, alpha=1 (opaque drawing) as of MacOS 10.1

a virtual class, the ancestor for curveplot, contourplot, vectorplot and imageplot

`Class curveplot` (inherits from plot)

Plural form
`curveplots`

Properties
`legend sample length` small real
`legend on curve` boolean – true, displays the curves' names on the curves, at the abscissa provided as the "legend abscissa" property
`legend abscissa` small real – effective only if "legend on curve" is true

Elements
`curve` by numeric index, name

use the curveplot to display 1-d curves. curveplots are the containers for curves.

`Class curve` (inherits from basic object)

Properties
`line style` small integer – 0 none, 1 line, 2 smooth. Smoothing makes more sense if the curve really represents some f(x) function.
`pattern style` small integer – 0 none, 1 circle, 2 square, 3 diamond, 4 upwards triangle, 5 downwards triangle, 6 x-cross, 7 cross, 8 point, 9 custom
`custom pattern` list of small reals – {x1,y1,..,xn,yn}, coordinates of the polygon which will be used as the pattern (effective only is "pattern style" is set to 9)
`pattern size` small real – size of the pattern if "pattern style" is not 0
`pen color` list of small reals – {red=0..1,green,blue,alpha=0..1}, alpha=1 (opaque drawing) as of MacOS 10.1
`fill color` list of small reals – {red=0..1,green,blue,alpha=0..1}, alpha=1 (opaque drawing) as of MacOS 10.1
`pen width` small real
`dash` list of small reals – {Lstart,Lstr1,Lsp1,..,Lstrn,Lspn}. Dash starts at Lstart and draws n sequences of stroke (Lstr) + space (Lsp). For instance use Lstart = Lstr1 to have dash start at beginning of first space.
`formula` string – any function of the x variable, for instance "sin(x)". Check the

Satimage osax dictionary regarding the available mathematical functions. Set the formula to the empty string to suppress it.
`step` small real – the distance between two consecutive x values where the formula will be computed. By default "step" is 0 and SmileLab computes the formula at 20 equidistant points.
`xdata` list of real – the list of the x values
`ydata` list of real – the list of the y values
`contained data` list of small reals – Obsolete. The x & y values as a list of two lists of equal length {{x1,..,xn},{y1,..,yn}} (effective only if "formula" is set to the empty string)
`antialiasing` boolean – default true. Plots made of a huge numbers of points (such as Poincar maps) may be nicer if "antialiasing" is set to false.
`in legend` boolean – Is the curve displayed in the curveplot's legend box? Default true.

a curve may plot, either an explicit function provided as its "formula", or a set of points provided as its "contained data"

`Class contourplot` (inherits from plot)

Properties
`userzlimits` boolean – true: use zmin and zmax, false: auto compute them
`zmin` small real – (effective only if "userzlimits" is set to true)
`zmax` small real – (effective only if "userzlimits" is set to true)
`level number` small integer – the number of contours
`color palette` list of small reals – a list of 4*n real numbers, {red0, green0, blue0, alpha0,...}, n is at most 256. Default is a rainbow palette.
`xdata` matrix – either the list of the x values, or the full 2D array of the x values, formatted as a matrix i.e.: {nrows:i,ncols:j,array of real:thedata}. If "xdata" is empty, the positive integers are used as the default x values.
`ydata` matrix – either the list of the y values, or the full 2D array of the y values, formatted as a matrix i.e.: {nrows:i,ncols:j,array of real:thedata}. If "ydata" is empty, the positive integers are used as the default y values.
`zdata` matrix – the 2D array of the z values, formatted as a matrix. A matrix is a record formatted as follows: {nrows:i,ncols:j,array of real:thedata}.

use the contourplot to display contours of a z(x,y) surface

`Class vectorplot` (inherits from plot)

Properties
`arrow def` list of small reals – {smallLength,overallLength,overallWidth}, defines the shape of the arrow
`vector scaling` small real
`xdata` matrix – either the list of the x values, or the full 2D array of the x values, formatted as a matrix i.e.: {nrows:i,ncols:j,array of real:thedata}. If "xdata" is empty, the positive integers are used as the default x values.
`ydata` matrix – either the list of the y values, or the full 2D array of the y values, formatted as a matrix i.e.: {nrows:i,ncols:j,array of real:thedata}. If "ydata" is empty, the positive integers are used as the default y values.
`vxdata` matrix – the 2D array of the x-coordinates of the vector field, formatted

as a matrix. A matrix is a record formatted as follows: {nrows:i,ncols:j,array of real:thedata}.
`vydata` matrix – the 2D array of the y-coordinates of the vector field, formatted as a matrix. A matrix is a record formatted as follows: {nrows:i,ncols:j,array of real:thedata}.

use the vectorplot to display a vector field

`Class imageplot` (inherits from plot)

Properties

`zdata` matrix – the 2D array of the z values, formatted as a matrix. A matrix is a record formatted as follows: {nrows:i,ncols:j,array of real:thedata}.
`userzlimits` boolean – true: use zmin and zmax, false: auto compute them
`zmin` small real – (effective only if "userzlimits" is set to true)
`zmax` small real – (effective only if "userzlimits" is set to true)
`color palette` list of small reals – a list of 4*n real numbers, {red0, green0, blue0, alpha0,...}, n is at most 256. Default is a gray palette.
`inverted` boolean – inverts the color palette

use the imageplot to visualize a 2D array of real numbers as a bitmap image. Default palette is gray.

`Class plot3D` (inherits from graphic view)

Properties

`frame` list of small reals – {x origin,y origin,width,height}. Defines the rectangular region which will be erased when the surface is redrawn. The rectangle is relative to the origin of the graphic window. Values are real numbers. Unit = 1/72 inch (1 pixel)
`eye position` list of small reals – {x,z,y}, note the special ordering of the coordinates (inherited from the OpenGL conventions)
`light position` list of small reals – {x,z,y}, note the special ordering of the coordinates (inherited from the OpenGL conventions)
`projection` list of small reals – {left,right,bottom,top,near,far}, defines the cube which is used as the orthographic parallel viewing volume to perform the 3D view (in the OpenGL framework, "projection" is glOrtho)
`rotation` list of small reals – {angle,vx,vz,vy}, defines an optional rotation of the surface (in the OpenGL framework, "rotation" is glRotatef). angle is in degrees (use with caution)
`legend frame` list of small reals – {x origin,y origin,width,height}. The frame for the color scale. Values are real numbers. Unit = 1/72 inch (1 pixel). Set to {0,0,0,0} to suppress the color scale.
`userlimits` list of booleans – a list of 4 booleans; true: use min and max given by limits, false: auto compute them
`limits` list of small reals – {xmin,xmax,ymin,ymax,zmin,zmax,colmin,colmax}, the limit values for the x, y, z and color
`xdata` matrix – either the list of the x values, or the full 2D array of the x values, formatted as a matrix i.e.: {nrows:i,ncols:j,array of real:thedata}. If "xdata" is empty, the positive integers are used as the default x values.
`ydata` matrix – either the list of the y values, or the full 2D array of the

y values, formatted as a matrix i.e.: {nrows:i,ncols:j,array of real:thedata}. If "ydata" is empty, the positive integers are used as the default y values.

`zdata` matrix – the 2D array of the z values, formatted as a matrix. A matrix is a record formatted as follows: {nrows:i,ncols:j,array of real:thedata}.

`colordata` matrix – the 2D array of the z values, formatted as a matrix. A matrix is a record formatted as follows: {nrows:i,ncols:j,array of real:thedata}.

`drawaxes` boolean

`xperiodicity` small integer – 0 non periodic, 1 data are periodic with period ncols-1, 2 data are periodic with period ncols

`yperiodicity` small integer – 0 non periodic, 1 data are periodic with period nrows-1, 2 data are periodic with period nrows

`orientation` boolean – surface orientation. Default is true. SmileLab renders the outer/upper side of the surface as a shining surface, and its inner/down side as a dull surface. Depending on how the surface is parametrized, you may want to inverse the default orientation.

`xlabel` string – text of label for x axis. Texts of labels support TeX conventions. For instance "\\p" will display the greek pi letter, "a^n" (resp. "a_n") will display n as a superscript (resp. subscript).

`xlabel offset` small real – vertical offset of label for x and y axis

`ylabel` string – text of label for y axis. Texts of labels support TeX conventions. For instance "\\p" will display the greek pi letter, "a^n" (resp. "a_n") will display n as a superscript (resp. subscript).

use the plot3D to display a realistic rendering of a surface. SmileLab implements an orthographic parallel viewing.

Appendix E

The dictionary of the Satimage osax

E.1 Satimage text Additions

mailto:support@satimage-software.com

`find text`

string – the substring to search for (or a result of re_compile for advanced use of regexp)
`in` string – if ”find text” is called from Smile, can be a reference to a window, or a reference to a range of text in a window
[`case sensitive`] boolean – default true
[`regexp`] boolean – default false
[`whole word`] boolean – default false
[`regexpflag`] list of string – a subset of {”EXTENDED”,”NEWLINE”,”ICASE”}; default {”EXTENDED”,”NEWLINE”}
[`using`] string – the pattern to generate the returned string (needs regexp true)
[`all hits`] boolean – returns a list of all hits instead of the first one only. Default : false
[`string result`] boolean – return only the matching string

Result
record – {matchLen: length of the match, matchPos: offset of the match, matchResult: the matching string}. The matching string may be formatted according to the ”using” parameter. If ”string result” is true, only the matching string is returned instead of the record.

searches a given string, or a given regular expression pattern (see Appendix A for the documentation about regular expressions), in a string. If called from Smile, ”find text” supports as its ”in” parameter — in addition to strings — a reference to a window of Smile, or a reference to any range of text in a window of Smile. Example:

```
change ":" into "/" in first paragraph
of selection of window 1
```

Regarding how to describe a range of text of a window, see Chapter 13 about Smile’s Text Suite.

`change`

string – the substring to search for (or a result of re_compile for advanced use of regexp)
`into` string – the replacement string
`in` anything – a string or a reference to a

file (alias). If "change" is called from Smile, can be a reference to a window, or a reference to a range of text in a window
[`case sensitive`] boolean – default true
[`regexp`] boolean – default false
[`whole word`] boolean – default false
[`regexpflag`] list of string – default {"EXTENDED","NEWLINE"}

Result
anything – the new string if the "in" parameter is a string or a reference to a file, otherwise the list {number of hits, offset of the last replace}

replace all occurrences of a literal substring or of a regular expression pattern. If the "in" parameter is a reference to a file, the file remains unchanged and "change" returns the new string.

`re_compile`
string – the regular expression
[`case sensitive`] boolean – default true
[`regexpflag`] list of string – a subset of {"EXTENDED","NEWLINE","ICASE"}; default {"EXTENDED","NEWLINE"}

Result
re_pattern – an opaque pattern which may be used as the search string for "find text" and "change"

compile a regular expression. The result can be supplied as the direct parameter for "find text" and "change".

`extract string`
string – the original string
[`from`] integer – index of the first character. Default : 1. Negative numbers index characters backwards.
[`to`] integer – index of the last character. Default -1. Negative numbers index characters backwards.

Result
string – the substring

extract a substring out of a string. Same as AppleScript's expression "text i thru j of s", but used to be safer.

`uppercase`
string – the original string

Result
string – the uppercase string

move to uppercase. Handles accented characters.

`lowercase`
string – the original string

Result
string – the lowercase string

move to lowercase. Handles accented characters.

`convert to Windows`
string – the original string

Result
string – the converted string

converts a Mac string into a Windows string

`convert to Mac`
string – the original string

Result
string – the converted string

converts a Windows string into a Mac string

`format`
real – the number
`into` string – the formatting string, using `#`,ˆ,O,.,`%`,',(,),+,-

Result
string – the formated number

format a real number using a specification string. Ex: format pi into "`##.##`"->"3.14". "0" instead of "`#`" forces trailing zeros. "`^`" adds a space. "+f1;-f2;f3" provides formats for numbers >0, <0, =0. Encapsulate custom strings with "".

E.2 Satimage files Additions

`alias description for`
alias – the remote item

Result
a list of string – {the AppleTalk zone name, the server machine name, the server volume name, folder name, [], item name}

provide info needed to refer to a remote item

`navchoose file`
[`with prompt`] string – a prompt to be displayed in the file chooser
[`of type`] list of string – restrict the files shown to only these file types
[`starting at`] alias – the default file or folder
[`multiple files`] boolean – allow multiple files selection (default true)
[`show packages`] boolean – (default true)
[`open packages`] boolean – (default false)

Result
a list of alias – the chosen files

choose file with navigation services

`navchoose folder`
[`with prompt`] string – a prompt to be displayed in the folder chooser
[`starting at`] alias – the default folder
[`open packages`] boolean – (default false)

Result
a list of alias – the chosen folders

choose folder with navigation services

`navchoose object`
[`with prompt`] string – a prompt to be displayed in the folder chooser
[`starting at`] alias – the default folder
[`show packages`] boolean – (default true)
[`open packages`] boolean – (default false)

Result
a list of alias – the chosen folders

choose file or folder with navigation services

`navchoose volume`
[`with prompt`] string – a prompt to be displayed in the folder chooser
[`starting at`] alias – the default folder

Result
a list of alias – the chosen folders

choose volume with navigation services

`navask save`
[`file name`] string – name of the file
[`action`] small integer – 1 on close, 2 on quit, 0 ?

Result
small integer – 1 save, 2 cancel, 3 don't save

prompt for save

`navchoose file name`
[`with prompt`] string – the text to display in the file creation dialog box
[`default name`] string – the default name for the new file
[`with menu`] list of string – list of menu items
[`starting at`] alias – the default folder
[`open packages`] boolean – (default false)

Result
file specification – the file the user specified

Get a new file specification from the user, without creating the file. Uses navigation services

`navnew folder`
[`with prompt`] string – the text to display in the file creation dialog box
[`starting at`] alias – the default folder
[`open packages`] boolean – (default false)

Result
file specification – the folder the user specified

Get a new folder specification from the user. Uses navigation services

`list files`
alias – a folder
[`recursively`] boolean – default: true
[`invisibles`] boolean – default: false

Result
a list of alias

the list of the files contained in the folder. By default, the list includes the files located in nested folders.

E.3 Satimage utilities Suite

Miscellaneous utilities.

`backup`
file specification – the source folder
`onto` file specification – the destination folder
[`level`] small integer – 0: report only, 1: synchronize folders, 2 : synchronize and report. Default 0.
[`after`] date – files older than this date are not processed.
[`recursively`] boolean – recursively synchronize subfolders. Default true.

Result
string – the (optional) report

synchronizes 2 folders. The aliases located at the first level of the source folder and of the destination folder are resolved, the aliases located deeper are not. Thus, you can choose to fill the

source folder with aliases to the original folders that you want to backup. In the destination folder, you will put aliases to the copies, that need to be synchronized, supplying to each alias the same name as the corresponding alias to an original folder.

`special concat`
record – the record
`with` record – the additional data

Result
record

concatenate {a_ppty:X, } and {a_ppty:Y, } into {a_ppty:Z, }, where Z is X & Y (resp. X + Y) if X,Y are lists (resp. numbers). Can also be used to append a new column to an array given in text format. The direct parameter should be a string representing an array with tab-delimited columns and return-delimited rows. The "with" parameter is the column to append: it is a return-delimited string. "special concat" will return, still as a string, the array with the new column appended.

`suppress item`
anything – the rank or key of the item. Use quotes around custom properties, and also around 4-characters codes. (If you don't know what this means, you don't need it).
`from` anything – a list or a record

Result
record

delete an item from a list or a record.

E.4 Resource Suite

Utilities to read and write resources from/to a file.

`load resource`
small integer – index of the desired resource
`type` type class – type of the desired resource
`from` file specification – file to read from
[`as`] type class – an AppleScript type for the returned result

Result
anything – any AppleScript data that is stored in the resource: data, object specification, reference, etc.

get the resource of the given type and id from the specified file

`list resources`
type class – type of desired resources
`from` file specification – file to read from

Result
anything – the list of ids

return the list of the ids of the resources of the specified type stored in the specified file

`get resource name`
small integer – index of the desired resource
`type` type class – type of the desired resource
`from` file specification – file to read from

Result
anything – the name of the resource

return the name of the resource of the specified type and id from the specified file

`put resource`
anything – the AppleScript data that will be stored in the resource
`to` file specification – the destination file
`type` type class – the resource type
`index` small integer – the resource id
[`with name`] string – the resource name

write the given resource to the specified file with specified type and id

E.5 Math

Some mathematical functions. Most functions accept as their direct parameter (and return) a list or an array of real. Notice: you may need more parenthesis than is intuitive. Ex: cos(a) - b returns cos(a - b), so you may want to write (cos(a)) - b.

`abs`
real

Result
real

absolute value of direct parameter

`acos`
real – -1 <= x <= 1

Result
real – in radians

arc cosine of direct parameter

`acosh`
real – a positive number

Result
real

hyperbolic arc cosine of direct parameter

`asin`
real – -1 <= x <= 1

Result
real – in radians

arc sine of direct parameter

`asinh`
real

Result
real

hyperbolic arc sine of direct parameter

`atan`
real

Result
real – in radians

arc tangent of direct parameter

`atan2`
list of real – 2 real numbers : y (ordinate) and x (abscissa)

Result
real – in radians

the angle of the line whose direction is the vector (x , y)

`atanh`
real – -1 < x < 1

Result
real

hyperbolic arc tangent of direct parameter

`cosh`
real

Result
real

hyperbolic cosine of direct parameter

`cos`
real – the angle (in radians). If the angle is in degrees, multiply it by pi / 180 before taking the cosine.

Result
real

cosine of direct parameter

`erf`
real

Result
real

the error function

`erfc`
real

Result
real

the complementary error function

`exp`
real

Result
real

exponential of direct parameter

`gamma`
real – a positive number

Result
real

the gamma function

`hypot`
list of real – 2 real numbers

Result
real

the square root of the sum of the squares of its arguments

`lgamma`
real – a positive number

Result
real

base-e logarithm of the absolute value of gamma

`ln`
real – a positive real

Result
real

base-e logarithm of direct parameter

`log10`
real – a positive real

Result
real

decimal logarithm of direct parameter

`sin`
real – the angle (in radians)

Result
real

sine of direct parameter

`sinh`
real

Result
real

hyperbolic sine of direct parameter

`sqr`
real

Result
real

square of direct parameter

`sqrt`
real – a positive number

Result
real

square root of direct parameter

`tan`
real – the angle (in radians)

Result
real

tangent of direct parameter

`tanh`
real

Result
real

hyperbolic tangent of direct parameter

`multlist`
list of real
`with` list of real

Result
a list of real

performs the product of the parameters. Each parameter may be a list, an array of real, or a number. multlist {x1,x2...} with {y1,y2...} returns {x1.y1, x2.y2, ...}; multlist x with {y1,y2...} returns {x.y1, x.y2, ...}

`divlist`
list of real
`with` list of real

Result
a list of real

same as multlist, but for quotient

`addlist`
list of real
`with` list of real

Result
a list of real

same as multlist, but for sums

`sublist`
list of real
`with` list of real

Result
a list of real

same as multlist, but for subtraction

`reversearray`
list of real – ... or an array of real

Result
array of real

returns reverse of the direct parameter.

`statlist`
list of real – ... or an array of real

Result
record

returns as a record the min, max, min index, max index, mean, standard deviation.

`replacemissingvalue`
list of small real
`with` small real

Result
a list of small real

replace missing values (or nans) in a list (or an array of real)

`read binary`
file specification – the file
`as` type class – the format of the data file: real (8 bytes) or small real (4 bytes)
[`skip`] integer – the number of bytes to skip
[`length`] integer – the number of real to read

Result
array of real

read a file of real or small real

`write binary`
file specification – the file
`with data` array of real
[`starting at`] integer – offset in bytes, default : append data at the end of the file

write the data into a binary file of small real (4 bytes per number)

`extractitem`
integer – the first item to read
[`thru`] integer – default -1
[`step`] integer – default 1

`in` array of real
[`blocksize`] integer – size of the block to read at each step. blocksize must be smaller than step

Result
array of real

`creatematrix`
string – "1": array of 1.0, "x": array of x values, "y": array of y values
`ncols` integer
`nrows` integer

Result
array of real

create an array of real of size ncols*nrows

`Class array of real` a packed list of small real. "array of real" is an opaque class, which can be coerced to and from a standard AppleScript list of real numbers. The Satimage osax and Smile use the "array of real" class for faster and safer computations on large lists of real numbers.
To make an array of real into a standard AppleScript list of real numbers, use "as list of real". Conversely, a list of real may be translated using "as array of real".

Appendix F

Built-in routines

When you run a script in Smile, the script is executed in Smile's context. This is true for scripts executed in a text window (as described in section 4.2). It remains true for any script that you run in Smile — e.g. a user script (see Chapter 9 about user scripts), or a script executing in a script window. In other words, within Smile, a `tell application "Smile"` is implicit.
The only exception is for windows connected to an application (see section 10 about connecting a window to an application), whose context restricts strictly to the context of the target application.

Smile's context contains a number of handlers, most of which are intended for internal use by Smile. Though, several of those handlers may be of more general use, and they are available to any script running in Smile. Here is that selection of handlers.

F.1 Handlers which display text

`FatalAlert (theString)` displays the string in an alert box, with the Stop icon and one OK button. Use `FatalAlert` to notify the user that some operation cannot be performed.
`FatalAlert` is one variant of `display dialog`.

`QuitAlert (theString)` same as `FatalAlert`, but the button reads Quit.
`QuitAlert` is one variant of `display dialog`.

`AskUser (thePrompt, theDefaultReply)` prompts the user to enter a string, and returns this string. The prompt is the string stored in `thePrompt`. The dialog box shows the Note icon, and displays the string stored in `theDefaultReply` as the default reply. If the user cancels, `AskUser` returns the error `User canceled` (error number -128). Use `AskUser` to have the user enter a string or cancel the current operation.
`AskUser` is one variant of `display dialog`.

`dd (theString)` displays the string in an alert box, with the Note icon and one OK button. Use `dd` to display an informative message, such as "Operation completed".
`dd` is one variant of `display dialog`.

`ShowMessage (theString)` displays the string in a small window named "Note". Use

ShowMessage to display an informative message while a script is running.

HideMessage () dismisses the message displayed with ShowMessage.

ShowHideMessage (theString, theDelay) same as ShowMessage, but dismisses the message after the delay stored in theDelay as seconds.

quietmsg (theString) appends the string to the Console window, in a new line. If the Console is not open, quietmsg will open it. If the Console is currently hidden by other windows, it remains hidden. If theString does not contain a string, quietmsg will attempt to apply the standard coercion into string: for instance you can pass a file descriptor as theString, but not a record.

msg (theString) same as quietmsg, except that msg brings to front the Console window.

log (theString) same as msg, except that if theString does not contain a string, log attempts to produce a string representation of theString. For instance, log can display a standard AppleScript record or a list. log will not display a value which belongs explicitly to an application other than Smile, such as startup disk of application "Finder".
To display the contents of a variable which contains such a value, use Smile's do script verb with as text. Since do script creates a temporary context for its own, use the my prefix to refer to the variable.

Example 104

```
quietmsg(do script "my x" as text)
```

F.2 Handlers which sort lists

sort (theList) returns a copy of theList sorted, by increasing values for numbers, resp. by alphabetic/ASCII order for strings. sort keeps theList unchanged. sort uses a recursive algorithm.

heapsort (theList) like sort except that heapsort uses a non recursive algorithm.

F.3 Miscellaneous helpers

make new name supplies a unique name, based on the current time and date, under the form YYMMDD_HHMMSS.

tid (theChar) a shortcut for setting AppleScript's text item delimiters to {theChar} tid("") restores the default value, which is the empty string.

Preset color constants Smile defines a set of colors. These are RGB colors described as lists of 3 integers between 0 and 65535:

Five grey levels black = {0, 0, 0}, charcoal, grey, mouse, white = {65535, 65535, 65535}

Seven colors red, green, blue, cyan, magenta, yellow and purple.

The Text suite uses that format for colors.

Example 105

```
set color of text of window 1 to
white
```

The SmileLab library and the pdf library ("Graphic Kernel") use colors described as lists of 3 or 4 numbers between 0 and 1. Thus, in SmileLab and in pdf's, you have to rescale their values in order to use the pre-defined constants.

Example 106

```
set pen color of theCurve to divlist
magenta with 65535
```

F.4 Handlers which open files

`OpenDictionary (thePath)` accepts a file reference or a list of such. Opens the dictionary(-ies) of the file(s).

`DoOpen (thePath)` the high-level handler for having Smile open a file. Performs more checking than the mere `open` event — that you can use as well. In particular, `DoOpen` will bring to front the window of the document if the file is already open in Smile.

`FileToWindow (thePath)` if the file is open in Smile, `FileToWindow` returns a reference to its window, otherwise it triggers `error number -1719 (Invalid index)`.

`GetText (thePath)` returns the source of the script if the file is a script document, otherwise `GetText` returns the contents of the file's data fork — in particular, its unformatted text for a text document.

`EditObjectScript (theObject)` opens the script of the object.

Example 107

```
EditObjectScript (last dialog item
of window 2)
```

Usually you can open the script of an object by *apple-option-click*. You may want to use `EditObjectScript` when for some reason you cannot use that combination — for instance the object may be invisible.

`EditClassScript (theObject)` displays the class script of the object. `EditClassScript` does not open the *Class script* file, it only displays a copy. To edit a *Class script*, open it normally.

F.5 Handlers which help manipulating Smile objects

`NewFileFromObject (theObject, thePath)` and

`NewObjectFromFile (thePath)`
`NewFileFromObject` saves the object into `thePath`, which should be a valid new file reference. `NewFileFromObject` stores the record obtained as `whole of theObject` in the resource fork of the new file `thePath`. `NewObjectFromFile` reloads that record and makes a new object from the data contained in the record — which includes the class of the object.
The `whole` property returns the "structural" information about the object, including its elements and its script, but not the data it may contain such as the contents of a text window or the current state or contents of an item of a dialog.

`PropagateBounds(theObject, theRect)` sets the width and height of the bounds

of `theObject` to those of the rectangle `theRect`

Appendix G

Reference of the PDF commands

G.1 Overview

Smile ships with a library of first-level primitives named *Graphic Kernel*, located in the *Class Scripts/Context Additions* folder. The functions of the Graphic Kernel form four groups:

- functions affecting the graphic state
- functions affecting the current path
- functions related to text
- basic geometric operations on points

The Graphic Kernel uses the following conventions to handle geometrical data:

a 2D point is either an AppleScript list of 2 real numbers or a record containing a `point` property (see section H).

a 2D vector assumes the same format as a 2D point

a color is an AppleScript list of 3 or 4 real rumbers belonging to [0.0, 1.0] (RGB or RGBA). A is the alpha channel, 0.0 is transparent and 1.0 is opaque

an angle is a real number in radian

G.2 Graphic state

G.2.1 Handling States

- `SaveState()`
- `RestoreState()`

`SaveState()` and `RestoreState()` are described in paragraph 23.3.2. They allow for saving and restoring the current stroke and fill settings, the current transformation, the current clip path (see G.3.1) and the current text settings (see G.4).

G.2.2 Stroke and Fill Settings

- `HSV2RGB` a command in Smile's dictionary to perform color translation
- `SetPenWidth(x)`
- `SetPenColor(rgba)`
- `SetFillColor(rgba)`
- `SetDashPattern(pat)` define the dash pattern
  `pat` is a list of real numbers `{phase, len1, len2, ...}` defining the dash pattern. `SetDashPattern({})` resets to no dash.

- `SetLineCap(lc)` sets the style for the endpoints of lines.
  `lc` is an integer in the range $[0, 2]$: 0 = LineCapButt (default), 1 = LineCapRound, 2 = LineCapSquare.

Figure G.1: LineCaps

- `SetLineJoin(lj)` sets the style for the juncture of connected lines.
  `lj` is an integer in the range $[0, 2]$. 0 = LineJoinMiter (default), 1 = LineJoinRound, 2 = LineJoinBevel.

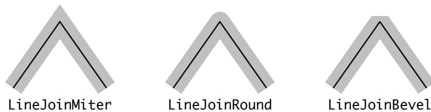

Figure G.2: LineJoins

- `SetMiterLimit(ml)` sets the miter limit for the juncture of connected lines.
  `ml` is a real number.

G.2.3 Applying transformations

- `SetTransformation(t)` apply `t` to subsequent commands

`t` is a list of 6 real numbers {`a, b, c, d, tx, ty`}. Any point will undergo the mapping:

$$\{x, y\} \longrightarrow \{ax + cy + tx, bx + dy + ty\}$$

Therefore the Identity transformation is {`1,0,0,1,0,0`}.
`SetTransformation(t)` combines `t` with the current transformation. To terminate applying the transformation, use `RestoreState`, that you must balance with a `SaveState` prior to `SetTransformation`.

G.3 Paths

G.3.1 Operations on paths

- `DrawPath(n)`: draws the path as described by the preceding graphic commands.
  `n` is an integer parameter in the range [0...4] corresponding to 0 for Fill, 1 for EvenOddFill, 2 for Stroke, 3 for FillStroke, 4 for EvenOddFillStroke. `DrawPath` makes the path effective using the current state settings.
  If you make any path, even a single `Lineto`, you must call `DrawPath` at least once as the last graphic command and before any change of the graphic state.

- `ClosePath()` completes the current path by adding a line from the latest point to the first point of the current path. Use `ClosePath` in order to tackle properly the ends of the strokes.

- `ClipToPath()` sets the clip region to the intersection of the current clip region with the current path. The clip region is the region to which drawing restricts. To restore the previous clip region (usually, the whole frame), use `RestoreState`, that you must balance with a `SaveState` prior to `ClipToPath`.

G.3.2 Building paths

- `Moveto(pt)` sets the current position to `pt`.

- `DMove(dv)` changes the current position by `dv`.
- `GetPathPosition()` returns the current position.
- `Lineto(pt)` appends a line from the current position to `pt`.
- `DLine(dv)` appends a line `dv` from the current position.
- `ArcToPointPath(p1, p2, theRadius)` appends an arc to the current path, possibly preceded by a straight line segment.
  `theradius` is the radius of the arc.
  The arc is tangent to the line from the current point to `p1`, and to the line from `p1` to `p2`. In other words, `ArcToPointPath` draws a broken line from the current point to `p1` then to `p2`, performing a rounded corner for `p1`.
- `ArcPath(theCenter, theradius, startAngle, endAngle, clockwise)` appends an arc of a circle to the current path, possibly preceded by a straight line segment.
  - `theCenter` is the center of the arc
  - `theRadius` is the radius of the arc
  - `startAngle` is the angle (in radian) to the first endpoint of the arc
  - `endAngle` is the angle to the second endpoint of the arc
  - `clockwise` is 1 if the arc is to be drawn clockwise, 0 counterclockwise.
- `CircleArcPath(theCenter, theRadius, startAngle, endAngle, clockwise)` appends an arc of a circle to the current path, line `ArcPath`, but without the straight line segment.
- `CirclePath(theCenter, theRadius)` appends a circle.
- `RectPath(r)` appends a rectangular path.
  `r` is a PDF rectangle, i.e. a list of 4 real numbers {left, bottom, width, height}.
- `QuadBezierPath(controlPoint, endPoint)` appends a quadratic Bezier path from the current point to `endPoint` with a control point `controlPoint` .
- `BezierPath(controlPoint1, controlPoint2, endPoint)` appends a cubic Bezier path from the current point to `endPoint` with 2 control points `controlPoint1` and `controlPoint2`.

G.4 Text

G.4.1 Text styles

- `SetTextMode(mode)` `mode` is an integer : 0 Fill, 1 Stroke, 2 FillStroke, 3 Invisible, 4 FillClip, 5 StrokeClip, 6 FillStrokeClip, 7 Clip.
- `SetTextFont(fontName)` `fontName` is a string.
- `SetTextSize(textSize)` `textSize` is a real number.
- `SetTextTransformation(t)` texts undergo the current path transformation and `SetTextTransformation` allows to combine the current transformation with a specific transformation for the text.

G.4.2 Drawing Text

- `TextMoveTo(pt)` sets the current text position to `pt`.
- `GetTexPosition()` returns the current text position. Useful to measure a string's length in conjunction with `SetTextMode(3)`.
- `DrawText(s)` draws the string `s` at the current text position with the current text settings, transformation and text transformation.
- `DrawString(s)` does not belong to *Graphic Kernel*: you have to install *GeomLib* (described in Appendix H) to use `DrawString`. `DrawString` is like `DrawText` except that it brings additional features intended to help positioning nicely the text.

G.5 2D geometry

- `vectfrompoint(A, B)` given the 2 points `A` and `B`, returns the vector $\overrightarrow{AB}$ as a list of 2 real numbers.
- `norm2(v)` returns the square of the norm of `v`.
- `det(v1, v2)` returns the determinant of the 2 vectors `v1` and `v2`.
- `scalpro(v1, v2)` returns the scalar product of the 2 vectors `v1` and `v2`.
- `normalize(v)` returns the normalized of `v`.

Appendix H

GeomLib, a graphical library for 2D geometry

GeomLib is an AppleScript library devoted to the creation of nice geometrical graphs like can be found in school books. For instance, GeomLib includes routines to display a point's name at a wanted location, or to mark angles.

If you want to take fully advantage of the GeomLib utilities on points, you may need to use "enriched" points: a point may be a record containing a property `point` (a list of 2 real numbers). The record may also include a `name` property (a string) and a `hint` property, which describes where the point's name is to be drawn (see H.1).

Thus {10,10} is a point and {name:"A", point:{10,10}} is the same point.
A circle is a point with an extra `radius` property:

```
{name:"O", point:{100,100}}, radius:50, hint:"tr"}
```

is a circle.

H.1 Text Utilities

- `DrawString` is similar to `DrawText(s)`. But, the string `s` in not drawn as such. First, `s` may begin with a bracketed positional information: `[pos]` where `pos` is 1 or 2 character(s). The characters must be `t` (top), `b` (bottom), `l` (left), `r` (right) or `h` (here).

  Example 108
  `DrawString("[br]A")`

  draws the character "A" below and on the right of the current text position.

  Example 109
  `DrawString("[h]A")`

  draws the character "A" centered around the current text position.

  Furthermore, the characters _ ^ \ are interpreted anywhere in the string: _ holds for subscript, ^ for superscript, and \ for Symbol font. Use brackets ({}) to group

characters.

Example 110

```
DrawString("[t]\\a=x_{ij}^2+1")
```

outputs:
$\alpha = x_{ij}^2 + 1$
above the current text position and centered horizontally.

- `SetTextOffset(dx, dy)` sets the offset from the current text position and the location where the text is to be drawn when using the positional parameters.

- `DrawName(pt)` draws the `name` property of `pt` at position `point` of `pt` with an offset defined by `hint` of `pt`; i.e. if `hint` is present `DrawName(pt)` calls:

  Example 111

  ```
  DrawString("[" & hint of pt & "]" &
  name of pt)
  ```

  If `hint` does not exist in `pt`, `DrawName(pt)` draws the name of the point in the area opposite to the "center" of the drawing.
  By default, the center of the drawing is `{0,0}`. You can change the center of the drawing with the function `SetCenter(pt)`.

H.2 Marking

H.2.1 Marking Angles

- `MarkAngle(B, A, C, r, dr, narc, ndash)` marks the $\widehat{BAC}$ angle with `narc` small arcs of radius $r + k.dr$ and `ndash` dashes (`ndash` should be 0 or 1).
  `MarkRightAngle(B, A, C, len)` marks the angle $\widehat{BAC}$ with a small square of side `len`.

Example 112

```
set a to {name:"A", point:{20, 20}}
set b to {name:"B", point:{20, 100}}
set c to {name:"C", point:{100, 20}}
BeginFigure(0)
DrawPolygonAndName({a, b, c})

DrawPath(2) SetPenWidth(0.5)
MarkRightAngle(b, a, c, 10)
MarkAngle(a, b, c, 15, 4, 2, 0)
MarkAngle(b, c, a, 15, 4, 2, 0)
DrawPath(2) EndFigure()
```

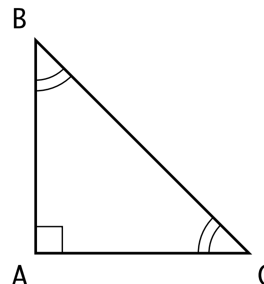

Figure H.1: Output of Example 112

H.2.2 Marking Points

- `MarkPointOnLine(pt, seg, dl)` marks the point `pt` on the segment `seg` (a list of 2 points) by adding a small dash of length `dl` orthogonal to `seg` at `pt`.

- `CrossPath(pt)` appends a cross at `pt`.

- `VertCrossPath(pt)` appends a vertical cross at `pt`.

- `SetCrossSize(dl)` defines the size of the crosses.

H.2.3 Arrows

- `ArrowPath(angle)` appends an arrowhead pointing towards the direction `angle`.
- `SetArrowSize(list)` defines the size of the arrowheads. `list` is a list of 3 real numbers (see figure H.2.3).

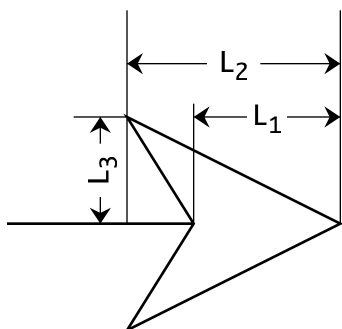

Figure H.2: Result of `SetArrowSize`$(\{L_1, L_2, L_3\})$

H.3 Basic Geometry

- `CenterOfMass(listp)` returns the center of mass of the list of points `listp`.
- `Bisector(A, B, C)` returns the normalized bisector of $\widehat{BAC}$.
- `Symmetric(A, P)` returns the symmetric of `A` with respect to `P`. `P` may be a point or a segment (a list of 2 points).
- `Project(A, seg)` returns the projected of `A` onto the segment `seg`.
- `CircleFromPoints(lp)` returns a record {point: center of the circle, radius: the radius}. `lp` is a list of 3 points.
- `Intersect(d1, d2)` returns the intersection of the 2 segments `d1` and `d2`.

H.4 Basic Geometric Figures

- `DrawCircle(c)` appends a circle path, and marks the center with a cross. If the center has a name, the name is displayed.
  Example : `DrawCircle({name:"O", point:{100,100}, radius:50, hint:"rt"})`.
- `DrawPolygonAndName(l)` appends a polygon path and writes the names of the points away from the center of mass. This is particularly convenient for convex polygons, in other cases you may want to add a hint to some points.
- `DrawLine(p1, p2, leftPercent, rightPercent)` appends a line which contains `p1` and `p2`. The line is extended by `leftPercent`% on the `p1` side and `rightPercent`% on the `p2` side.
- `DrawEllipsis(P, axish, axisv)` appends an ellipsis whose center is the point `P` and whose axes are the vectors `axish` and `axisv`.
- `DrawVector(A, B)` appends the segment `AB` plus an arrow at `B`.

Example 113

```
BeginFigure(0)
set l to {}
```

```
set n to 5
set dphi to 2 * pi / n
set {x, y} to {100, 100}
set r to 50
repeat with i from 0 to (n - 1)
    set phi to i * dphi
    set p to {x + r * (cos phi), y + r *
(sin phi)}
    set end of l to {point:p, name:"A_{"
& (i + 1) & "}"}
end repeat
set item 1 of l to item 1 of l &
{hint:"rb"}
DrawPolygonAndName(l)
DrawPath(2)
SetPenWidth(0.5)
DrawLine({x + r, y}, {x - r, y}, 20, 20)
set c to CircleFromPoints(l) &
{name:"O", hint:"t"}
DrawCircle(c)
set l to {item n of l} & l & {item 1 of
l}
repeat with i from 2 to (n + 1)
    MarkAngle(item (i + 1) of l, item i
of l, item (i - 1) of l, 10, 0, 1, 0)
end repeat
DrawPath(2)
EndFigure()
```

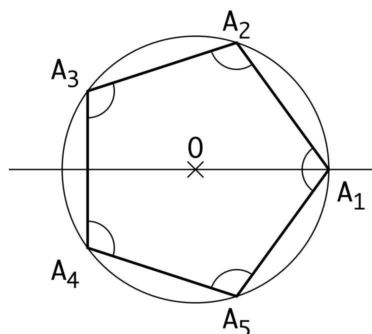

Figure H.3: Output of example 113