

Scilab Bag Of Tricks

The Scilab-2.5.x IAQ (Infrequently Asked Questions)

Lydia E. van Dijk
Hammersmith Consulting

Christoph L. Spiel
Hammersmith Consulting

Scilab Bag Of Tricks: The Scilab-2.5.x IAQ (Infrequently Asked Questions)

by Lydia E. van Dijk and Christoph L. Spiel

Copyright © 2000, 2001 by L. E. van Dijk, Ch. L. Spiel

sci-BOT – the Scilab Bag Of Tricks – is a collection of Scilab experience that come from every day use. We warn of common pitfalls, discuss stylistic issues, shed light on unknown spots, and show many different ways of increasing the performance of Scilab functions.

The document is not meant to be comprehensive or even suitable to a particular level of knowledge. Some sections are at the beginners level, some even surprise long-time users.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no invariant sections, with the Front-Cover Texts being “Scilab Bag Of Tricks”, and with no Back-Cover Texts. A copy of the license is included in the appendix “GNU Free Documentation License”.

Trademarks:

DIGITAL™ is a trademark of Digital Equipment Corp.

IBM™ is a trademark of International Business Machines Corp.

Intel™ is a trademark of Intel Corp.

Matlab™ is a trademark of The Mathworks, Inc.

Microsoft™ and Microsoft Windows™ are trademarks of Microsoft Corp.

PostScript™ is a trademark of Adobe, Inc.

SGI™ is a trademark of Silicon Graphics International, Inc.

SUN™ is a trademark of Sun Microsystems, Inc.

TeX™ is a trademark of the American Mathematical Society

Copyrights:

GNU/Linux© copyrighted by Linus Torvalds

GNUPlot© copyrighted by Thomas Williams, Colin Kelley and many others

MuPAD© copyrighted by Benno Fuchsensteiner

Octave© copyrighted by John W. Eaton

Pari© copyrighted by C. Batut, K. Belabas, D. Bernardi, H. Cohen, and M. Olivier

PlotMTV© copyrighted by Kenny Toh

Scilab© copyrighted by INRIA, France

Tela© copyrighted by Pekka Janhunnen

Revision History

Revision 0.15 2001-7-4 Revised by: lvd

New sections and new chapter

Revision 0.14 2001-1-11 Revised by: lvd

Major additions, lots of fixes

Revision 0.13 2000-8-2 Revised by: lvd

Minor update

Revision 0.12 2000-6-4 Revised by: lvd

First public release

Revision 0.11 2000-5-1 Revised by: lvd

First semi-public release

Revision 0.10 2000-3-17 Revised by: cls

Translated from 0.1 pod-version

Table of Contents

Preface	15
1. Outline.....	15
2. Other Formats of sci-BOT.....	16
3. Packed examples	17
4. Typographic conventions	17
5. Scilab Release History	18
6. Contributions.....	19
7. Acknowledgments.....	20
1. Introduction.....	21
2. Common Pitfalls.....	23
2.1. The Infamous Dot	23
2.2. Vector Construction.....	24
2.3. Function head	25
2.4. Last Newline	26
2.5. Variable Lifetime And Scoping.....	27
2.5.1. Local Variable Scoping.....	27
2.5.2. Global Variables	30
2.5.3. Clearing Variables	31
2.6. Dangerous Range Generation	32
3. Programming Style	35
3.1. Spacing and Formatting	35
3.1.1. Intra-Expression Spacing.....	35
3.1.2. Line Breaking	35
3.1.3. Setting Brackets Apart.....	36
3.1.4. Vertical Spacing.....	36
3.2. Indentation	39
3.3. Single Quotes vs. Double Quotes	41
3.4. Choice Of Control Structures.....	41
3.4.1. while/for.....	41
3.4.2. if/select.....	42
3.4.3. Strict Block Structure/Early Return.....	43
3.5. Size of a Function	44
4. Unknown Spots	47
4.1. Keywords and Commands	47
4.2. Operator Overloading ¹	48
4.2.1. Overloading crash course	49
4.2.2. Overload syntax.....	51

1. Thanks go to Bruno Pinçon for carefully proofreading this section.

4.2.3. Overloading example.....	56
4.3. Operator Precedence And Associativity	58
4.3.1. Numeric Operators	58
4.3.2. Relational Operators.....	59
4.3.3. Logical Operators	59
4.4. Boolean Peculiarities.....	60
4.4.1. Implicit Cast To Boolean.....	60
4.4.2. Boolean Vector- or Matrix-Indices.....	61
4.5. Integers.....	62
4.5.1. Missed Opportunities.....	62
4.5.1.1. No Integer Literals	62
4.5.1.2. No Implicit Conversion in Mixed Typed Expressions	62
4.5.1.3. No Integer Array Indices	63
4.5.1.4. Limited Support of Integers in Functions	64
4.5.1.5. Partial Support of Integer Values in Data Files.....	64
4.5.2. Digest of Integer Go And No-Gos.....	64
4.5.2.1. Which Functions Support Integers?.....	65
4.5.2.2. Modular Integers.....	66
4.5.2.3. Troublesome Spots.....	67
4.5.2.3.1. Array Concatenation	67
4.5.2.3.2. Mixed Type Expressions	67
4.5.2.3.3. Mixed Type Comparisons.....	69
4.5.2.3.4. Vector-Scalar Comparison of Identical Type Integers.....	71
4.5.2.3.5. System Dependence of Type int8.....	72
4.5.2.4. Integers in Bitwise Operations.....	72
4.6. Miscellaneous Unknown Spots	73
4.6.1. Maximum variable name length	74
4.6.2. Starting scilex	74
4.6.3. Tuple Assignment.....	75
4.6.4. Dot as Member Selector	77
5. User Functions	79
5.1. Functions	79
5.1.1. On(e)line Function Definitions.....	79
5.1.2. Nested Function Definitions	80
5.1.3. Functions Without Parameters or Return Value	81
5.1.4. Named Parameters.....	82
5.1.5. Bulletproof Functions.....	83
5.1.6. Function Variables	85
5.1.7. Functions as Parameters in Function Calls.....	86
5.1.8. Omitting Parentheses on Function Call.....	87
5.1.9. Functions in tlists and mlists.....	88
5.1.10. macrovar	89
5.2. Libraries of sci-functions	89
5.2.1. getf vs. lib	90

5.2.2. genlib	94
5.2.3. Library Caveats.....	94
5.2.3.1. Library Files and Library Functions	95
5.2.3.2. On-Demand Loading	95
5.2.4. Loading Non-Functions With lib	96
6. Performance	99
6.1. High-Level Operations	99
6.1.1. Vectorized Operations	99
6.1.2. Avoiding Indexing and Resizing	101
6.1.2.1. \$-Constant.....	104
6.1.2.2. Reshaping.....	104
6.1.2.3. Flattened Matrix Representation.....	105
6.1.3. Built-In Vector-/Matrix-Functions.....	106
6.1.3.1. Vector Generation	106
6.1.3.1.1. Operator “:”	106
6.1.3.1.2. linspace.....	107
6.1.3.1.3. logspace.....	107
6.1.3.2. Whole Matrix Construction	107
6.1.3.2.1. zeros	107
6.1.3.2.2. ones	108
6.1.3.2.3. eye	108
6.1.3.2.4. diag	109
6.1.3.2.5. rand	110
6.1.3.2.6. emptystr.....	110
6.1.3.3. Functions Operating on a Matrix as a Whole	111
6.1.3.3.1. find	111
6.1.3.3.2. max, min.....	113
6.1.3.3.3. and, or	114
6.1.3.3.4. Operator “&”, Operator “ ”	114
6.1.3.3.5. sum, cumsum, prod, cumprod.....	114
6.1.3.3.6. gsort	115
6.1.3.3.7. size	117
6.1.3.3.8. matrix.....	118
6.1.4. Evaluation of Polynomials.....	119
6.2. Extending Scilab	120
6.2.1. Comparison Of The Link Overhead	121
6.2.2. Preparing And Compiling External Subroutines	125
6.2.2.1. Fortran-77	125
6.2.2.2. Fortran-9x	126
6.2.2.3. (ANSI-) C.....	127
6.2.2.4. C++	127
6.2.2.5. Ada.....	130
6.2.2.6. Visual C++	131
6.2.2.7. Borland C 5.01	133

6.2.3. Pushing It Further	135
6.2.3.1. Scilab as Prototyping Environment.....	135
6.2.3.2. Scilab to Fortran-77 Compiler	135
6.3. Building an Optimized Scilab	135
7. Scilab Core	139
7.1. Introduction To Pseudo-Ada	139
7.2. Internal Data Structure	141
7.2.1. Parameter Stack And Data Stack.....	141
7.2.2. Storage of Complex Matrices	141
7.3. Writing Native Scilab Functions	146
7.3.1. Simple Functions	146
7.3.2. Functionals	153
7.3.3. Library Interfaces	158
7.3.3.1. Dispatch Tables	158
7.3.3.2. Interface Generator	160
7.4. Error Handling	163
7.4.1. Fatal Errors	163
7.4.2. Warnings	164
7.4.3. Messages.....	164
7.5. Fortran Interface to Scilab's Core	165
7.5.1. Query	165
7.5.1.1. checkrhs	165
7.5.1.2. checklhs	166
7.5.1.3. lhs	167
7.5.1.4. rhs	167
7.5.2. Access Object	167
7.5.2.1. getmat.....	168
7.5.2.2. getrmat	169
7.5.2.3. getrvect	169
7.5.2.4. getvect	170
7.5.2.5. getscalar	170
7.5.2.6. getexternal.....	171
7.5.3. Create Object.....	173
7.5.3.1. Cremat.....	174
7.5.4. Miscellaneous	174
7.6. C Interface to Scilab's Core	174
7.6.1. Query	174
7.6.1.1. CheckRhs	174
7.6.1.2. CheckLhs	175
7.6.1.3. Lhs	176
7.6.1.4. Rhs	177
7.6.1.5. GetType	177
7.6.2. Access Object	178
7.6.2.1. GetRhsVar	179

7.6.2.2. GetMatrixptr.....	180
7.6.2.3. GetMatrixDims	180
7.6.2.4. GetRhsCVar	181
7.6.2.5. GetListRhsVar	181
7.6.2.6. GetListRhsCVar	182
7.6.2.7. GetFuncPtr	182
7.6.2.8. LhsVar.....	183
7.6.3. Create Object.....	184
7.6.3.1. CreateVar	184
7.6.3.2. CreateVarFromPtr	185
7.6.3.3. FreePtr	186
7.6.3.4. CreateCVar	186
7.6.3.5. CreateCVarFromPtr.....	186
7.6.3.6. CreateData	187
7.6.3.7. CreateListCVarFromPtr	187
7.6.3.8. CreateListVarFromPtr	188
7.6.3.9. Createlist	188
7.6.3.10. WriteMatrix.....	188
7.6.3.11. WriteString.....	189
7.6.4. Miscellaneous	190
7.6.4.1. Scierror	190
7.6.4.2. PExecSciFunction.....	190
7.6.4.3. ReadString.....	191
7.6.4.4. SciFunction.....	191
7.6.4.5. SciString	192
8. Further Information.....	193
8.1. Coping With Scilab	193
8.1.1. Distribution Size	193
8.1.1.1. CVS.....	193
8.1.1.2. locate	193
8.1.1.3. Glimpse	194
8.1.2. Bug Hunting	194
8.2. Local Documents	195
8.3. Hyperlinks	197
9. Notes for Contributors.....	199
9.1. Writing Style	199
9.2. Technicalities.....	200
9.2.1. DocBook.....	200
9.2.1.1. Guidelines	201
9.2.1.2. Indentation style.....	201
9.2.2. Tables.....	202
9.2.3. Examples	204
9.2.4. Graphics.....	205

9.2.5. Mathematics	206
9.2.6. Index terms	207
10. Complete Examples	209
10.1. frac.sci	209
10.2. benchmark.sci.....	220
10.3. listdiff.sci.....	222
10.4. whatis.sci	225
10.5. Auto-Determination of Precedence and Associativity	229
10.5.1. assoc.sci.....	229
10.5.2. prec.sci.....	230
10.5.3. parser.sci.....	232
10.6. cat.sci.....	233
10.7. quadpack.sci.....	235
A. GNU Free Documentation License.....	237
0. PREAMBLE	237
1. APPLICABILITY AND DEFINITIONS	237
2. VERBATIM COPYING.....	238
3. COPYING IN QUANTITY	238
4. MODIFICATIONS.....	239
5. COMBINING DOCUMENTS.....	241
6. COLLECTIONS OF DOCUMENTS	241
7. AGGREGATION WITH INDEPENDENT WORKS.....	241
8. TRANSLATION	242
9. TERMINATION.....	242
10. FUTURE REVISIONS OF THIS LICENSE.....	242
B. GNU General Public License	243
0. APPLICABILITY ¹	244
1. VERBATIM COPYING.....	244
2. MODIFICATIONS.....	244
3. DISTRIBUTION.....	245
4. TERMINATION.....	246
5. ACCEPTANCE.....	246
6. REDISTRIBUTION.....	246
7. CONSEQUENCES	247
8. LIMITATIONS.....	247
9. FUTURE REVISIONS OF THIS LICENSE.....	247
10. AGGREGATION WITH INDEPENDENT WORKS.....	248
11. NO WARRANTY	248
12. LIABILITY	248

1. The titles of the sections have been added by the authors. They do not occur in the original GNU General Public License. Everything else has been copied in verbatim.

Bibliography	251
Index.....	253

List of Tables

4-1. Reserved Words.....	47
4-2. Commands.....	47
4-3. List of all operand type codes.....	52
4-4. List heads used by Scilab	53
4-5. Operator type codes.....	54
4-6. Arithmetic Operators.....	58
4-7. Boolean Operators.....	60
4-8. Selected Functions and Operators That <i>Work With Integers</i>	65
4-9. Selected Functions and Operators That <i>Do Not Work With Integers</i>	65
6-1. Comparison of various vectorization levels	100
6-2. Mode Specifiers for <code>gsort</code>	115
6-3. Direction Specifiers for <code>gsort</code>	115
6-4. Performance comparison of different polynomial evaluation routines	120
7-1. pAda to Fortran-77 and C type mappings – elementary types	139
7-2. pAda type mappings – Scilab Fortran-77 interface	140
7-3. pAda type mappings – Scilab C interface	140
7-4. TString Identifiers	178
9-1. DocBook approximation of a DIN conforming table.....	203

List of Figures

6-1. Benchmark results for the <code>mirror</code> functions.....	124
9-1. Table formatted according to DIN 55301.....	204

List of Examples

2-1. Building a matrix column-by-column and row-by-row	24
2-2. Canonicalization of Scilab files.....	26
2-3. Shadowing of local variables.....	27
2-4. Accessing variables from the enclosing scope	27
2-5. Dynamic Scoping	28
2-6. Equivalent Representation of a Colon-Expression.....	32
3-1. Function <code>whocat</code>	39
3-2. Function <code>mysign</code>	43
4-1. Manually launching scilex	74
5-1. Function <code>tauc</code>	80
5-2. Function accepting named arguments	82
5-3. Function accepting optional arguments.....	83
5-4. Function <code>cat</code>	84
5-5. Generate names for <code>lib: gen-names</code>	93

6-1. Variants of a matrix mirror function.....	102
6-2. Function deblank.....	110
6-3. Naive functions to evaluate a polynomial	119
6-4. Less naive functions to evaluate a polynomial	120
6-5. Sample interface description (“.desc”)	122
6-6. Makefile for static Scilab interfaces via intersci	123
7-1. Multiplication of complex matrices	144
7-2. Simple native Scilab function.....	148
7-3. Scilab functional.....	154
7-4. Handling of warnings in Scilab	164
9-1. SGML-code for a DIN compliant table.....	202
9-2. Inclusion of graphics	205
9-3. Inclusion of mathematics.....	206

Preface

Often we encounter technical problems that we have to solve, to overcome somehow, or just to work around. After having mastered the difficulty, we gladly add it to the knowledge-base in our mind, but from a certain level of difficulty we start to make notes in one form or the other. These notes then serve for later reference. A collection of related notes can be exploited to gain further insight in the class of problems it describes. Last but not least one can get ambitious to fill the holes of knowledge that an existing set of notes leaves unanswered.

Richard B. Johnson

An expert in a particular computer language is really an expert in the work-arounds necessary to use this language to perform useful work. An ideal computer language would do exactly what it was told simply from reading a specification. In the absence of a specification, it would ask enough questions to produce such a specification, then it would generate the code necessary to perform the specified functions.

...

Even C has its shortcomings which have to be handled with assembly language extensions. A Master Carpenter has many tools and is expert with most of them. If you only know how to use a hammer, every problem begins to look like a nail. Stay away from that trap. It bytes (sic).

This is the story of sci-BOT paraphrased. It started with bits of experience gathered in our heads and scattered e-mail correspondence. After more and more e-mails piled up, telling the same old stories, one of the authors (lvd) decided to compile the problems and their solutions into a convenient format. Perl's plain old documentation, POD, was chosen for its simplicity paired with a multitude of output formats. However, after 2000+ lines it became clear that POD was missing a feature that would be needed as sci-BOT grows bigger: cross references. A more powerful documentation format and the associated tools had to be found. A two week web research resulted in one clear winner: DocBook. The downside of the necessary switch of formats was that the previous work done with POD had to be converted into DocBook. Daytime work plus adding new material to sci-BOT plus converting the old work into the new format is too much for a single volunteer. So, a second idiotM-DELauthor was searched and found (cls). His ten years of experience with the TeX/LaTeX typesetting system, his accuracy, and his intensity with which he attacks any obstacle made him the ideal choice for this madnessM-DELproject.

1. Outline

We open up talking about some of the most common syntactic pitfalls when using Scilab in Chapter 2. Finding that some of these syntax problems can be avoided with a clear programming style, the next chapter, Chapter 3, deals with coding issues. In Chapter 4 we then focus on the parts of Scilab that are not well documented, and therefore widely remain unknown spots. For many users not only enjoy the nice user interface of Scilab, but demand high performance from the interpreter the massive Chapter 6 about performance issues covers these needs. It begins by introducing techniques suitable at a high level like vectorization which do not require low level programming and then dives

down into the extension of Scilab by compiled routines. This is a vast field by itself. Therefore we have devoted a full chapter, Chapter 7, to the low level API. sci-BOT closes with Chapter 8 containing remarks on compiling and debugging as well as comments on the supplied documentation and available web pages. All of the programming snippets that belong to longer examples which do not fit in the main text have been gathered in Chapter 10, where they show up in full length.

At the end of the document we have put two appendices with the GNU Free Documentation License, and the GNU Public License, a bibliography, and an index.

2. Other Formats of sci-BOT

sci-BOT, the Scilab Bag-of-Tricks is available as SGML, as HTML, or several “printer-ready” versions. Check out *Hammersmith Consulting* for the latest release. Each variant is available in different packing-/compression formats.

SGML source distribution. The Real Thing (tm)! These are our SGML-sources. Building sci-BOT from source requires XML DocBook version 4.x.

data	checksum
<i>sci-bot-sgml.tar.gz</i>	<i>sci-bot-sgml.tar.gz.md5</i>
<i>sci-bot-sgml.tar.bz2</i>	<i>sci-bot-sgml.tar.bz2.md5</i>
<i>sci-bot-sgml.tar.Z</i>	<i>sci-bot-sgml.tar.Z.md5</i>
<i>sci-bot-sgml.zip</i>	<i>sci-bot-sgml.zip.md5</i>

Web collection. This is sci-BOT rendered in HTML; conveniently bundled for your offline reading pleasure.

data	checksum
<i>sci-bot-html.tar.gz</i>	<i>sci-bot-html.tar.gz.md5</i>
<i>sci-bot-html.tar.bz2</i>	<i>sci-bot-html.tar.bz2.md5</i>
<i>sci-bot-html.tar.Z</i>	<i>sci-bot-html.tar.Z.md5</i>
<i>sci-bot-html.zip</i>	<i>sci-bot-html.zip.md5</i>

Print versions. The printable versions are formatted for DIN A4 paper and are single files. By the way, you do not have to print them; they look great with Ghostview, too.

data	checksum
<i>sci-bot.ps.gz</i>	<i>sci-bot.ps.gz.md5</i>

data

sci-bot.ps.bz2
sci-bot.ps.Z
sci-bot.ps.zip
sci-bot.pdf.gz
sci-bot.pdf.bz2
sci-bot.pdf.Z
sci-bot.pdf.zip

checksum

sci-bot.ps.bz2.md5
sci-bot.ps.Z.md5
sci-bot.ps.zip.md5
sci-bot.pdf.gz.md5
sci-bot.pdf.bz2.md5
sci-bot.pdf.Z.md5
sci-bot.pdf.zip.md5

3. Packed examples

Some of the examples in the main text and all examples in the Appendix can be obtained in a single tar or zip-file.

data

scibot-examples.tar.gz
scibot-examples.tar.bz2
scibot-examples.tar.Z
scibot-examples.zip

checksum

scibot-examples.tar.gz.md5
scibot-examples.tar.bz2.md5
scibot-examples.tar.Z.md5
scibot-examples.zip.md5

4. Typographic conventions

This section covers the conventions used in this book. Depending on what version you are currently reading some fonts may look the same.

Typographic Conventions

`filename`

This font designates the name of a file. A filename optionally includes a path.

`user input`

This font is used for the user's input. This refers only to things that can be typed in at the console.

meta-variable

This typeface is reserved for placeholders, i.e. stuff that always is replaced with the real input.

`literal piece of code`

We use this font to display literal pieces of code, variables, constant as well as operators.

`variable`

Variables of all kinds are marked up this way.

`function`

Functions or procedures of all kinds are marked up this way.

command

We use this font for shell commands, but also for Scilab commands.

`environment-variable`

To distinguish environment variables from program variables a separate font is used.

Transcripts from actual interactions with an interpreter, which can be Scilab, bash(1), or any other interactive program are displayed like this.

```
->x = 1e22
x  =

      1.000E+22

->sin(x), cos(x)
ans =

      0.4626130
ans  =

- 0.8865603
```

In examples, which show some source-code, additional comments always start with two dashes independent of the language. JadeTeX coerces these two dashes into one longer dash, called en-dash.

```
#include <stdio.h>

/* The world's most famous C-program */

int
main(void)
{
    printf("Hello world!\n");
    return 0;
}
- exit code for success
```

5. Scilab Release History

by Enrico Segre

Some details of the Scilab distribution depend on the version. Whenever the distinction between different versions is necessary, the following identifiers will be used.

Scilab-2.4.1 (official release)

stable version

Scilab-2.5 (official release)

stable version, released December 1999

Scilab-2.5.1 (alpha version)

official unstable release as of 2000-7-21

Scilab-2.5.1 (first beta version)

unofficial release from the Saphir site; intended for an INRIA course; first spotted 2001-1-22.

Scilab-2.5.1 (second beta version)

unofficial branch from the INRIA sources as of 2001-1-10 modifications to up to and including 2001-1-18, courtesy of Stéphane Mottelet.

Scilab-2.6 (alpha version)

official unstable release

Scilab-2.6 (official release)

official version, released 2001-3-26

6. Contributions

Contributions, corrections, hints, and tips always are welcome. If you are willing to contribute a whole section or even chapter, please take a look at Chapter 9, or contact the authors.

This version of sci-BOT contains contributions from

Name	E-mail	Section[s]
Glen Fulford	FulfordG@agresearch.cri.nz	Section 5.1.8
Enrico Segre	fesegre@wisemail.weizmann.ac.il	Section 5, Section
Dave Sidlauskas	dsidlauskas@worldnet.att.net	Section 6.2.2.6

7. Acknowledgments

Lydia van Dijk: To the CCMR system administrators Daniel Blakely and Berry Robinson for providing a rock solid multi-platform environment at the early stages (prior to version 0.14) of this project.

Christoph “Solo” Spiel: First of all thanks go to Lydia. Before working with her I did not know whether I am insane. This project has removed all doubt. “Wonderful girl! Either I’m going to kill her, or I’m going to like her.” – Han Solo about Princess Leia in “A New Hope”.

To F. Maximilian “Tiger” Pitschi. You have shown me the difference between software engineering and hacking. Kick me again...

Chapter 1. Introduction

“I have read your posting as of ... to the Scilab newsgroup. It was very clear. Can you make a FAQ out of it?” Yes, we can, and here it is!

The hints, tricks, and information put together in sci-BOT come from our own experience (read: daily struggle), problems we have solved for our colleagues, and of course questions answered on the newsgroup. Therefore, this document is a rather loose collection of facts, and is not necessarily to be read cover to cover.

What this document is not:

- An introduction to Scilab

There already is an excellent “Introduction to Scilab”, the Scilab User’s Guide, `SCI/doc/Intro.ps`.

- A replacement for reading the documentation

IONSHO (“In Our Not So Humble Opinion”) folks who do not read the documentation get what they deserve. Scilab’s documentation is truly great, so why not using it? To get a command’s manual-page type **help** at the command prompt. The same is achieved in the graphical environment with the **Help** button. If the exact command name is unknown, the powerful cousin of **help**, **apropos** jumps in. It can be used from the command line as well as from the Help Panel.

- Another *FAQ* list

We do not follow the simple Question-and-Answer style, instead we try to explore Scilab right to its very end.

In the spirit of the Free Software any helpful suggestion or correction concerning this collection will be acknowledged with the author’s name and email address. If you want to tell us of a mistake, or want an item added, please drop the authors an email at

`<lvandijk@hammersmith-consulting.com>` or
`<cspiel@hammersmith-consulting.com>`.

Chapter 2. Common Pitfalls

The nice thing about Scilab? It is almost usable!

Enrico Segre

There are several peculiarities in Scilab's way of interpreting an expression that will trip the unwary. Some of them are a result of "compatibility" to a certain commercial product of similar sounding name (which one?), others are home grown quirks.

2.1. The Infamous Dot

In Scilab a digit in front or after the decimal point is *not* enforced. This is similar to e.g. Fortran and C, but contrary to Ada. Thus, for Scilab the following three numbers are well formed

```
87.492211
.32493
6857.
```

As an aside:

```
digit+.0
digit+.
digit+
```

e.g. 123.0, 123., and 123 are considered identical. The last of the three examples, a decimal point at the end of the numeral, baffles users who want to invert a vector or matrix component-wise.

```
->1 ./ [1 2 3]
ans =
! 1. 0.5 0.3333333 !
```

But, hey this is correct! Then, let us squeeze out the spaces in front of the ./ operator.

```
->1./ [1 2 3]
ans =
! 0.0714286 !
! 0.1428571 !
! 0.2142857 !
```

Oops! What happened? The last expression is not interpreted as

```
(1) ./ ([1 2 3])
```

but as

```
(1.) / ([1 2 3])
```

where the parentheses have been introduced for clarity. This behavior is described in SCI/README, and in the *Scilab FAQ*.

We suggest to avoid whitespace that influences the calculation by not letting the decimal point stick out on either side. That way expressions with numerals will always be interpreted correctly. For our example this means

```
->1.0./ [1 2 3]
ans =
! 1.      0.5      0.3333333 !
```

which gives what we had in mind.

2.2. Vector Construction

The square bracket operator `[]` is a convenient means to construct vectors. There even exists an idiom to build a matrix with brackets, which is shown in Example 2-1.

Example 2-1. Building a matrix column-by-column and row-by-row

```
mat = []
for i = 1:n
    row = []
    for j = 1:m
        ...                               – compute matrix entry
        expr = ...
        row = [row expr]
    end
    mat = [mat; row]
end
```

Rows are separated by semi-colons or newlines, which actually is straight forward. Columns are separated by commas, or spaces—and here comes trouble. First, comma and space serve the same purpose, and are interchangeable. Thus, the following expressions have the same result.

```
[1 2 3 4]
[1,2,3,4]
[1 2 3,4]
[ 1, 2   3   , 4 ]
```

Second, a space is sometimes considered a column-separating space, sometimes a intra-expression space. This can lead to some confusion as the following three matrix definitions demonstrate. Who gets all of them right without peeking at the answers?

```

->m1 = [1+%i -1+%i; -1+%i 1-%i]
m1 =
!   1. + i   - 1. + i   !
! - 1. + i   1. - i   !

->m2 = [1 +%i - 1 + %i; - 1 + %i 1 - %i]
m2 =
!   1.   - 1. + 2.i   !
! - 1. + i   1. - i   !

->m3 = [1 +%i -1 + %i; - 1 + %i 1 -%i]
m3 =
!   1.   i   - 1. + i   !
! - 1. + i   1.   - i   !

```

Confusion makes the programmer susceptible to writing code she did not intend. To make the matrix expression clear to you and to Scilab there are at least two possibilities.

1. Using no spaces in the construction of the elements of a matrix. This is e.g. demonstrated in `m1` above, or
2. Putting every compound expression in parentheses, like

```

->[(1 +%i) (-1 + %i); (- 1 + %i) (1 -%i)]
ans =
!   1. + i   - 1. + i   !
! - 1. + i   1. - i   !

```

Both ways avoid the ambiguity.

Actually, matrices as simple as the ones shown in the examples can be arranged in a neat way. It is discussed in Section 3.1.2. See also Section 3.1.1 on how to improve the legibility of Scilab code by the judicious use of whitespace.

2.3. Function head

Scilab treats the first (logical) line of a function definition, the function head, differently from any other line in a sci-file. Any non-whitespace after the closing parenthesis *must* be avoided. It is even illegal to add a comment at the end of the function head. On the other hand it is legal to extend the function head over more than one physical line by using “.” as long as the continuation happens before the final parenthesis.

Here are some correct function heads:

```
function y = foo(x)
```

```
function y ..
    = foo(x)

function y = foo(a, b, c, d, ..
    e, f, g, h)
```

The following examples are all *illegal*:

```
function y = foo(x) // This is foo!

function y = save_space(x); y = 1 + x

function y = bar(x) ..
    y = 1 + x
```

See also Section 5.1.

2.4. Last Newline

In the Scilab versions prior to Scilab-2.6 (official release), the last line in a script or function is ignored if the line is not terminated by a newline. (On UNI* systems the line terminator is `^J`, or, written in C-style `"\n"`) This is emphasized at several places in the official Scilab documentation, but it is so common to forget it especially when using emacs that we repeat it here.

emacs can be told always to add a final newline by adding `(setq require-final-newline t)` to the startup-file, `.emacs` or `.gnu-emacs`. See “Learning GNU Emacs” [Cameron:1996], Table C-8.

For function files the workaround is as simple as it is elegant. For Scilab allows a function optionally to be terminated with the `endfunction` keyword. The keyword being optional it does not matter whether the last line is completely parsed.

Another weapon against this kind of syntax flaw, and a few other pesky things, is e.g. the Perl-script shown in Example 2-2, which fixes part of the format of a Scilab script.

Example 2-2. Canonicalization of Scilab files

```
use Text::Tabs;

while (<>) {
    chomp;                                # remove newline if there is one
    tr/\200-\377/ /;                      # map 8-bit chars to spaces
    s[\s+$] [];                           # kill whitespace at end of line
    $_ = expand $_;                       # convert tabs to spaces
    print "$_\n";                         # print adding a newline
}
```

2.5. Variable Lifetime And Scoping

2.5.1. Local Variable Scoping

Scilab's visibility rule for locally defined variables follow those of block structured languages:

Variables local to a block shadow all variables of the same name not local this this block.

When we say variable v “shadows” variable v' , we mean that v' is not accessible neither for reading nor for writing. What is available for manipulation is v .

Example 2-3. Shadowing of local variables

```
->deff('y = foo(x)', 'a = 2*x, y = a + 1')
->a = 1.0 // top level
a =
    1.
->foo(3.5)
ans =
    8.
->a
a =
    1.
->foo(a)
ans =
    3.
->a
a =
    1.
```

Example 2-3 demonstrates that variable a , which is local to function `foo` has no influence on variable a in the surrounding environment. Even calling `foo` with a variable named a does not break this rule.

As usual in block structured languages variables from *all* enclosing scopes can be accessed, unless they are shadowed. Example 2-4 shows usage of variable a from an enclosing scope.

Example 2-4. Accessing variables from the enclosing scope

```
->deff('y = bar(x)', 'y = a + 1')
->a = 1 // top level
a =
    1.
->bar(3.5)
ans =
```

```

2.
->bar(-1)
ans =
2.
->a = 2
a =
2.
->bar(-1)
ans =
3.

```

Now what *is* the “enclosing scope”? It is the call stack, Scilab scopes dynamically!

Example 2-5. Dynamic Scoping

```

// scoping in Scilab
deff('first_local()', 'x = "foo", second()');
deff('first()', 'second()');
deff('second()', 'disp(x)');

x = 1;
first_local()           // prints 'foo'
first()                 // prints 1

```

Example 2-5 deserves a close look. Dynamic scoping can be confusing for people used to e.g. C’s lexically scoped auto variables.

```

/* lexical scoping in C */

void first_local(void);
void first(void);
void second(void);

int x = 1;

int
main(void)
{
    first_local();           /* prints 1 */
    first();                 /* prints 1 */

    return 0;
}

void first_local(void)

```

```

{
    int x = 123;                /* warning: unused variable 'x' */
    second();
}

void first(void)
{
    second();
}

void second(void)
{
    printf("%d\n", x);
}

```

Now compare this to Perl¹:

```

# dynamical scoping with Perl's local variables

sub first_local { local $x = 'foo'; second(); }
sub first { second(); }
sub second { print "$x\n"; }

$x = 1;
first_local();           # prints 'foo'
first();                 # prints 1

```

Dynamic scoping is an inherently dangerous feature for it might not be obvious where a variable gets its value.

Let us look at functions which try to change variables from an enclosing scope.

```

->deff('y = baz(x)', 'a = 2*a, y = a + 1')
->a = 3      // top level
a =
3.
->baz(1)
ans =
7.
->baz(1)
ans =
7.
->a
a =
3.

```

1. The behavior of the C example is reproduced by replacing `local` with `my`.

Obviously, `a` is unchanged by the calls to `baz`. What happens is the following:

1. A local variable named `a` is created, and the contents of `a` from the enclosing scope is copied into it. Within `baz` the *local* `a` is changed.
2. When the thread of control leaves `baz` the previous value of `a` is restored.

In other words: A local variable *cannot* influence a variable of the same name in any enclosing scope. The only ways to “export” a – possibly modified – value is either via the list of return values, which is the preferred way, or with a `global` variable.

As strange as this may sound to programmers accustomed to languages that require an explicit declaration of all variables, this is a necessary feature in Scilab as variables are created when they are first written to (e.g. as in Perl and Python). If a local variable in a function would change a global variable or local variable of the same name in another function, adding a new function to an existing system or library became a major maintenance headache.

2.5.2. Global Variables

The `global` attribute of a variable `var` is often misunderstood. It does *not* place `var` in an all encompassing name space so that it could be accessed from everywhere without further ado. Instead, `global` places the variable `var` in a separate name space; separate from the interpreter’s name space, and separate from all local functions’ name spaces. — And this is only the first half of the story.

```
->v = -1
v =
- 1.

->global('v')

->who('global')
ans =
v

->clear v

->who('global')
ans =
v

->deff('y = useglobal()', 'y = v')

->useglobal()
!-error 4
undefined variable : v
at line 2 of function useglobal called by :
```

```
useglobal()
```

As promised, this is only one half. After saying `global var`, the variable lives in its new name space, but it cannot be accessed. Doh! To work with it, we must *import* it explicitly, using the `global` modifier again. Therefore, a slightly modified version of `useglobal` works.

```
->deff('y = useglobal2()', 'global v, y = v')
```

```
->useglobal2()
```

```
ans =  
- 1.
```

```
->v = 1 + 2*i
```

```
v =  
1. + 2.i
```

```
->useglobal2()
```

```
ans =  
- 1.
```

Now what if we want to access `v` from the interpreter level again? It must be imported just as it must be imported into any function.

```
->global('v')
```

```
->v  
v =  
- 1.
```

```
->v = 17 + 4
```

```
v =  
21.
```

```
->clear v
```

```
->useglobal2()
```

```
ans =  
21.
```

One last hint: global variables even survive a restart. If this is not desired, `clearglobal` should be called in the user's Scilab startup file, `~/ .scilab`.

```
clearglobal()
```

will clear all global variables.

2.5.3. Clearing Variables

During everyday programming it is not necessary to explicitly remove variables from the work space. All local variables of a function die on exit from that function anyhow, and the variables in the global name space usually do not need a special treatment.

However, there are conditions under which it is preferable to completely wipe out a variable. This happens if one needs to avoid a pollution of the name space e.g. while working with the list of all variables, `who('local')`. The correct command to kill the non-global variable `v` is

```
clear v
```

Note that there are no parentheses. The assignment

```
v = []
```

sets `v` to the empty matrix. It does *not* remove the variable from the workspace.

Global variables are cleared with the `clearglobal` function, whose syntax is the same as `clear`'s syntax.

There is no need to worry if you do not understand how and why to kill a variable. This feature is only needed in very rare occasions.

2.6. Dangerous Range Generation

Range generation with the colon-operator “:” (see also Section 6.1.3.1.1) holds ready an unpleasant surprise that is caused by the finite precision of Scilab's floating-point numbers.

Colon expressions are used most often with integral *initial_value*, *final_value*, and *stride*. These are the safe uses. However, all three parameters can be decimal fractions! Not all decimal numbers have a finite representation in the binary system. For example, 0.1 has the infinite binary representation

$$2\#1.\overline{10011}\#E-4,$$

where the bar over the last four bits denotes infinite repetition of this bit pattern. Of course numbers with an infinite binary representation must be approximated by finite binary numbers. In our example the approximation is

$$2\#1.1001100110011001100110011001100110011001100110011001100110011\#E-4,$$

for Scilab uses IEEE 754 double-precision reals, which carry a 53 bit mantissa.

For a detailed analysis of the floating-point induced problems, Example 2-6 rewrites a general colon-expression as a loop.

Example 2-6. Equivalent Representation of a Colon-Expression

```
v = initial_value : stride : final_value
```

translates into

```
v = []
if initial_value <= final_value
    x = initial_value
    while x <= final_value      – note the comparison
        v = [v, x]
        x = x + stride          – x accumulates rounding error
    end
end
end
```

The approximate binary representation can influence all three parameters *initial_value*, *final_value*, and *stride*. Furthermore, each addition of *stride* to *x* – as shown in Example 2-6 – can introduce a rounding error. To stay clean of the rounding errors, colon-expressions involving decimal fractions are best avoided.

A possible workaround is to generate a vector with an integral-only colon-expression, and then rescale the vector to the desired range. For example, if a vector from 1.0 to 8.0 in 0.7... (= 7/9) increments is wanted, a save expression is `(9 : 7 : 72) / 9.0`.

Another possible substitute for a fractional colon-expression is the built-in function `linspace(initial_value, final_value [, n = 100])` that generates a vector of *n* evenly spaced numbers starting and including *initial_value* up to and including *final_value*. See Section 6.1.3.1.2 for a detailed discussion of this function.

The following piece of code shows that the rounding can strike in surprising ways. The vector generated with a colon expression ranges from 1.0 to 2.0 - *x* in increments of 0.25. Both, 1.0 and 0.25 have exact binary representations, thus they do not introduce any rounding error, neither does the repeated addition of 0.25 to 1.0. The upper end of the interval is decreased in increments of $x = 2^i$. Again, 2.0 and 2.0 - *x* have exact representations for the chosen values of *i*.

```
function e = eps_hi
    // Return the smallest positive number E for
    // which 1 + e is not equal to 1. This equals one
    // ULP (unit least precision) for 1 <= |x| < 2.
    x = 1.0;
    while 1.0 + x ~= 1.0
        e = x;
        x = x / 2.0;
    end;
endfunction
```

```
function y = pow2(n)
```

```

    // Return 2^n, but do not rely on the built-in
    // exponentiation operator.
    y = prod(2.0 * ones(1, n))
endfunction

```

```

eps = eps_hi();
for i = 0:5
    printf("i = %d      ", i);
    disp(1.0 : 0.25 : (2.0 - eps*pow2(i)));
end

```

```

->exec("colon.sce");
i = 0
!   1.      1.25      1.5      1.75      2. !

i = 1
!   1.      1.25      1.5      1.75      2. !

i = 2
!   1.      1.25      1.5      1.75      2. !

i = 3
!   1.      1.25      1.5      1.75 !

i = 4
!   1.      1.25      1.5      1.75 !

i = 5
!   1.      1.25      1.5      1.75 !

```

An upper boundary of $2.0 - 2^2 \cdot \text{eps}$ yields a vector of length 5, whereas $2.0 - 2^3 \cdot \text{eps}$ as the upper boundary gives a vector of length 4. The conclusion is that the rounding procedure, which Scilab imposes on the upper boundary, considers all numbers smaller than 2.0 by less than $4 \cdot \text{eps}$ as equal to 2.0. However, when asked directly, Scilab notices the difference as it should be:

```

->2.0 - 4*eps == 2.0
ans =
F

```

Conclusion: Using colon expressions with fractional parameters is not recommended and `linspace` should be used instead.

Chapter 3. Programming Style

The one and only general guideline to good programming style is: “Make it clear!” And one might extend that to

Make it clear – first of all to *you*, and then to the poor persons that take over your project (after you have been fired, because of writing illegible code).

Every possible style feature of the language should be used to express the meaning of the code more clearly.

3.1. Spacing and Formatting

Although often underestimated, the format, i.e. the visual layout of the source code itself can greatly help in the understanding of the actions described therein.

3.1.1. Intra-Expression Spacing

We often run into code like this

```
x=a*c+(x-y)^2*b
```

This is not bad, especially when typed at the command line for one-time use. However, the expression is not as clear as it could be. It can easily be improved by making the precedence levels (see also Section 4.3) of the operators stand out, as e.g.

```
x = a*c + b*(x-y)^2
```

Now, the assignment is intuitively clear at first glance. We use the word “intuitive” here alert the reader of the consequences of incorrectly formatting an expression. Then our intuition will mislead us, as in

```
x = a * c+(x-y)^2*b
```

Ouch! This expression is evaluated differently from what it is telling us. We should call it a liar.

See also Section 2.2 for the influence of whitespace on the evaluation of dotted operators.

3.1.2. Line Breaking

Breaking a long expression into lines can improve its readability dramatically. It is particularly recommended for matrix definitions with the square bracket operator. See also Section 2.2.

For example

```
m1 = [ 1+%i  -1+%i; ..
      -1+%i   1-%i ]
```

is superior to

```
m1 = [1+%i -1+%i; -1+%i 1-%i]
```

If an arithmetic expression is split into lines the operator at which the split occurs always goes onto the *next* line. Preferred break points occur right before operators of equal precedence.

```
d2 = fact * (a/(a+d)*(b*(1-delta) + d*delta) - d) * (P./K).^theta
```

for example becomes

```
d2 = fact * (a/(a+d)*(b*(1-delta) + d*delta) - d) ..
      * (P./K).^theta
```

or

```
d2 = fact ..
      * (a/(a+d)*(b*(1-delta) + d*delta) - d) ..
      * (P./K).^theta
```

or more dramatic

```
d2 = fact ..
      * ( ..
          a / (a+d) * (b*(1-delta) + d*delta) ..
          - d ..
        ) ..
      * (P./K).^theta
```

3.1.3. Setting Brackets Apart

If spaces right inside parentheses or brackets of an expressions make the subexpression stand out more clearly, they should be used. That way

```
B(k) = a1 * exp(-b1*P(k)/K(k) + b2*Q(k)/K(k))
```

becomes

```
B(k) = a1 * exp( -b1*P(k)/K(k) + b2*Q(k)/K(k) )
```

3.1.4. Vertical Spacing

All previous formatting suggestions of this sections have been concerned with horizontal spacing, and indentation. Vertical spacing is as important as horizontal!

As sentences belonging together go into one paragraph and paragraphs are separated by one or more blank lines, Scilab statements that belong together go into one visual block and the blocks should be separated by single blank lines.

```
n = 1;
lo = 1.0 - n*2*epslo();           // lo = pred(1)
hi = 1.0 + n* epslo();           // hi = succ(1)

for k = 2 : 4
    x = lo : (hi - lo)/(k - 1) : hi;
    y = linspace(lo, hi, k);

    disp(size(x, "c") - k);
    disp(x - 1);
    disp(y - 1);
end
```

If blank lines tear apart blocks of code within functions, it might be preferable to separate pairs of adjacent functions by at least two blank lines.

```
function y = baseconv(x, b)
// Convert decimal fraction X into a base-B number. Each
// "digit" of the result is one element in the result
// vector Y. To get a monolithic string, apply strcat to Y,
// like
//      strcat(["#", baseconv(x, b), "#", string(b)])

    if x >= 0
        y = []
    else
        y = ["-"]
    end
    y = [y, baseconv_integral(int(abs(x)), b)]
    if frac(x) == 0.0
        return
    end

    if y == [] | y == ["-"]
        y = [y, "0"]
    end
    y = [y, ".", baseconv_frac(frac(abs(x)), b)]
```

```
function y = baseconv_integral(x, b)
// Convert decimal integer X >= 0 into a base-B number. Each
```

```

// "digit" of the result is one element in the result
// vector Y.
    if x < 0
        error("integer X (arg 1) out of range; X >= 0.")
    end
    if b < 2 | b > 36
        error("base B (arg 2) out of range; 2 <= B <= 36.")
    end

    if x == 0
        y = ["0"]
        return
    end

    y = []
    xv = abs(x)
    while xv >= 1
        r = modulo(xv, b)
        if r <= 9
            rs = string(r)
        else
            rs = code2str(r)
        end
        xv = int(xv / b)
        y = [rs, y]
    end

function y = baseconv_frac(x, b)
// Convert decimal fraction 0 < X < 1 into a base-B number. Each
// "digit" of the result is one element in the result
// vector Y.
    if x <= 0 | x >= 1
        error("fraction X (arg 1) out of range; 0 < X < 1.")
    end
    if b < 2 | b > 36
        error("base B (arg 2) out of range; 2 <= B <= 36.")
    end

    y = []
    xv = x * b
    max_mant = prod(2.0 * ones(1, 52)) // 2^52
    n = 1
    while xv > 0 & n <= max_mant
        r = int(xv)
        if r <= 9
            rs = string(r)
        else
            rs = code2str(r)
        end

```

```

        end
        xv = frac(xv) * b
        y = [y, rs]
        n = n * b
    end

    if n >= max_mant
        warning("loss of precision")
    end

function f = frac(x)
// Return the fractional part of X.
// int(X) + frac(X) == X for all X.
    f = x - int(x)
end

```

3.2. Indentation

Heavy indentation does not hurt! No, in fact it is a great help in finding out the control flow quickly. Let us start with a good example this time, Example 3-1.

Example 3-1. Function `whocat`

```

function s = whocat(cat)
// return all local variables, functions,
// etc. that are in category cat.

s = [];
nl = who('local');

for i = 1:size(nl, 1)
    execstr( 'typ=type(' + nl(i) + ')' );
    if typ == cat then
        s = [s; nl(i)];
    end
end
end

```

The `for` loop, and the `if` branch are immediately recognizable.

There are blank lines between the logical blocks of the function. They too aid the reader's comprehension of `whocat`'s inner workings. As a rule of thumb lines of code that achieve a sub-goal of the computation should be grouped together as sentences are grouped in a paragraph.

In longer functions the indentation becomes essential for the orientation of the maintainer. Here is a excerpt of a longer function, that would be terribly hard to understand if not massively indented.

```

i = 1;
j = 1;
while i <= n1 & j <= n2
    while i <= n1 & j <= n2
        if ~equ(lst1(i), lst2(j)), break, end
        i = i + 1;
        j = j + 1;
    end
    if i >= n1 | j >= n2, break, end

    icurs = i;
    while icurs <= min(n1, i+fuzz)
        if equ(lst1(icurs), lst2(j)), break, end
        icurs = icurs + 1;
    end
    if icurs <= n1 then
        if equ(lst1(icurs), lst2(j)) then
            // record element(s) missing from lst1
            for p = i : icurs-1
                this_diff = [lst1(p), string(-p)];
                diff = [diff; this_diff];
            end
            // re-sync
            i = icurs;
        end
    end
    ...
end // while

```

The complete listing of this function can be found in Chapter 10.

The last example also shows that we are switching between several style paradigms:

- Neither the “One statement per line” rule is followed consistently,

```
if equ(lst1(icurs), lst2(j)), break, end
```

could be

```
if equ( lst1(icurs), lst2(j) ) then
    break
end
```

- Nor is the intra-line spacing always consistent with the guidelines presented here:

```
for p = i : icurs-1
```

could be

```
for p = i:icurs-1
```

The Golden Rule is that there are no golden rules... This is best known under the term “freedom”.

3.3. Single Quotes vs. Double Quotes

Single or double quotes enclose literal strings in Scilab. The opening quotes must match the closing ones, otherwise single and double quotes can be used interchangeably.

The single quote, used as postfix operator, has the additional meaning of Hermitian (complex) transposition. This double use almost causes not problems, but if you want to play it extra safe, using double quotes for strings only adds clarity to your scripts.

3.4. Choice Of Control Structures

Though not recognized as that by all programmers, the flow control structures themselves are first class indicators of the code’s workings. We consider three important cases here.

1. `while` vs. `for`,
2. `if` vs. `select`, and
3. strict block structure vs. premature return.

3.4.1. `while/for`

Expressed in words a `for` loop tells us:

- We know exactly how many iterations we shall need before we start looping.
- Nothing in the loop body will change this.

Whereas the `while` loop says:

- We must check whether we should loop at all, and
- we have to re-check after each iteration whether we need another round-trip.

Corollary: The termination condition of a `while` must be influenced in the loop’s body.

Compare the next two code snippets, the first one calculating the average value of a vector of numbers, the second searching zeroes of a given function.

```
values = [1.0, 2.0, 3.0, 4.0, 5.0];
```

```

average = 0.0;
n = size(values, 'c');           // line 3
for i = 1:n
    average = average + values(i);
end;
average = average / n

```

From 7nbsp;3 on, we know the number of iterations, `n`, and we know that nothing will change that. Thus a `for` loop is adequate.

```

deff('[y, dy] = fun(x)', ..
    'y = -0.5 + 1.0/(1.0 + x^2), ..
    dy = -2.0 * x / (y + 0.5)^2');

x0 = 0.76;
[y, dy] = fun(x0);
while abs(y) > sqrt(%eps)
    x = y/dy - x0;
    x0 = x;
    [y, dy] = fun(x);
end;
x

```

Assuming that the function `fun`, and the start guess `x0` are supplied by the user, we do not know how many loops it will take for Newton's algorithm to converge, if it does converge at all. (In the example it does.) Here, the `while`-loop expresses this lack of a-priori knowledge.

3.4.2. `if/select`

The relationship between `if` and `select` bears similarity to `while` and `for`, respectively. In a `select` clause the different cases are known – and spelled out explicitly – before the thread of control enters the construct. There is a one to one relationship between the states of the selecting expression and the case branch taken. The `else` branch in a `select` works exactly as the `else` in an `if`.

```

function f = fibonacci(n)
// return n-th Fibonacci number

select n
case 0 then
    f = 1
case 1 then
    f = 1
else
    f = fibonacci(n - 1) + fibonacci(n - 2)
end

```

The `select` expression is not restricted to scalars, vectors for example work too:

```
function s = shape4(m)
// classify a 2x2 matrix according to its shape

select abs(m) <= %eps
case [%t %t; ..
      %t %t] then
    s = 'empty'
case [%t %f; ..
      %f %t] then
    s = 'diagonal'
case [%f %f; ..
      %t %f] then
    s = 'upper triangular'
case [%t %t; ..
      %f %t] then
    s = 'lower triangular'
case [%f %f; ..
      %f %f] then
    s = 'dense'
else
    s = 'general'
end
```

An `if` clause is more flexible than a `select` clause, but at the price of being less expressive. Whenever a whole range of values has to be covered the `if` clause is the only way to go, as is demonstrated by Example 3-2.

Example 3-2. Function `mysign`

```
function y = mysign(x)
// re-write of the sign-function, taking
// floating-point precision into account

if abs(x) < %eps
    y = 0.0
elseif x >= %eps
    y = 1.0
else
    y = -1.0
end
```

3.4.3. Strict Block Structure/Early Return

The paradigm of structured programming is: “Every block has one and only one entry point.” That is it! Nothing is said about the number of exit points. The purists often misinterpret the paradigm, demanding a single exit point, too. We prefer our freedom, and choose whatever we find adequate to the problem.

Here are two different implementations of an algorithm calculating the factorial of a given integral number.

```
function y = fact_block(x)
// faculty of x; block-structured version

select x
case 0 then
    y = 1
case 1 then
    y = 1
else
    y = x * fact(x - 1)
end
```

The two special cases 0, and 1 are tested separately, and the general case is handled in the `else` branch.

```
function y = fact_early_ret(x)
// faculty of x; early-return version

if x >= 0 & x <= 1 then
    y = 1
    return
end

y = x * fact(x - 1)
```

This version immediately returns after having treated the special cases, leaving the general case to the “rest” of the function. In this very short function the advantages of the early return are not striking, however they are if many special cases are to be handled. The “rest” of the function can then concentrate on the core of the problem without being obscured by deeply nested conditionals.

3.5. Size of a Function

There is a rule of thumb for the length of a C-function:

Linus Torvalds

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

In older versions of sci-BOT the reader found the following paragraph:

It is also true for Scilab functions with the exception that high level functions, or functions that are called from the command line directly should be harnessed, see Section 5.1.5. Therefore, they are usually much longer than just two screenful. Yet, their structure decomposes quite naturally into two parts: the argument checking, and the computation. What remains true is that a Scilab function too should do only one thing and do that well.

Meanwhile, the authors have slightly changed their minds, opting for short functions throughout. All functions should be short, no matter how much argument checking is done. If the argument checking code bloats a function definition, the checking code must go into a separate function or even separate functions in the case the things checked are unrelated. We follow one of the prophets of concise programming, Martin Fowler, and recommend his book “Refactoring” [Refactoring:1999]. Of particular interest in connection with this section are the refactorings “Decompose Conditional” (238), “Consolidate Conditional Expression” (240), and “Replace Nested Conditional with Guard Clauses” (250).

For more information about programming style consult “The practice of programming” [Kernighan:1999] which is centered around C-like languages, but offers extremely valuable advice throughout. “Programming Perl”, also known as “The Camel”, [Wall:1996] has a section called “Efficiency” in chapter 8. It is as insightful as it is fun to read for the authors discuss the various optimization directions. They do not hesitate to put up contradictory suggestions along the different paths.

Conclusion of this section: Whatever makes the code’s workings more obvious to the reader is good. In other words: “If it makes ya high, or saves you taxes, then – by any means – do it!”

Chapter 4. Unknown Spots

In this chapter we shed some light onto widely unknown features. Parts like the operator precedence unconsciously are exploited in every-day programming by all of us. Others, like integer variables are easily misused. So, read on and become a Yedi^H^H^HScilab master.

4.1. Keywords and Commands

The Scilab language protects only twelve words against any modification by the user. These identifiers cannot be used as variables or function names. Any attempt to do so immediately raises an error, which typically reads “incorrect clause”.

Table 4-1. Reserved Words

Name	Description
<code>break</code>	Force (premature) exit from a <code>for</code> or <code>while</code> loop
<code>case</code>	Start clause within a <code>select</code> statement
<code>do</code>	Synonym for “,” after <code>for</code> , <code>while</code> , <code>if</code> , etc.
<code>else</code>	Start alternative in an <code>if</code> or <code>case</code> statement
<code>elseif</code>	Add a conditional branch to an <code>if</code> statement
<code>end</code>	Terminate <code>for</code> , <code>if</code> , <code>select</code> , and <code>while</code> statements
<code>endfunction</code>	Terminate a function definition
<code>for</code>	Start a loop with a known number of iterations
<code>function</code>	Start a function definition
<code>if</code>	Start a conditional
<code>select</code>	Start a multi-branch conditional
<code>then</code>	Synonym for “,” after expression in <code>if</code> or <code>select</code>

Reserved words are protected against abuse by the interpreter, commands which follow in Table 4-2 are not! Some of the commands ought to be reserved words, but they are not. Commands can be used in contexts where variables are valid, however, the results are surprising. Therefore they should not be used as names for variables or functions.

Table 4-2. Commands

Name	Description
<code>abort</code>	Stop current evaluation and return to primary command-level
<code>apropos word</code>	Search for manual-pages whose synopsis matches <i>word</i>
<code>clear varname</code>	Remove variable (or function) <i>varname</i> from workspace; see also Section 2.5.3
<code>exit</code>	Terminate Scilab session
<code>help word</code>	Display manual page on topic <i>word</i>
<code>pause</code>	Switch into pause mode (can be used multiple times)
<code>pwd</code>	Print the current working directory
<code>quit</code>	Jump out of pause mode (can be used multiple times) or quit Scilab session
<code>resume</code>	Stop execution of a function or, in pause mode, return from function
<code>return</code>	Return from function
<code>what</code>	List all Scilab reserved words
<code>while</code>	Start a conditional iteration
<code>who('local' 'global')</code>	List local or global variables in workspace; see also Section 5.2.1
Some uses of <code>do</code> :	
<code>for i = 1:n do ..., end</code>	
<code>while i < n do ..., end</code>	
<code>if a < b do ..., else ..., end</code>	

4.2. Operator Overloading¹

Scilab bears a feature which strongly reminds one of object-oriented programming languages: operator overloading. Yet, Scilab is not object oriented. Strictly speaking operator overloading has nothing to do with object-oriented programming, but as it turns out overloading operators is

1. Thanks go to Bruno Pinçon for carefully proofreading this section.

particularly useful in object-oriented languages. Overloading grafts new functionality onto an existing function-name. Scilab accomplishes this with a special syntax similar to mangled symbol names of a C++-compiler's output.

The overloading of operators is described in Section 3.3 ("Definition of Operations On New Data Types") of in `SCI/doc/Intro.ps`, and information is also available through `help overloading`.

Even if you never will overload an operator, knowing the syntax and the function codes helps when deciphering error messages that involve overloaded operators. The following session transcript shows what happens if you request the boolean matrix `bm` to be converted into a string.

```
->bm = [%t, %f; %f, %f]
bm =
! T F !
! F F !

->string(bm)
!-error 246
impossible to overload this function for given argument type(s)
undefined function %b_string
```

Without further knowledge the user is nothing but puzzled by "undefined function %b_string".

4.2.1. Overloading crash course

If we drop the buzzword "Operator overloading", which comes from the OO-camp, and call every operator a function, we are (a) absolutely right in a mathematical sense, and (b) get a good grasp of what is going on. *Operators are simply functions written in a special syntax*. In most imperative languages (C and descendants, Pascal and descendants) functions are written in prefix notation, i.e. the function name precedes all arguments. In the same languages operators are written in infix notation, i.e. the operator put between the operands.

Lisp as a (functional) language which employs pure prefix syntax all functions and operators are written before all arguments.

```
(+ 3.9 54.0 -4.5 74.5 -57 -56)
(setq x (list "a" -1 (- 3 10)))
(length x)
```

The same expressions look more or less differently in Scilab:

```
3.9 + 54.0 + (-4.5) + 74.5 + (-57) + (-56)
sum( [3.9 54.0 -4.5 74.5 -57 -56] )      - alternative to previous line
x = list("a", -1, (3 - 10))
length(x)
```

As becomes clear in the above example, operators are specially written functions, but otherwise behave like ordinary functions.

Overloading has been hyped since the advent of C++. A closer look reveals that even Fortran-77 endows certain intrinsics with an overloaded syntax. What the heck is overloading? *To overload a symbol means assigning another meaning to it, augmenting the existing meaning(s)*. Typically, the symbol is a function name and the additional meaning is an additional function definition.

How can the language decide which definition to take? That depends on the language. The most common scheme to determine which function definition to trigger is the analysis of the *actual* function arguments. Fortran-77 provides so-called generic functions, `sin` is one example, which can be called with arguments of several types and the compiler selects the routine that matches that type.

```

program f77ovl

  implicit none

  real xr, s1
  double precision xd, s2
  complex xc, s3

*   floating point literals default to real*4 in f77
  xr = 1.0
  s1 = sin(xr)                                – compiler selects single precision routine

  xd = 1.0d0
  s2 = sin(xd)                                – compiler selects double precision routine

  xc = (1.0, 0.0)
  s3 = sin(xc)                                – compiler selects complex routine

*   alternative using explicit call
  s2 = dsin(xd)                                – user demands double precision routine
  s3 = csin(xc)                                – user demands complex routine

end

```

Modern languages like e.g. C++, F9x, and Ada let the user define functions with the same name as long as they can be uniquely identified by their argument list (C++) or argument list and return value (Ada).

We can define three `Maximum` functions. The Ada-compiler distinguishes them by their arguments and return values.

```

function Maximum(X1, X2 : Float) return Float;
function Maximum(F1, F2 : Fraction) return Fraction;
function Maximum(I1, I2 : ArbitraryPrecisionInteger) re-
turn ArbitraryPrecisionInteger;

```

As we know from the beginning of this section, operators are functions written in a special way. Thus it is easy to imagine that an operator can be overloaded just the way a function can. The next Ada example shows how the addition operator can be overloaded.

```
function "+"(Left, Right : Fraction) return Fraction is
begin
    return – code for addition of two fractions
end "+";
```

If you want to learn more about overloading and class construction and object oriented (C-) programming, we recommend Scott Myers' books [Myers:EffCPP:1998] , and [Myers:MoreEffCPP:1996] .

4.2.2. Overload syntax

In Scilab an operator gets overloaded with a new function, if we define this function having a special name in a particular format. For unary operators the format is

```
function result = %o $\text{type}$ _opcode(argument)
```

whereas for binary operators except insertion and extraction it is

```
function result = %o $\text{type}1$ _opcode_o $\text{type}2$ (argument1, argument2)
```

where valid operand-variable-type codes for *otype*, *otype1*, and *otype2* are defined in Table 4-3 and Table 4-4, and the operator-codes *opcode* are defined in Table 4-5. The formal function arguments *argument*, *argument1*, *argument2*, and *result* are usual argument and return-value names. To describe the syntax in words: a percent-sign starts the definition followed by the type[s] of the operands and the operator separated by [an] underscore[s].

The syntax for overloading vector/matrix insertion

```
target(index1, index2, ..., indexN) = source
```

and vector/matrix extraction

```
target = source(index1, index2, ..., indexN)
```

is a bit more convoluted as it has to account for the indices:

```
// insertion
function target = %targettype_i_sourcetype(index1, index2, ..., indexN, source, target)
// extraction
function [result1, result2, ..., resultM] = %sourcetype_e(index1, index2, ..., indexN, source)
```

Warning

The online-help of Scilab-2.5, **help overloading**, is incorrect in its explanation of the argument names to insertion-overloading. It says that *target* is the next-to-last, and *source* is the last argument. In fact the two arguments occupy exchanged positions as we have listed them.

Note that for extraction the number of return values, *M*, is completely independent of the number of index expressions, *N*.

Table 4-3. List of all operand type codes

Variable type	Code string	optype	Code index	
floating point scalar, vector, or matrix	s	a	1	
polynomial	p	a	2	
boolean	b	a	4	
sparse matrix	s	p	5	
sparse boolean matrix	s	p	b	6
Matlab® sparse matrix	m	s	p	7
matrix of integers (8, 16, or 32bit i entries)			8	
string	c	a	10	
uncompiled function	m	a	11	
compiled function	m	c	13	
function library	f		14	
untyped list	l	a	15	
typed list	name of the tlist	b	16	
matrix list	m	l	17	
pointer	p	t	r	128
?	i	p	129	

Notes: a. This type code is already overloaded by Scilab itself. b. The formal code string for typed lists is *t*1.

Two types are particularly well suited for overloading; these are the *tlist* and its close relative the *mlist*. *tlists* are used by Scilab itself to define some sophisticated types like polynomials, or sparse boolean matrices. Table 4-4 summarizes all types *t*, in use as of version 2.5.1. As the type of a *tlist* is the list's first element, we sometimes call it, in a Lisp like manner, the head of the *tlist*. Note that when working with *tlists* of type *t*, Scilab calls the predefined function for untyped lists, `%l_opcode`, or `%l_opcode_1` until the user provides [a] replacement function[s] with the name `%t_opcode`, or `%t_opcode_t`.

Note: Only the first 8 characters of the name of a `tlist` or `mlist` are significant when overloading any unary or binary operator! See also Section 4.6.1.

The first column of Table 4-4 states the name of the variable type, column two lists the `tlist` identification heads, and column three holds the code number, Scilab associates with the specific head. The type-name/type-code – not only for `tlists` – translation can be queried with the `typename` function.

Table 4-4. List heads used by Scilab

Variable type	Code string	Code index
sparse matrix	sp	5
sparse boolean matrix	spb	6
Matlab® sparse matrix	msh	7
linear-state system	lss	16
rational function, i.e. quotient of r two polynomials		16
hyper-matrix	hm	17
?	ip	129

Now that we have defined all operand type codes, we can turn to the operator type codes.

Caution

Several operators listed in Table 4-5 behave specially! Some, like equality and inequality tests, are auto-overloaded, i.e. are available even *before* the user defines her replacement function. Others *cannot* be overloaded at all, like unary plus (all types), and insertion/extraction (`tlists`).

Warning

There is a mistake in the `SCI/doc/Intro.ps` concerning this operator. The first table in Sec. 3.3, page 64 states that code `b` defines the row-separator “;”. `SCI/doc/Intro.ps` is wrong here, but the online help is correct in that the row-seperator “;” is overloaded with the code `f`.

Warning

The online-help of Scilab-2.5, **help overloading**, is incorrect, and `SCI/doc/Intro.ps` is correct. `u` is associated with “*.”, and `x` with “. *”.

Table 4-5. Operator type codes

Operator	Code	Note
<code>.'</code>	0	pure transposition (no complex conjugate)
<code><</code>	1	less-than
<code>></code>	2	greater-than
<code><=</code>	3	less-or-equal
<code>>=</code>	4	greater-or-equal
<code>~</code>	5	logical-not
<code>+</code>	a	binary operator. The unary plus is automatically overloaded for any new type with the identity-transformation or “do nothing”. Unary plus cannot be overloaded!
<code>:</code>	b	range generator
<code>[,]</code>	c	matrix row constructor “,”
<code>./</code>	d	element-wise division
<code>()</code>	e	extraction from a matrix, like <code>s = v(k)</code> . The operator is automatically defined for new types. Extraction from tlists cannot be overloaded, use mlists instead.
<code>[;]</code>	f	concatenation or matrix column construction “;”
<code> </code>	g	logical-or
<code>&</code>	h	logical-and
<code>()</code>	i	insertion into a matrix, like <code>v(k) = s</code> . The operator is automatically defined for new types. Insertion into tlists cannot be overloaded, use mlists instead.
<code>.^</code>	j	element-wise exponentiation
<code>.*.</code>	k	Kronecker multiplication

Operator	Code	Note
\	l	left division; solve a linear system of equations
*	m	matrix multiplication
<>, ~=	n	unequality test. Both operators are automatically defined for any new type with list semantics, i.e. component-wise comparison and a boolean return vector. Both can be overloaded with user functions.
==	o	equality test. The operator is automatically defined for any new type with list semantics, i.e. component-wise comparison and a boolean return vector. It can be overloaded with a user function.
disp	p	unary operator; display results with disp or at the command line
^	p	binary operator; matrix exponentiation
.\	q	element-wise left division
/	r	right matrix division
-	s	unary <i>%head_s</i> , and binary <i>%head1_s_head2</i> operator; see also: <i>overloading of unary plus %head_a</i> .
'	t	unary operator, Hermitian (complex) transposition
*.	u	element-wise multiplication
/.	v	element-wise division
\.	w	element-wise right division
.*	x	element-wise multiplication
./.	y	Kronecker division

Operator	Code	Note
<code>.\.</code>	<code>z</code>	Kronecker right division

Almost all unary built-in functions like `abs`, `ceil`, `floor`, `imag`, `int`, `real`, `round`, `sqrt`, and `string` can be overloaded, too. The syntax borrows for the syntax for unary operators. Function names which are not already used by Scilab cannot be used for overloading.

```
function result = %optype_functionname(argument)
```

where *functionname* is the name of the function.

4.2.3. Overloading example

Tip: Lots of overloading functions can be found in `SCI/macros/percent`.

After so much theory, definitions and tables we deserve an example that demonstrates operator overloading in action. As usual for sci-BOT the complete example can be found in Section 10.1. The following is not production strength code, most error checks are left out.

```
function f = frac(p, q, reduce)
// constructor for fractions

select type(p)
case 1 then // constant
    if size(p, '*') ~= 1 then
        error('argument p is non-scalar')
    end
    p0 = p
    q0 = 1
case 16 then // tlist
    // copy constructor behavior
    p0 = p('num')
    q0 = p('denom')
else
    error('argument p has wrong type')
end

if isdef('q') then // q is an optional argument
    select type(q)
    case 1 then // constant
        if size(q, '*') ~= 1 then
            error('argument q is non-scalar')
        end
        q0 = q0 * q
    case 16 then // tlist
```

```

        // copy constructor behavior
        p0 = p0 * q('denom')
        q0 = q0 * q('num')
    else
        error('argument q has wrong type')
    end
end

if isdef('reduce') then // (isdef('reduce') & re-
    duce == %t) does not work, for
        // Scilab performs a complete boolean evaluation
    if reduce == %t then
        [p_red, q_red] = reduce_int(p0, q0)
    else
        p_red = p0
        q_red = q0
    end
else
    [p_red, q_red] = reduce_int(p0, q0) - reduce_int defined in complete example
end
f = tlist(['frac'; 'num'; 'denom'], p_red, q_red)

function s = %frac_p(f)
// display function for fractions
s = string(f)
disp(s)

//
// addition
//

function r = %frac_a_frac(f1, f2)
d1 = gcd_int(f1('denom'), f2('denom'))
if d1 == 1 then
    r = frac(f1('num')*f2('denom') + f1('denom')*f2('num'), ..
        f1('denom')*f2('denom'))
else
    t = f1('num')*(f2('denom') / d1) + f2('num')*(f1('denom') / d1)
    d2 = gcd_int(t, d1)
    r = frac(t/d2, (f1('denom') / d1)*(f2('denom') / d2))
end

//
// conversion
//

function fl = frac2float(f)
// convert a fraction to a floating point number

```

```

fl = f('num') / f('denom')

function s = %frac_string(f)
// string( frac(...) )
if f('denom') == 1 then
    s = sprintf('%.0f', f('num'))
else
    s = sprintf('%.0f/%.0f', f('num'), f('denom'))
end

```

After loading these definitions a new type named `frac` exists. It can be used like this:

```

f = frac(2, 3);
g = frac(1, 3);
h = frac(-1, 3);
i = frac(12);

f + g
g + h
i

frac2float(h)

```

4.3. Operator Precedence And Associativity

Strange but true, there is no listing of the precedence and associativity of neither class of Scilab's operators anywhere in the documentation. So, we discuss the operator precedence and associativity in detail.

4.3.1. Numeric Operators

Table 4-6 shows the list of all numeric operators up to digraphs,² sorted in descending order of their precedence. An equal precedence value (column 1) means the operators are evaluated following the given associativity (column 3).

The table has been generated with a Scilab script, i.e., we had the interpreter determine its own precedence rules. These scripts are listed in Chapter 10.

2. The trigraph operators `. * .`, `. / .`, and `. \ .` are left out.

Table 4-6. Arithmetic Operators

precedence	operator	associativity	comment
21	+	right	unary
20	^	right	
20	.^	right	
19	-	right	unary
8	*	non	
8	/	left	
8	.*	non	
8	./	left	
4	\	left	
4	.\	left	
1	+	non	binary
1	-	left	binary

Warning

One line begs for an additional warning, and that is the unary minus ranking at level 19. It loses against the power operator, \wedge . Therefore, -1^2 gives -1 , and not 1 . In other words Scilab sees -1^2 as $-(1^2)$.

The association rules follow those of standard algebra. Thus, nobody should be surprised that a^b^c is interpreted as a^b^c .

4.3.2. Relational Operators

Scilab implements the usual gang of relational operators with some syntactic sugar of having two “unequality”-operators $<>$, and $\sim=$. The relational operators’ precedences rank in between the numeric and the logical operators like they do in many other modern programming languages. This allows for a minimal use of parentheses in larger expressions like

```
if 2.0*n > 1+1.0 | n/3.0 <= k then
    ...
end
```

which evaluates exactly the same way as

```
if ((2.0 * n) > (1 + 1.0)) | ((n / 3.0) <= k) then
    ...
end
```

just with much less line-noise.

4.3.3. Logical Operators

There are three logical operators: `&`, `|`, and `~`, meaning “and”, “or”, and “not”. The twiddle, `~` has the unique syntactic property that any number of consecutive twiddles are allowed and evaluated. But unless you want to enter the obfuscated Scilab contest, sticking with one probably is best as e.g. `15 ~` are as good as one, and therefore

```
~~~~~%t
```

returns F.

Table 4-7 shows the complete list of Scilab’s logical, also known as boolean, operators sorted according to decreasing precedence.

Table 4-7. Boolean Operators

operator	associativity	comment
<code>~</code>	right	unary
<code>&</code>	non	
<code> </code>	non	

4.4. Boolean Peculiarities

Scilab’s booleans are much more versatile than in most conventional programming languages. This section explains the enhancements that make the boolean type powerful.

4.4.1. Implicit Cast To Boolean

For the logical operators have boolean expressions as their arguments, it is time now to discuss the implicit promotion of numeric types to boolean type, something very familiar to C, Perl, and Python programmers. You have guessed right, the rule is: “Zero is false, everything else is true.” Here are some examples of that rule at work:

```
->%t & 0
ans =
F

->%t & 0.1
ans =
```

```

T

->6.34 | %f
ans =
T

->6.34 | -0.3
ans =
T

```

Scilab always evaluates boolean expressions completely. No operator is defined with short-circuit evaluation semantics.

```

->deff('b = ret_false()', 'b = %f, disp("ret_false")');

->ret_false() & ret_false()

ret_false

ret_false
ans =

F

```

4.4.2. Boolean Vector- or Matrix-Indices

Booleans are valid indices for vectors or matrices. Both, the host object, which is indexed, and the index itself are used in their flattened representation (Section 6.1.2.3). A boolean `%t` at position i selects element i from the host object; `%f` does nothing.

If the size of the boolean index does not match the host object's, missing indices are implicitly assumed to be `%f`. Extraneous that are `%f` do not produce a runtime error, only `%t` index values at positions after the host object's end.

```

->a = [11, 12; 21, 22; 31, 32]
a =
!   11.    12.  !
!   21.    22.  !
!   31.    32.  !

->a(%t)
ans =
    11.

->a([%t, %t; %f, %t])
ans =
!   11.  !

```

```
! 31. !
! 12. !
```

4.5. Integers

by Enrico Segre

Integer types were introduced in Scilab-2.5 (official release); they are an important concept, but to date their support still is incomplete and partially buggy. In many situations the use of integer variables can provide dramatic storage improvements; moreover, large problems, for example those occurring in image manipulation, often fit the hardware constraints when integer storage is exploited. Thus, even though the integer types in Scilab still leave something to be desired, their use may be a matter of necessity; and even considering that integer support is largely broken, yet, the existing possibilities can provide workable solutions. The following section is a guide to what is available and what is not when it comes to integer expressions.

4.5.1. Missed Opportunities

The following spots are – to our opinion – missing parts in the current implementation of integers.

4.5.1.1. No Integer Literals

Integer constants can only be defined as results of an `intN` function ($N = 8, 16, \text{ or } 32$) with a real argument. No special notation exists for integer literals as for example `123#` or `!123`. Variables are declared as integral when they are assigned an integer value. The integer value has to be produced first, and this is only possible with a function.

This is inconvenient, and often also performance critical, for instance when defining large integer arrays. The requirement of duplicate storage for passing by value and the calling overhead can be demanding. For example,

```
ia = int8(modulo(1:1e6, 16))
```

produces the array `ia` that occupies 1 MB of RAM. Even though, the definition procedure requires an intermediate storage of 24 MB (IEEE 754 double-precision reals have a size of 8 bytes each): 8 MB go for defining `1:1e6` and 16 MB for passing by value the result to `modulo` and to some other internals of `modulo`. Scilab goes on a detour in the construction of integral variables instead of attacking this area directly: the parser ought to recognize terminal symbols making up integral expressions, so that no double-precision intermediate result is called into play. The pitfall lies partly in the missing notation itself, and partly in the need to do the integer conversion only at the last step of the evaluation, for lack of usable integer constructs (see Section 4.5.2.1).

4.5.1.2. No Implicit Conversion in Mixed Typed Expressions

Altogether, the introduction of integers has brought 6 new data types:

- int8,
- uint8,
- int16,
- uint16,
- int32, and
- uint32.

Scilab generally does implicit type conversions in expressions involving reals, booleans, and several other types, but *not* when at least one of the operands is an integer. Automatic conversions – for example, the result of an addition of an int8 and an int16 becomes an int16, an int16 plus a real makes a real, etc. – are *not* implemented. In some programming languages, strong typing can be a design decision; here, it is probably just a lack. The only automatic conversion takes place when assigning a real value to elements of an array, which has been predefined as integral. Then, the right hand side is silently cast to the left hand side's type.

```
->a = int8(zeros(1, 8));
->a(2:4) = 5.3
a =
!0  5  5  5  0  0  0  0 !
```

In addition to the lack of automatic type conversion, a few bugs involving mixed type expressions are exemplified below (see Section 4.5.2.3.2).

4.5.1.3. No Integer Array Indices

Indexing of array elements is a classical use for integers. However, Scilab solely supports double-precision, and not integer-typed indices for arrays and hyper-arrays. In many situations juggling integer indices would be more memory efficient than dealing with double precision. The double-precision indices finally have to be (internally) cast to integers to actually index into an array. Consider for example

```
a = rand(1, 1e6)
a = a(1e6:-1:1)
```

The reversal of the elements of the array requires 24 MB: 8 MB for storing a, 8 MB for storing the right hand side of the assignment, and 8 MB for storing the index expression `1e6:-1:1`. If int32s were used instead, half a megabyte could be saved.

Indexing Done Right

If indexing were done right, it would not require any additional core. The stride of `-1` magically turns your index expression into a efficient call to `dcopy`.

4.5.1.4. Limited Support of Integers in Functions

Only a small subset of the functions which work on reals, or of the syntactical constructs which involve reals, can be applied to integers. Which functions support integers and which not do not, does not follow a rule – it just looks like unfinished work. A practical account is given in Section 4.5.2.1.

4.5.1.5. Partial Support of Integer Values in Data Files

Reading and writing of integer data from and to data files is still imperfect. As for reading: in Scilab-2.5 (official release), values retrieved with `mget` or `read` from external data files always are rendered as double precision reals. Only afterwards they can be converted to integers. This once more carries the disadvantage of the real (no pun intended) detour, as discussed in the previous section. An external datafile containing many short integers might not be loadable, because the data are expanded to double precision reals, filling the available memory, though, once reconverted to short integers, the data would fit. From Scilab-2.5.1 (alpha version) on, function `mgeti` exists, and is well suited for integers stored in binary files, but no integer equivalent of `read` yet exists.

As for the complementary operation, writing integers into a binary file, function `mput(data, type)` has been present before Scilab-2.5 (official release). There, however, `mput` accepted only real data, even though data could be written into the file as an integer of any type, if specified. Only from Scilab-2.5.1 (alpha version) on, it has become possible to pass integers to `mput`. Actually, in Scilab-2.5.1 (alpha version) there were still a couple of bugs lying around: when integer data was output, extensive garbage was printed, and explicit reference of the unit number was impossible. So, in Scilab-2.5.1 (alpha version):

```
->fd = mopen('my_file', 'wb');
->mput(int8(1:1000), 's', fd);
      !-
error 201 : argument 3 should be a real or complex matrix
->mclose(fd);
```

while

```
->fd = mopen('my_file', 'wb');
->mput(int8(1:1000), 's');
->mclose(fd);
```

worked, but printed a lot of output to the console, considerably slowing down the computation. Both of these bugs are ironed out in Scilab-2.6 (official release).

In contrast, `load` seamlessly retrieves integer values, if the corresponding `save` wrote them as such.

4.5.2. Digest of Integer Go And No-Gos

With integers, some Scilab constructs work, some simply do not, and others apparently work, but incorrectly, and are thus best avoided. If a user *is forced* to use integers, she needs a road map to what is viable and where to stay away from. The following considerations can help in surviving with integer data.

4.5.2.1. Which Functions Support Integers?

Plainly, some Scilab functions work as expected with integer arguments, and some do not. In many cases this seems a matter of lazily done homework or homework not done at all. The proper overloading alternatives to the real constructs are missing! We cannot give any general rules, except for these two:

1. Functions that can give a real or complex result with an integer input, for example `sqrt` or `spec`, in most of the cases do not accept integer types.
2. It is naive to expect any function or expression which relies on indices or index counts to work with integer enumerators.

Table 4-8. Selected Functions and Operators That *Work* With Integers

Operator or Function	Comment
"+", "-", "*", "/", "^", and "'"	their dotted cousins are also working
":"	colon operator used as implicit or explicit indexer of integer arrays, with real indices. For example, <code>il = int8(1:10); il(:)</code> is accepted.
<code>min(ival)</code> , <code>max(ival)</code> , <code>matrix(ival)</code> , <code>hypermatrix(zval, ival)</code> ,	returning integer values <i>ival</i> of the same type as their arguments; <i>zval</i> is real or complex.
<code>size</code>	real result!
<code>eye(ival)</code> , <code>ones(ival)</code> , <code>zeros(ival)</code>	real result!
<code>eye(ival)</code> , <code>cumsum(ival)</code> , <code>sum(ival)</code> ,	integer result
<code>disp(ival)</code>	string result
<code>fft(ival)</code>	complex result

In this section, “not work” means that Scilab complains with an error, usually about a wrong argument type or a missing overload function.

Table 4-9. Selected Functions and Operators That *Do Not Work* With Integers

Operator or Function	Comment
<code>“:”</code>	colon operator used as binary or ternary range generator
<code>length(ival), mean(ival)</code> <code>eye(ival1, ival2), ones(ival1, ival2), zeros(ival1, ival2)</code> <code>sqrt(ival)</code> <code>cumprod(ival), prod(ival)</code> <code>ceil(ival), floor(ival), int(ival), modulo(ival)</code> <code>gsort(ival), lex_sort(ival), sort(ival)</code>	where <i>ival1</i> and <i>ival2</i> are integer variables These are all real-to-real functions!
Sparse integer matrices are not supported at all.	

4.5.2.2. Modular Integers

Unsigned integer expressions *cannot overflow*; in particular, no warnings are issued. The result of an expression involving unsigned integers is always computed with respect to the modulus of the type.

```
->uint8(129) + uint8(129)
ans =
2

->int16(32769)
ans =
-32767
```

This is not surprising, but has to be kept in mind when doing integer calculations.

Background Information

On our days hardware, integer arithmetic is almost always done modulo 2^{width} , where *width* is the number of bits (typically 32 or 64) to represent an integer as a two's complement. This behavior kind of “leaks through” from the central processing unit (CPU), where neither integer overflow nor underflow exists. The main reason for implementing modular integers is speed. Implementing integers as range-checked type would incur a vast overhead and massively hurt performance.

However, integer divisions by integer zero are trapped, even when setting `ieee(2)`:

```
->ieee(2);
->int8(4) / int8(0)
!-error 27
division by zero...
```

Incidentally, `intN(%nan)`, `intN(-%inf)` and `intN(%inf)`, where N is 8, 16, or 32 all return 0. The same holds for all `uint` functions.

No overflow or underflow warning is reported either, if a longer integer is converted to a shorter one, whereas no loss of precision ever occurs when any kind of integer is cast to a real, because real mantissas (also known as significant) are represented by more bits – 52 to be precise – than any Scilab integer type.

4.5.2.3. Troublesome Spots

Integer types are a nice idea, and were definitely missing to Scilab before Scilab-2.5 (official release), but this said, we regretfully continue with our list of bugs. Unfortunately, it is not just a matter of implemented versus unimplemented integer constructs. Even some seemingly working constructs are problematic. Short of discouraging the use of integers types altogether, we go on reporting some troublesome spots, hoping that they will be addressed in future releases. We point out alternatives where appropriate.

4.5.2.3.1. Array Concatenation

In Scilab-2.5 (official release), there were serious bugs, which gave rise to wrong results even in the simplest concatenations of integer arrays. For instance,

```
->[uint16(1), uint16(2)]
ans =
!1  0 !

->[ans, ans]
ans =
!1  0  2  0 !
```

Similar things happen with `int8` and `uint8`, but not with `int32` and `uint32`. These bugs appear to have been corrected in subsequent versions of Scilab-2.5 (official release).

4.5.2.3.2. Mixed Type Expressions

Here, anything can happen, depending on the context and on the Scilab version. Most of the time, overloading functions (see also Section 4.2) for operators that involve two different types are undefined. Consequently, errors result from calling them. In several cases, however, wrong results show up.

```
->int16(10) .* 3.2
!-error 4
undefined variable : %i_x_s
```

The proper overloading function, integer-times-real `%i_x_s`, for the “`.*`” operator is missing, and this is reported as an error. If, however, the user enters

```
->int16(10) * 3.2
ans =
30
```

in Scilab-2.5 (official release), while

```
->int16(10) * 3.2
ans =
-32678
```

in Scilab-2.5.1 (second beta version), and

```
->int16(10) * 3.2
ans =
4
```

in Scilab-2.6 (official release).

Among the numerical operators, “ \wedge ” is a little more sophisticated. Mixed power operations are often correct, they also retain the type of the integer operand for positive integer exponents, while they give a real or complex result if the exponent is negative or non-integral. Thus,

```
->int16(2)^(-4)
ans =
0.0625
```

```
->int16(4)^(1/2)    – exponent is real!
ans =
2.
```

```
->4.0^int16(2)
ans =
16
```

```
->typeof(int8(4)^int16(2))
ans =
int8
```

```
->int16(-4)^0.5
ans =
1.225E-16 + 2.i
```

All is OK here? Well, not all doughnuts come out with a hole.

```
->uint16(-4)^0.5
ans =
255.99219
```

What about booleans? When booleans enter the game, the standard behavior in mixed real boolean expressions is to treat %t as 1.0 and %f as 0.0 (see also Section 4.4.1).

```
->%t + 1.0
ans =
    2.
```

Not so with integers! Most of the time the user again runs into missing overloading functions. In Scilab-2.5 (official release), however, the door was open to further bugs and oddities, which have been addressed in the later versions. For instance, operations with `int8`'s were accepted, but the results did depend on the order of the operands.

```
->%t + int8(1)
ans =
    T
```

```
->int8(1) + %t
ans =
    1
```

```
->int8(1) + %f
      !-error      4
undefined variable : %i_a_b
```

Other integer types triggered undefined overload functions, reporting errors. Fortunately that is what happens in any case from Scilab-2.5.1 (alpha version) on. Moreover, sneaking through the definition holes of Scilab-2.5 (official release), the game went on with even stranger results, which changed after each call. The following example was reported by Tom Bruhns <tom_bruhns@agilent.com>.

```
->f1 = %t + int8(0:20)
f1 =
! T T T T T T T T T T T T T T T T T T T T T !

->f2 = %t + int8(0:20)
f2 =
! T T T T T T T T T T T T T T T T T T T T T !

->f1 == f2
ans =
! T T T T T T T F F F F F F F F F F F T F T F !
```

Oh, maybe it was my imagination that `f1 == f2` did not make all “T” results ...

```
->f1 == f2
ans =
! T T T T T T T F F F F F F F F F F F F F F F !
```

What, not even the same answer as one lines ago? Ouch! Does this build confidence, or what?

Upshot. Avoid mixed typed expressions like the plague, at least for the moment; avoid them harder if you are still using Scilab-2.5 (official release). Peruse the `double`, `intN`, and `uintN` converters (or `iconvert`) as often as needed.

4.5.2.3.3. Mixed Type Comparisons

Up to the latest Scilab-2.6 (official release), comparisons between values of different types (doubles, integers) are allowed. However, the results are not always consistent. This is yet another example of mixed type expressions, now with relational operators. For instance, comparing a real scalar or vector with a real scalar is valid.

```
->(1:2) > 1
ans =
! F T !
```

This is the standard behavior. Trying to do with integers, you enter dangerous grounds. Comparing scalar integers of the same type is safe.

```
->int16(9) > int16(8)
ans =
T
```

```
->int16(9) < int16(8)
ans =
F
```

Sometimes even comparing different types yields correct results, as, for example,

```
->int16(1:2) > int32(1)
ans =
! F T !
```

```
->int16(2) > 1
ans =
T
```

This would suggest that some sort of type conversion takes place before the comparison, however, up to Scilab-2.5.1 (second beta version) this impression is wrong.

```
->int16(1:2) > 1
ans =
! F F !
```

To put it another way, maybe this result is correct, as both `int16(1)` and `int16(2)` appearing on the left hand side are different from 1, which is a double precision real value! But this latter interpretation is inconsistent with the two examples above, which is disturbing. In Scilab-2.6 (official release),

```
->int16(1:2) > 1
ans =
! T T !
```

which is different, wrong, and not even amenable to the previous interpretation.

Similarly, consider the different behavior of a (meaningless) comparison of real and complex.

```
->%i > 1
!-error      4
undefined variable : %s_2_s
```

Fine! Now comparing an integer with a complex does neither produce an error, nor a correct result:

```
->%i > int16(1)
ans =
F

->%i < int16(1)
ans =
T

->%i == int16(0)
ans =
T

->-%i == int16(127)
ans =
T
```

with small differences depending on the actual Scilab version.

4.5.2.3.4. Vector-Scalar Comparison of Identical Type Integers

Here too, bugs are lurking under the surface. In principle an array can be compared to a scalar, resulting in a boolean array of the size of the former. When doing that with integers of the same type, the results can be wrong, in a way which strangely seems to be more related to indexing than to comparison.

```
->ia = int16(1:20);

->ia > int16(21)
ans =
! F F F F F F F F F F F F F F F F F F T T !
```

The last two entries of the boolean result are plain wrong, even though inspection proves that the corresponding elements of `ia` are correct. If instead, the elements of `ia` are explicitly referenced,

```
->ia(1:20) > int16(21)
```

```
ans =
! F F F F F F F F F F F F F F F F F F !
```

the answer is correct. On the other hand, wrong results are also returned by expressions as `ia(:) > int16(21)`, `ia(1:$) > int16(21)` and `int16(21) < ia(1:20)`.

Upshot. It seems that the only relational expressions one can really trust are either *int_array* *relop* *int_scalar*, with identically-typed operands and explicit reference to the array elements, or *int_array* *relop* *int_array*, with equally sized and identically-typed arrays.

4.5.2.3.5. System Dependence of Type int8

On GNU/Linux PPC, we found that the range of type `int8` is identical to that of `uint8`; both assume values from 0 to 255.

```
->int8(-1)
ans =
255
```

However, Scilab regards the two as different types, and refuses to evaluate expressions involving both of them.

```
->int8(1) + uint8(1)
!-error 4
undefined variable : %i_a_i
```

4.5.2.4. Integers in Bitwise Operations

To conclude with something functional: the operators “~”, “&”, and “|” can be used in integer expressions. In this case, they act on the single bits of the representation of the integer value.

```
->~uint16(1)
ans =
65534

->~int16(1)
ans =
-2

->int16(1) | int16(4)
ans =
5
```

Bitwise and/or of two different integral types is not possible.

Bitwise operations are a bonus, when programming hardware at the register level. This is a case often encountered in interfacing with external instruments such as data acquisition cards.

To print integer values as hexadecimal strings, function `dec2hex` exists. Though, funnily, `dec2hex` is meant to accept reals as its only arguments. As previously mentioned for integer constants, no special notation exists for hexadecimal values; the function `dec2hex`, and its dual `hex2dec`, are mere formatting functions.

An alternative approach to bitwise operations, that might allow greater flexibility than `intN` operations, is the following. Binary strings can be represented (wasting memory) by boolean arrays. For example, for 8 bit strings, to fix the idea:

```
b8 = [%t %f %t %t %f %f %f %f]    // for 10110000
```

First define a suitable vector with the powers of two.

```
pow2 = 2^(-1:0)
```

which is used in boolean to integer conversion

```
s = sum(pow2(b8))
```

and integer to boolean conversion

```
d2b = zeros(1, 8)
```

```
for i = 1:8
    d2b(i) = int((s - d2b*pow2) / pow2(i))
end
```

```
b8 = d2b==1
```

Logical “and” and “or” operations map onto the usual logical expressions

```
c8 = a8 & b8
d8 = a8 | b8    // etc.
```

and even bit shifts can be written clearly with vectors.

```
e8 = b8([2:8, 1])
f8 = b8([8, 1:7])
```

Such an approach has some advantages and some disadvantages. The main advantage is direct access to a single bit, while the disadvantage is the larger memory consumption, the use of an extra array dimension, and the need of time consuming boolean to integer conversion functions.

4.6. Miscellaneous Unknown Spots

4.6.1. Maximum variable name length

Scilab accepts variables names that are longer than 1024 characters, but only the first 24 characters are significant. The identifying string of a `tlist`, or `mlist` can have more than 1024 characters, all of which are significant. The `tlist/mlist`-identifier length limit when overloading functions (see Section 4.2.2) with the percent-syntax is 8 characters.

4.6.2. Starting `scilex`

For debugging purposes it is sometimes desirable to directly start the main Scilab binary, **scilex**. Scilab is usually launched via the `scilab` shell script. Both, the script and the binary live in the `SCI/bin` directory. The script takes care of setting all environment variables, and finally fires up **scilex**. On the other hand, if one wants to run a debugger, say **gdb**, or **ddd**, or a profiler on Scilab, then a manual invocation is the order of the day. See also Section 8.1.2.

Starting **scilex** directly is an option as long as the command-line editing goodies are not required, and there is no need for any graphics. Actually, for minimum functionality only the environment variable `SCI` must be set, then we are all set to call **scilex**. A **bash** sequence to start Scilab “manually” could look as shown in Example 4-1.

Tip: From Enrico Segre: Under Win*, **runscilab** can be called from DOS prompt much as **scilab** is in UNI*, e.g., **runscilab -nw**. The DOS output of commands invoked with `unix` go to the shell window.

Example 4-1. Manually launching `scilex`

```
lydia@orion:~$ cd /site/X11R6/src/scilab
lydia@orion:/site/X11R6/src/scilab$ SCI=`pwd`
lydia@orion:/site/X11R6/src/scilab$ export SCI
lydia@orion:/site/X11R6/src/scilab$ cd bin
lydia@orion:/site/X11R6/src/scilab/bin$ ./scilex -nw
=====
S c i l a b
=====
```

Scilab-2.5
Copyright (C) 1989-99 INRIA

Startup execution:

```

loading initial environment

->

or shorter

lydia@orion:~$ export SCI=/site/X11R6/src/scilab
lydia@orion:~$ $SCI/bin/scilex -nw
=====
S c i l a b
=====

Scilab-2.5
Copyright (C) 1989-99 INRIA

Startup execution:
loading initial environment

->

```

where we are assuming that Scilab is installed in `/site/X11R6/src/scilab`.

4.6.3. Tuple Assignment

The most commonly used form of assignment is single variable assignment. Nonetheless, assigning multiple values in one statement is possible (and no surprise for Perl or Python programmers).

```

->[x1, x2, x3] = (1, 2, 3)
x3 =
3.
x2 =
2.
x1 =
1.

```

Tuple assignment works as expected, performing the whole assignment operation in one single step.

Warning

In version 2.5 the online documentation, **help parents**, gives the following, wrong explanatory code:

```
[x1, x2, ...] = (e1, e2, ...) is equivalent to x1
= e1, x2 = e2, ...
```

The correct explanation is

```
[x1, x2, ...] = (e1, e2, ...) is equivalent to first
performing %t1 = e1, %t2 = e2, ..., and then x1 =
%t1, x2 = %t2, ..., where the variables %ti, i = 1, 2, ...
are invisible to the user.
```

To prove that tuple assignment works as promised, we swap the values of two variables `a` and `b`.

```
->a = 1, b = 2
a =

1.
b =

2.

->[b, a] = (a, b) // swap
a =

2.
b =

1.
```

What one might expect, but what does not work is multiple assignment to parts of matrices (or lists), i.e. the following code snippet does not work as naively expected

```
->v = [0, 0, 0], a = 0
v =
!  0.    0.    0. !
a =
0.

->[a, v(1)] = (1, 2)
Warning: obsolete use of = instead of ==
!
!-error 41
incompatible LHS
```

The obvious, but ugly workaround is using only scalar variables on the left-hand side of a aggregate assignment, and then assigning these scalars to the appropriate matrix or list parts.

4.6.4. Dot as Member Selector

Scilab provides two different syntax constructs for the symbolic extraction/insertion of elements of a tlist. (The extraction/insertion by index numbers follows the extraction/insertion) of elements from matrices.) The documented syntax uses parenthesis and a selector string which has been defined for the specific tlist.

```
->d = tlist(["dict", "key", "value"], "snafaz", 3.0 + 4.0*i)
d =

      d(1)
!dict key value !

      d(2)
snafaz

      d(3)
3. + 4.i

->abs( d("value") )
ans =
5.
```

The alternative syntax uses dots “.” to separate the name of the tlist-variable from the name of the element, uncluttering the code.

```
->-real(d.value) + imag(d.value)
ans =
1.
```

The advantages of the string notation are: (i) The element-names can contain whitespace. (ii) Extraction/insertion under program control is easier.

Everything in this sub-section applies to mlists, too.

Chapter 5. User Functions

This chapter treats Scilab's most powerful code abstraction: functions. The first section, Section 5.1, introduces in the darkest details of user-defined functions. The second section, Section 5.2, treats libraries of user-defined functions.

5.1. Functions

Functions are Scilab's the main feature for the abstraction of programming tasks. Thus, they deserve a closer look.

See also Section 2.3

5.1.1. On(e)line Function Definitions

Scilab allows functions to be defined online, this is, at the command line, in two different forms. The first form uses the builtin function `deff`

```
deff(function_head, function_body [, compile_flag])
```

where *function_head*, *function_body*, and *compile_flag* are character strings. Most often these strings are given literally, for example,

```
deff('y = heavyside_theta(x)', 'if x <= 0, y = 0, else y = 1, end')
```

If *function_body* contains statements that include literal strings themselves, the quotes of these strings must be doubled, creating a hard to understand mess. This quoting disaster is avoided by using the second form of online function definition, which uses the keyword `function` and the syntax of function files (`".sci"`).

```
function function_head, function_body, endfunction
```

The crucial difference between `function` in a function file and in an online definition is that in a file the `endfunction` keyword is *optional*, whereas it is *mandatory* in an online definition.

```
function a = row_avg(m), s = sum(m, 'rows'), a = sum(s)/size(m, 'cols'), endfunction
```

The definition of a function with the `function` and `endfunction` keywords does not have to fit on a single input line. It can span multiple lines as the interpreter goes into "function definition mode" when it parses `function`. This mode resembles multi-line input in command shells and Python (though Scilab does not change its prompt to notify the user).

```
->function y = foo(x)
->    y = 1.0 + x + 0.5*x^2
->endfunction
```

```
->foo(4)
ans =
    13.
```

Both forms allow for nested function definitions – see the following section, Section 5.1.2.

5.1.2. Nested Function Definitions

Function definitions can be nested. The usual scoping rules apply. Online nested function definitions with `deff` are possible, but some kind of awkward, because of the massive number of quotes. `deffs` in functions are easy to the eye.

Example 5-1 shows a function that defines four functions in its body.

Example 5-1. Function `tauc`

```
function [t, rmin, r0] = tauc(E0, M, s, D)
    // Compute the round-
    trip time t, the minimum distance rmin and the point
    // of vanishing potential for a point-
    like particle with kinetic energy (at
    // r -> infinity ) E0, mass M in a Morse potential of steepness s and
    // depth D.

    // Morse potential
    deff('U = Umorse(r, steepness, depth)', ..
        'e = exp(-r * steepness); ..
        U = depth*(e^2 - 2*e)')

    // point of vanishing potential
    deff('y = equ0(x)', 'y = Umorse(x, s, D)')

    // reflection point
    deff('y = equ1(x)', 'y = Umorse(x, s, D) - E0')

    deff('tau = integrand(x)', ..
        'tau = sqrt( M / (2*(E0 - Umorse(x, s, D))) )')

    // rationalized units...
    units = 10.0e-10 / sqrt(1.380662e-23 / 1.6605655e-27)

    // calculate endpoints of definite integral
    r0 = fsolve(-10.0, equ0)
    rmin = fsolve(-10.0, equ1)

    // evaluate definite integral
    [t_unscaled, err] = intg(rmin, r0, integrand)
```

```

    t = 2 * units * t_unscaled
endfunction

```

As of Scilab version 2.6, nested functions do not work reliably, therefore, constructs like

```

function foo
    function bar
        ...
    endfunction
    ...
endfunction

```

should be avoided by using `deff` when defining `bar`. The fingerprint of nested functions is error 37, “incorrect function at line ...”. The do-not-nest limitation is raised for *one-liners*, where nesting works without problems.

```

->function y = foo(x), ..
->    function a = bar(b), ..
->        a = 1.0 + 2.0*b, ..
->    endfunction, ..
->    y = bar(x) / x, ..
->endfunction

```

5.1.3. Functions Without Parameters or Return Value

The “Introduction to Scilab”, `SCI/doc/Intro.ps`, solely explains functions that have one or more parameters, and return one or more values. Yet, Scilab permits all conceivable combinations of number of parameters and return values, including functions that have no parameters, or no return values.

If only one value is returned the square brackets in the function definition are optional. Therefore, the function head

```
function [y] = foo(x)
```

can be abbreviated to

```
function y = foo(x)
```

However, this is 100% pure syntactic sugar. What is much more important – and a valuable feature – is the possibility of defining a function that returns nothing as

```

function ext_print(x)
printf("%f, %g", x, x)

```

does. In Fortran parlance `ext_print` would be called a SUBROUTINE, whereas Ada programmers would term it a procedure.

Of similar importance is the definition of parameterless functions.

```
function t = hires_timer()
cps = 166e6
t = rdtsc() / cps
```

The parentheses after the function name are optional when defining the function, but not when calling it. Therefore the declaration of the last function could have been abbreviated to `function t = hires_timer`, but the call to `rdtsc` could not have been written as `t = rdtsc / cps`.

For further information about the omission of parenthesis when calling a function, see Section 5.1.8.

5.1.4. Named Parameters

The associations between the formal parameters of a function and its actual parameters may be positional or named. A positional parameter association is simply an actual parameter. All the positional parameter associations in a function call must precede all the named parameter associations. Thus, in the function call (see `myplot`'s definition in Example 5-2)

```
myplot(x, y, pointtype = 4, style = 'linespoints', linetype = 2)
```

the first two parameter associations (`x, y`) are positional, and the last three (`style, linetype, pointtype`) are named. Two things in the previous line of code are worth noting:

- When parameters are associated via their names the formal parameter's position is irrelevant.
- Positional parameter associations have nothing to do with optional parameters. A named parameter can be handled as an optional parameter as well as a positional parameter.

Calling a function with named parameters does not require any special code in the function. Function `myplot` is an simple user-defined function:

Example 5-2. Function accepting named arguments

```
function myplot(x, y, style, linetype, pointtype)

// checks for optional parameters would go here :)

select style
case 'lines' then
    plot2d(x, y, linetype)
case 'points' then
    plot2d(x, y, -pointtype)
case 'linespoints' then
    plot2d(x, y, -pointtype, '020')
```

```

    plot2d(x, y, linetype, '000')
end

```

To make the two parameters `linetype`, and `pointtype` optional parameters, we add a check for the existence of these parameters in the function's, i.e. the local scope. In Example 5-3 `myplot` gets extended in this direction.

Example 5-3. Function accepting optional arguments

```

function myplot(x, y, style, linetype, pointtype)

if ~exists('linetype', 'local')           – quotes around the parameter name are required
    linetype = 1
end
if ~exists('pointtype', 'local')         – 'local' excludes global variables from search
    pointtype = 1
end

select style
case 'lines' then
    plot2d(x, y, linetype)
case 'points' then
    plot2d(x, y, -pointtype)
case 'linespoints' then
    plot2d(x, y, -pointtype, '020')
    plot2d(x, y, linetype, '000')
end

```

Now `myplot` can be called in any of the following forms:

```

myplot(x, y, 'lines')                    – only positional parameters
myplot(x, y, style = 'linespoints')      – 3rd parameter is named
myplot(x, y, 'points', 2, 3)             – override defaults
myplot(x, y, linetype = 5, ..            – named params, one override
    style = 'linespoints')
myplot(x, y, pointtype = 4, ..
    style = 'linespoints', .
    linetype = 2)                        – named params where possible

```

5.1.5. Bulletproof Functions

If we want to write bulletproof Scilab functions, we have to take care that our functions get the right number of arguments which are furthermore of the right type, and correct dimension. This is

necessary because of Scilab's dynamic nature allowing us to pass arguments of different types, dimension, etc. to a single function.

We discuss the issues of writing robust function using Example 5-4 as an illustration. The complete function definition is given in Chapter 10.

Example 5-4. Function `cat`

```
function [res] = cat(macname)
// Print definition of function 'macname'
// if it has been loaded via a library.

[nl, nr] = argn(0);                                ❶
if nr ~= 1 then
    error('Call with: cat(macro_name)');
end

if type(macname) ~= 10 then                          ❷
    error('Expecting a string, got a ' ..
        + typeof(macname));
end
if size(macname, '*') ~= 1 then                      ❸
    sz = size(macname);
    error('Expecting a scalar, got a ' ..
        + sz(1) + 'x' + sz(2) + ' matrix')
end

[res, err] = evstr(macname);                          ❹
if err ~= 0 then
    select err
    case 4 then
        disp(macname + ' is undefined. ');
        return;
    case 25 then
        disp(macname + ' is a builtin function');
        return;
    else
        error('unexpected error', err);
    end // select err
end // err ~= 0

...
```

- ❶ First, we check how many actual parameters `cat` has received. The built-in `argn` returns the number of left-hand side – or output – variables `nl` (In this example we do not make use of `nl`.), and the number of right-hand side – or input – values `nr`.

Ensuring the correct number of *input* arguments always is the first step. Otherwise we cannot assume that even accessing a parameter is valid. The number of output values is not as critical,

for calling a function with less output variables than specified in the function's signature causes the extra output values to be silently discarded.

After learning the number of actual parameters, we immediately check whether it is in the right range. Our example simply terminates with an error if the number of arguments is incorrect.

- ② The next thing to address are the types of the arguments. Again we let the function fail with an error if it does not get what it wants, but this is not the only possible way of handling these kinds of errors.

It is conceivable that we convert from one type to another, say from numeric to string.

Furthermore, it is possible that the type of the arguments determines the algorithm chosen, a feature normally advertised under the name “function overloading” (see Section 4.2).

- ③ Finally, we examine the arguments' structure. A function can e.g. allow scalars only, or accept scalars and matrices. Here, we enforce a scalar. In other functions certain dimensional relations of several input parameters must be enforced. E.g. the matrix multiplication $A * B$ is only defined for `size(A, 'c') == size(B, 'r')`.

- ④ Now we can start with the real work.

At first glance all this checking gizmos might seem exaggerated. To do it justice we should keep in mind that it is only necessary if a function must work reliably in different environments. All functions that a library exports belong to that class, because the library writer does not know how the functions will be used in the future. Quick-and-dirty functions are a different thing, so are functions that are never called interactively.

5.1.6. Function Variables

Functions are a data type on their own right. Therefore, they themselves can be arguments to other functions, and they can be elements in lists.

```
->deff('y = fun(x)', 'if x > 0, y = sin(x); else, y = 1; end')
```

```
->fun(%pi / 2)
```

```
ans      =  
      1.
```

```
->fun(-3)
```

```
ans      =  
      - 1.
```

```
->bar = fun
```

```
bar      =  
[x]=bar(y)
```

– bar is a complete copy of fun

```
->typeof(bar)
```

```

ans      =
function

->deff('a = fun(u, v, w)', 'a = u^2 + v^2 + 2*u*v - w^2')
Warning :redefining function: fun

->bar(%pi / 4)^2
ans      =
      0.5

->fun(2, 3, 4)
ans      =
      9.

```

As the example shows, Scilab employs its usual copy-by-value semantics when assigning to function-variables, consistent with the assignment of any other data type.

5.1.7. Functions as Parameters in Function Calls

As mentioned above, user-defined functions can be passed as parameters to (usually different) functions. Builtin functions have to be wrapped in user-defined functions before they can be used as parameters.

The following example defines a functional that implements a property of Dirac's delta distribution.

```

->deff('y = delta(a, foo)', 'y = foo(a)')

->delta(cos)
      !-error      25
bad call to primitive :cos

->deff('y = mycos(x)', 'y = cos(x)')

->delta(0, mycos)
ans     =
      1.

```

The next example is a bit more convoluted, but also closer to the real world. We define a new optimizer function, called `minimize`, which is based on Scilab's `optim` function. `minimize` expects two vectors of data points: `xdata` and `ydata`, a vector of initial parameters `p_ini`, the function to be minimized `func`, and an objective functional `obj`.

The advantage of defining separate model and objective functions is an increased flexibility as both can be replaced at will without changing the core minimization function, `minimize`.

```

function [f, p_opt, g_opt] = minimize(xdata, ydata, ..
                                     p_ini, func, obj)

```

```
// on-the-fly definition of the objective function
deff('[f, g, ind] = _cost(p_vec, ind)', ..
    '[f_val, f_grad] = func(xdata, p_vec); ..
    [f, g] = obj(f_val - ydata, f_grad)');

[f, p_opt, g_opt] = optim(_cost, p_ini);
```

`minimize` needs the model function `func` that returns the value and the gradient at all points `x` for a given vector of parameters `p_vec`. Moreover, we need the objective functional `obj` that gives the “cost”, as well as the direction of steepest descent in parameter space.

In the example we choose a quadratic polynomial for the model, `my_model`, and least squares for the objective `lsq`.

```
function [f, g] = my_model(x, p)
g = [ones(x), x, x.*x];
f = p(1) + x.*(p(2) + x*p(3));

function [f, g] = lsq(diff, grad)
f = 0.5 * norm(diff)^2;
g = grad' * diff;
```

Given these definitions, we can call `minimize`:

```
dx = [0.0 1.0 2.0 2.5 3.0]';
dy = [0.0 0.9 4.1 6.1 9.5]';
p_ini = [0.1 -0.2 0.9]';

[f_fin, p_fin, p_fingrad] = ..
    minimize(dx, dy, p_ini, my_model, lsq)

xbasc(); // clear window
plot2d(dx, dy, -1); // plot data points ...
xv = linspace(dx(1), dx($), 50)';
yv = my_model(xv, p_fin);
plot2d(xv, yv, 1, '000'); // ... and optimized model function
```

5.1.8. Omitting Parentheses on Function Call

by Glen Fulford

The parentheses of any one-parameter function can be omitted, if the function accepts a string argument. Moreover, the quotes for a literal string argument can be left out, too.

This is especially useful, when working interactively, and loading functions, or scripts. There is no need to type until your fingers bleed by saying

```
->getf('foo.sci')
```

as the next two examples work just as well.

```
->getf 'foo.sci'
```

and even

```
->getf foo.sci
```

is OK. Note that this is not only true for built-in, but also for user-defined functions.

Function `exec` is an exception to the rule that a semicolon suppresses any output of the preceeding clause, if it is invoked without parenthesis. In fact, `exec` does echo the commands it executes if used without parenthesis *despite* a trailing semicolon, this is

```
->exec script.sci;
```

with semicolon gives same results as

```
->exec('script.sci')
```

without semicolon, whereas

```
->exec('script.sci');
```

does not echo the commands of the script file.

5.1.9. Functions in `tlists` and `mlists`

Currently the only composite data structures that allows for storage of functions are the typed list, `tlist`, and the matrix-like list, `mlist`.

Given the typed-list `t = tlist(['funlist_t', 'x0', 'x1', 'fun'], -0.5, 0.5, f)`, where `f` is e.g. defined as `def('y = f(x)', 'y = 2.0*x + 1.0')`, the non-function components are accessed as usual, i.e.,

```
->t("x1")
ans =
    0.5
```

```
->t.x0
ans =
    - 0.5
```

See also Section 4.6.4.

However, function components cannot be called directly, e.g. `t("fun")(0)` or `t.fun(0)`. Instead, we go on a little detour, either by calling `feval` or by using a dummy variable.

```
->feval(0, t("fun"))
```

```

ans =
    1.

->feval(0, t.fun)
ans =
    1.

->_f = t("fun"); _f(0), clear _f
ans =
    1.

->_f = t.fun; _f(0), clear _f
ans =
    1.

```

Both workarounds go well with argument vectors to the function. Assigning to a dummy variable is faster than using `feval`.

5.1.10. `macrovar`

The `macrovar` function could be called the functional cousin of the `size` function. The primary purpose of `macrovar` is to support the Scilab-to-Fortran translator, but it can be useful for other purposes, too.

`macrovar` reveals five important attributes of a user function. These are the names of all

- input variables,
- output variables,
- global variables,
- functions called, and
- local variables.

One example of an interesting use of `macrovar` is an integration routine that accepts integrand functions with an arbitrary number of arguments, i.e. over arbitrary many dimensions.

```

function vol = int_cube(ifun)
// integrate ifun in an appropriate hypercube
// (0, ..., 0), ..., (1, ..., 1)

ifun_var = macrovar(ifun)
ifun_sz  = size(ifun_var(1)) // names of input arguments
ifun_dim = ifun_sz(1)

for d = 1:ifun_dim
    // integrate in one dimension
end

```

5.2. Libraries of `sci`-functions

Most users think there is no difference between loading a function immediately via `getf`, or loading it on-demand via `lib`. However, there are cases when `getf` and `lib` produce different results¹. To stay clear of trouble it is useful to know what exactly `getf` and `lib` do.

5.2.1. `getf` vs. `lib`

getf. `getf("filename")` immediately, i.e. when `getf` is executed, loads *all* functions in *filename*. It is like saying: “Your functions – give them to me!” After a successful `getf` all functions from *filename* show up in the workspace. (Try **who** before and afterwards.)

`getf` is most useful during the development process, when functions are changed often. It also works well during production runs, if the number of functions loaded from the file is not too high. To suppress repeated loading of the same function-file, the following construct can be used:

```
if ~exists("myfunction") then
    getf("myfile.sci")
end
```

where `myfunction` is one of the functions in `myfile.sci`. Do not forget the quotes around the function’s name in the call to `exists`!

lib. `libvar = lib("lib-directory")`² on the other hand does not load any function when the `lib` is executed. Instead, it marks all function-names listed in the file *lib-directory*/names as available for later loading.

Note: Note that *lib-directory* must end with a directory separator, i.e. a “/” in UNI*.

A function from *libvar* will be loaded when its – at that time undefined – name is first encountered during execution. It will never be re-loaded afterwards, even if the defining *bin*-file or the library change. The crucial word in the next-to-last sentence is “undefined”: If the library function’s name coincides with the name of a built-in function or an already defined user-function, the function definition from the library will *not* get loaded!

-
1. Thanks for pointing out the problems of Scilab’s library handling in general and `lib/genlib` in particular go to Alexander Vigoder.
 2. The online documentation, **help lib**, somewhat misleadingly calls *lib-directory* a *lib_path*, though it is only a single directory, not a path.

The function-names in *lib-directory*/names must refer to compiled functions, i.e. bin-files, in *lib-directory*. How to generate bin-files? Scilab offers three ways to convert a human readable sci-file into compiled bin-format.

Caution

The bin-format might change from one version to the next. When switching Scilab versions, it is advisable to delete all bin-files and regenerate them with the newer Scilab.

- Function save copies an arbitrary Scilab variable or user-defined function to a file while compiling it to binary format. Given the function `foo`, we can generate its bin-file interactively with

```
->foo
    foo =
    [y]=foo(x)

->save("foo.bin", foo)

->unix_w("ls -l foo.bin")
-rw-rw-r-  1 lvandijk users      204 Nov 14 09:30 foo.bin
```

`foo` must be accessible when `save` is executed, and the function's name in the call to `save` is not quoted.

- Scilab supports the (undocumented) `-comp` command-line parameter to compile a sci-file into a bin-file.

```
lvandijk@hydra:~/hsc/scilab/src/minilib $ cat foo.sci
function y = foo(x)
y = 1 + x

lvandijk@hydra:~/hsc/scilab/src/minilib $ scilab -comp foo.sci
generating foo.bin

lvandijk@hydra:~/hsc/scilab/src/minilib $ ls -l foo.*
-rw-rw-r-  1 lvandijk users      204 Nov 14 09:37 foo.bin
-rw-r-r-  1 lvandijk users      30 Nov  8 11:45 foo.sci
```

This way goes very well with Makefiles, as it implies the simple rule

```
# -*- makefile -*-

%.bin: %.sci
    scilab -comp $<
```

- The `genlib` function, which is described further down in Section 5.2.2.

The names of the files which are part of the library are collected in `names` in a very simple format: one function-name per line, e.g.:

```
lvandijk@hydra:~/hsc/scilab/src/minilib $ cat names
bar
baz
foo
multi
myfun
```

One function per file workaround: If a `sci`-file is intended to hold more than one function which all should be equally accessible from within Scilab, the following workaround can be used, given the operating system supports (symbolic) links.

Let us assume the multi-function `sci`-file is `manyfun.sci`. For every function `fun1`, `fun2`, ..., create a (symbolic) link to the “main” `bin`-file, `manyfun.bin`, like

```
ln -s manyfun.bin fun1.bin
ln -s manyfun.bin fun2.bin
...
```

and generate `names` afterwards.

The advantage of this hack is that it makes all functions from `manyfun.bin` available. Its disadvantage is that it is hard to maintain, e.g. if functions are added to or deleted from `manyfun.sci` and `manyfun.bin` has to be re-compiled, it is possible that some new links must be set up, and old links must be deleted.

There are plenty of possibilities to generate `names` outside Scilab, like e.g.

```
# -*- makefile -*-

sci_src:=$(wildcard *.sci)
sci_bin:=$(subst .sci,.bin,$(sci_src))

names: $(sci_bin)
    rm -f names
    for n in $(sci_bin); do echo $$ (basename $$n .bin) » names; done
```

or alternatively, if the shell has process substitution,

```
# -*- makefile -*-
```

```
names: $(sci_bin)
      sed -e 's/\.bin$//' <(ls -l *.bin) > names
```

However, these solutions are unsatisfactory for large numbers of filenames. The reason for this shortcoming is the limited command-line length in most shells. In the first example `make(1)` expands `$(sci_bin)` to the names of all bin-files, in the second the shell does. Both might overrun the shell's command-line length limit. Therefore, a reliable solution does avoid expanding the filenames at the command-line. The following Perl-script, Example 5-5, demonstrates a more robust solution.

Example 5-5. Generate names for lib: `gen-names`

```
#!/usr/bin/perl -w
# name:      gen-names  -  generate "names" file for the use with
#                               Scilab's built-in lib function
# author:    L. van Dijk
# last rev.: Tue Nov 14 09:10:31 UTC 2000
# Perl ver.: 5.005_03

use strict;
use IO::File;

unless (@ARGV) { $ARGV[0] = '.' }
foreach my $dir_name (@ARGV) { process_directory($dir_name) }

sub process_directory {
    my $dir_name = shift;

    my $names = IO::File->new("> $dir_name/names")
        or die "Cannot open \'$dir_name/names\': $!\n";
    opendir DIR, $dir_name or die "Cannot open \'$dir_name\': $!\n";
    while (defined($_ = readdir DIR)) {
        next unless s/\.bin$//;
        print $names "$_\n";
    }
    closedir DIR;
}
```

gen-names is either called without an argument, then it creates names from the bin-filenames in the current working directory. If the arguments to **gen-names** are directories, they are processed in turn, each directory getting their respective names file.

After all desired sci-files have been converted to bin-files and the matching names-file has been written, the library is activated from the Scilab prompt:

```
->minilib = lib("/home/lvandijk/hsc/scilab/src/minilib/")
minilib =

Functions files location :/home/lvandijk/hsc/scilab/src/minilib/
```

```
baz      bar      foo      myfun      multi
```

libvar, in our example *minilib* has got a special type, *library*. As you can see, *minilib* holds the information about the library, namely its defining directory and all functions it exports. To lookup the functions in a library, simply type the library variable's name. For a reverse lookup, i.e. searching to which library a function belongs, Scilab has the *whereis* function.

```
->whereis myfun
ans  =
minilib
```

The non-mandatory naming convention for library variables suggests to append *lib* to a library variable name, e.g. *percentlib*, *fracclablib*, *soundlib*, *xdesslib*.

libvar contains all necessary information about the library, and it is just an ordinary variable. Thus, it is lost when the Scilab session is closed. To make a library definition persistent, we have to perform two further steps:

1. Translate *libvar* into a re-loadable format. Our old friend, the *save* function does that job for us.

```
->save("/home/lvandijk/hsc/scilab/src/minilib/lib", miniblib)
```

2. Reload the library definition on every start of Scilab by placing the line

```
load("/home/lvandijk/hsc/scilab/src/minilib/lib");
```

in the run-code file *~/ .scilab*. See also the primary Scilab run-code file, *SCI/scilab.star*.

The – again non-mandatory – file naming convention for saved library variables is to call them *lib*.

5.2.2. genlib

genlib reduces the process of compiling all necessary *sci*-files, generating names, and finally saving the library variable to one step:

```
genlib("library-variable", lib-directory)
```

where the library variable names must be passed as a string. *genlib* always saves *library-variable* in *lib-directory/lib*.

5.2.3. Library Caveats

5.2.3.1. Library Files and Library Functions

Important: Scilab's library mechanism only works well if

- every `sci`-file in the library contains *only one* function, and
- the `sci`-file name without extension is identical to the function name in the file.

In other words: function `foo` must live in file `foo.sci` as a hermit.

Multiple functions per file are allowed; Scilab will not even generate a warning if a file with more than one function is used in a library. But the user should restrict the use of this feature to helper functions. A helper function is a function that only assists the main (not in the C-meaning) function, the one which gives the function-file the name. Special attention should be paid to the names of these “hobo” functions, which ride in the name of a real library function. They can cause name clashes with other functions. To avoid these underscores, dollar signs, or sharp symbols should be prepended to the function name, thereby faking a separate name space. This is demonstrated in the function file `my_gamma.sci`:

```
// file: my_gamma.sci

function a = my_gamma(z)
    if abs(z - int(z)) <= %eps and z > 0.0 then
        a = $_faculty(int(z) - 1)
    else
        a = gamma(z)
    end
endfunction

function k = $_faculty(n)                - prepend "$_"
    k = prod(1 : n)
endfunction

// end file my_gamma.sci
```

5.2.3.2. On-Demand Loading

The on-demand loading of symbols from libraries can cause confusion (on the user's side) when a library symbol name clashes with the name of a “normal” variable.

Assume the library in the current directory holds the single function `foo`, which is defined as follows.

```
function y = foo(x)
```

```

    y = x + 0.5
endfunction

```

Consider a session that activates the library and defines a variable with the same name as the function from the library.

```

->foolib = lib("./")
foolib =
Functions files location :./
foo

->foo
foo =
[y]=foo(x)

->foo(2)
ans =
    2.5

->foo = 100
Warning :redefining function: foo
foo =
    100.

->foo
foo =
    100.

->foo(2)
ans =
    2.5

```

Function `foo` in library `foolib` and variable `foo` peacefully coexist. If variable `foo` is defined before library `foolib` gets activated the same behavior results, only the warning message does not appear as library functions load silently.

Clearing `foo` removes the variable. If variable `foo` does not exist, clearing `foo` removes function `foo`, but the next time symbol `foo` is referred to again, function `foo` in library `foolib` will be loaded. To permanently clear function `foo` from the workspace, the association with library `foolib` must be removed first: `clear foolib; clear foo`. Now, function `foo` is undefined *and* its definition will not be reloaded from library `foolib`.

5.2.4. Loading Non-Functions With `lib`

The `lib` function is not picky in what it loads into the workspace. In the previous sections `lib` has been applied to directories that contain only *files of compiled functions*, this is “.bin” files.

However, there is no restriction at all to save other entities than functions, for example variables, to “.bin” files. In case `lib` finds a saved variable in a directory, it will load it into the workspace as a *local* variable.

```
// my_sinc(x) uses the external parameter my_sinc_n
function y = my_sinc(x), ..
    if x == 0 do ..
        y = my_sinc_n, ..
    else ..
        y = sin(my_sinc_n * x) / (my_sinc_n * x), ..
    end, ..
endfunction

save("my_sinc.bin", my_sinc)
clear my_sinc

my_sinc_n = 100;
save("my_sinc_n.bin", my_sinc_n)
clear my_sinc_n
```

Now assume that names in directory `/tmp` contains

```
my_sinc
my_sinc_n
```

Note: `genlib` only considers “.sci” files. To get a variable into a library, the variable has either to be defined in a separate “.sci” file, or it manually must be `save`d and its name added to `names`.

Then both, `my_sinc` and `my_sinc_n` are loaded with the following `lib` call and can be used in the usual way:

```
->sinc_lib = lib("/tmp/")
sinc_lib =
Functions files location :/tmp/
my_sinc_n      my_sinc

->my_sinc_n
my_sinc_n =
    100.

->my_sinc(1.0)
ans =
    - 0.0050637

->my_sinc_n = 1
Warning :redefining function: my_sinc_n
my_sinc_n =
```

```
1.  
  
->my_sinc(1.0)  
ans =  
0.8414710
```

The above excerpt of a session transcript shows that the value of the variable `my_sinc_n`, which has been loaded with `lib` is used in the call of function `my_sinc`.

Chapter 6. Performance

*Scilab—The fastest thing from France
since Django Reinhardt.*

Ch. L. Spiel

In this chapter we discuss how expressions can be written to execute more quickly while doing the same thing. Scilab is powerful and flexible, therefore there are plenty of things one can do to speed up function execution. On the downside there are a lot of things that can be done the wrong way, slowing down the execution to a crawl.

In the first part of this chapter, Section 6.1, we focus on high-level operations that are inherently executed fast. The main class to name here are vectorized operations. Another class are all functions that are constructing or manipulating vectors or matrices as a whole. The second part of this chapter, Section 6.2, deals with the extension of Scilab through compiled functions for the sake of increased execution speed. We close with a section on how to compile Scilab itself to increase its performance with Section 6.3.

6.1. High-Level Operations

Not using vectorized operations in Scilab is the main source for suffering from a slow code. Here we present performance comparisons between different Scilab constructs that are semantically equivalent.

6.1.1. Vectorized Operations

The key to achieve a high speed with Scilab is to avoid the interpreter and instead make use of the built in vectorized operations. Let us explain that with a simple example.

Say we want to calculate the standard scalar product s of two vectors a and b which have the same length n . Naive as we are, we start with

```
s = 0 // line 1
i = 1 // line 2
while i <= n // line 3
    s = s + a(i) * b(i) // line 4
    i = i + 1 // line 5
end // line 6
```

Here Scilab re-interprets lines 3 to 5 in every round-trip, which in total is n times. This results in slow execution. The example utilizes no vectorization at all. On the other hand it uses only very little memory as no vectors have to be stored.

The first step to get some vectorization is to replace the `while` with a `for` loop.

```

s = 0                                // line 1
for i = 1:n                          // line 2
    s = s + a(i) * b(i)              // line 3
end                                  // line 4

```

Line 2 is only interpreted once; the vector `i = 1:n` is set up and the loop body, line 3 is threaded over it. So, only line 3 is re-evaluated in each round trip.

OK, it is time for a really fast vector operation. In the previous examples the expression in the loop body has not been modified, but we can replace it with the element wise multiplication operator `.*`, and replace the loop with the built-in `sum` function. (See also Section 6.1.3.3.)

```
s = sum(a .* b)
```

One obvious advantage is, we have a one-liner now. Is that as good as it can get? No, the standard scalar product is not only a built-in function it is also an operator:

```
s = a * b'
```

We summarize the timing results of a PII/330 GNU/Linux-system in Table 6-1.

Table 6-1. Comparison of various vectorization levels

construct	MFLOPS
while	0.005
for	0.008
.* and sum	1.7
*	2.8

In other words the speed ratio is 1:1.6:330:550. Of course the numbers vary from system to system, but the general trend is clear. The figures tell us two things:

1. Keeping the problem size the same, a vectorized operation is over a hundred times faster than the comparable interpreter (emulated) operation.
2. In the same time Scilab executes several hundreds or thousands of vectorized operations, it can only run a single interpreted operation.

```

->n=1000; timer(); for i=1:n, sqrt((i-1)*%pi/n); end; timer()
ans =
    0.05

->n=100000; timer(); sqrt((1:n)*%pi/n); timer()
ans =
    0.04

```

The latter point is a valuable starting point for many vectorizations. This holds particularly for partial vectorizations, where the operations under consideration cannot be replaced by a single operator or function call. If a slow interpreted command cannot be replaced by a vectorized operator – which would result in a speed-up of a factor of 500 say, parts of the command might be amenable to vectorization. This partial vectorization can replace parts of the expression with vectorized operations. The important rule is that several hundred up to thousands of vectorized operations can be traded in for the interpreted operation to be replaced.

In the next example the matrix `a` is treated as a collection of row-vectors. The problem is to subtract row-vector `b` from the rows in `a`. Obviously, this can be achieved with a loop. The faster way is to cast `b` into a matrix of the same shape as `a` and then subtract the two matrices. What seems to be a detour – duplication the entries of `b` – turns out to be advantageous for performance.

```
a = [2.56, 2.85, 2.66; ..
      3.74, 3.25, 3.21; ..
      4.05, 4.89, 4.49; ..
      5.90, 5.94, 5.37];
b = [1.01, 1.67, 1.79];
[m, n] = size(a);

// non-vectorized
c0 = zeros(a);
for i = 1:m
    c0(i, :) = a(i, :) - b;
end
c0

// partial vectorization
c1 = a - b(ones(m, 1), :)
```

```
->m = 1000; n = 200; a = rand(m, n); b = rand(1, n);

->timer(); c0 = zeros(a); for i = 1:m, c0(i, :) = a(i, :) -
b; end; timer()
ans =
    0.19

->timer(); c1 = a - b(ones(m, 1), :); timer()
ans =
    0.07
```

6.1.2. Avoiding Indexing and Resizing

Accessing a single vector-element or matrix-element in a (often even nested) loop is slow. Sometimes the loop/index construct cannot be avoided, but in many cases it can be replaced with an equivalent vectorizable expression. Moreover, if you cannot get around indexing single elements, at

least avoid resizing (most often: growing) the vector or matrix. Compare the following three examples.

```
// (1) insert element at non-existent position => autovivicate element
v = []
for i = 1:n
    v(i) = i
end

// (2) insert into pre-sized vector
v = zeros(1, n)
for i = 1:n
    v(i) = i
end

and

// (3) append to existing vector
v = []
for i = 1:n
    v = [v, i]
end
```

Snippet (2) is the fastest of the three. It should be used whenever the final size is known in advance, or if the final size can be calculated in an easy way. Appending to an existing vector or matrix (3) is almost twice as fast as forcing a new element to spring into existence by indexing (1). In the authors' opinion, snippet (3) is the clearer solution in comparison to (1) for all problems where the final vector size cannot be determined in advance.

But again for our specific example a built-in operator exists that does the same job at lightning speed: the range operator, colon “:”, which is described in detail in Section 6.1.3.1.

```
// (4) range generator (colon operator)
v = 1:n
```

The speed ratio of examples (1), (2), (3) and (4) is approximately 1:20:2:4000.

In the next example, Example 6-1, the functions actually try to do something useful: they mirror a matrix along its columns or rows. We show different implementations of `mirrorN` that all do the same job, but utilize more and more of Scilab's vector power with increasing function index N .

Example 6-1. Variants of a matrix mirror function

```
function b = mirror1(a, dir)
// mirror matrix a along its
// rows, dir = 'r' (horizontal)
// or along its columns, dir = 'c' (vertical)

[rows, cols] = size(a)
select dir
```

```

case 'r' then
    for j = 1 : cols
        for i = 1 : rows
            b(i, j) = a(rows - i + 1, j)
        end
    end
case 'c' then
    for j = 1 : cols
        for i = 1 : rows
            b(i, j) = a(i, cols - j + 1)
        end
    end
else
    error("dir must be \"r\" or \"c\"")
end

```

```

function b = mirror2(a, dir)
// same as mirror 1

[rows, cols] = size(a)
b = []
select dir
case 'r' then
    for i = rows : -1 : 1
        b = [b; a(i, :)]
    end
case 'c' then
    for i = cols : -1 : 1
        b = [b, a(:, i)]
    end
else
    error("dir must be \"r\" or \"c\"")
end

```

```

function b = mirror3(a, dir)
// same as mirror 1

[rows, cols] = size(a)
select dir
case 'r' then
    i = rows : -1 : 1
    b = a(i, :)
case 'c' then
    i = cols : -1 : 1
    b = a(:, i)
else
    error("dir must be \"r\" or \"c\"")
end

```

```

end

function b = mirror4(a, dir)
// same as mirror 1

select dir
case 'r' then
    b = a($:-1:1, :)
case 'c' then
    b = a(:, $:-1:1)
else
    error("dir must be \"r\" or \"c\"");
end

```

Besides the performance issue discussed here the functions in Example 6-1 demonstrate how much expressiveness Scilab has got. The solutions look quite different, though they give the same results. The benchmark results of all functions are plotted in Figure 6-1, and an extensive discussion is found in Section 6.2.1. In brief the functions get faster from top to bottom, function `mirror1` is the slowest, `mirror4` the fastest.

6.1.2.1. \$-Constant

The last of the examples, `mirror4`, introduces a new symbol, the “highest index”, `$` along a given direction. The dollar sign is *only* defined in the index expression of a matrix. As 1 always is the lowest (or first) index, `$` always is the highest (or last). The dollar represents a constant, but this constant varies across the expression! More precisely it varies with each matrix dimension. Let us make things clear by giving an example.

```

->m = [ 11 12 13; 21 22 23 ];

->m(2, $)
ans =
    23.

->m($, $)
ans =
    23.

->m(:, $/2 + 1)
ans =
!   12. !
!   22. !

```

6.1.2.2. Reshaping

Reshaping a matrix in Scilab is a cheap operation. A 1000-times-1000 matrix is reshaped into a 2000-times-500, or a 250-times-4000 matrix at very little computational cost. However, keep in mind that the time to reshape is proportional to the total size of the matrix, i.e., reshaping an n -times- m matrix is an $O(n*m)$ operation.

When to use reshaping? If an algorithm that requires multiple indices into a matrix can be mapped onto an equivalent one that accesses a vector, or vice versa, it can be a benefit to work with the more convenient representation and reshape afterwards.

Our example to illustrate this is simple, but gives you the gist of reshaping. Sorting into lexicographical order is most easily done with a vector. (`gsort` can sort a *matrix* into lexicographical order, see Section 6.1.3.3.6, but we want to demonstrate reshaping and not the functionality of `gsort`) To get a matrix where strings of same first letters are in the same rows, we use `matrix`.

```
->perm3 = ['cab', 'bca', 'acb', 'bac', 'cba', 'abc'];
->sorted_perm3 = gsort(perm3, 'c', 'i');
->matrix(sorted_perm3, 2, 3)'
ans =
!abc  acb  !
!      !
!bac  bca  !
!      !
!cab  cba  !
```

See also Section 6.1.3.3.8 about `matrix`, and the following section, Section 6.1.2.3 about the flattened matrix representation.

6.1.2.3. Flattened Matrix Representation

The `$` sign leads us to the flattened or vector-like representation of a matrix, if we rewrite the third line of the above example to

```
->m(1:$)'1
ans =
! 11. !
! 21. !
! 12. !
! 22. !
! 13. !
! 23. !
```

1. Remember
that the colon operator returns a row-vector.

The expression $u = v(:)$ is reshape operation, assigning to u the column-representation of v . For general reshaping of matrices, see the `matrix` function in Section 6.1.3.3.8.

Tip: Given the vector v , the expression $v = v(:)$ is a very convenient idiom in a function to force v into column (i.e. 1-times- N) form.

In general a $n \times m$ matrix mat can be accessed in three ways:

- as a unit by saying mat ,
- by referencing its elements according to their row and column with $mat(i, j)$, or
- via indexing into the flattened form $mat(i)$.

The following equivalence holds:

$$mat_{i,j} = mat_{i+(j-1)n}.$$

Scilab follows Fortran in its way to store matrices in column-major form. See also the discussion of the function `matrix` in Section 6.1.3.3.

6.1.3. Built-In Vector-/Matrix-Functions

Scilab provides many built-in functions that work on vectors or matrices. Knowing what functions are available is important to avoid coding the same functionality with slow iterative expressions.

For further information about contemporary techniques of processing matrices with computers, the classical work “Matrix Computations” [Golub:1996] is recommended.

6.1.3.1. Vector Generation

There are two built-in functions and one operator to generate a row-vector of numbers.

6.1.3.1.1. Operator “:”

This syntax of the colon operator is

```
initial [: increment] : final
```

with a default *increment* of +1. To produce the equivalent piece of Scilab code, we write

```
x = initial
v = [ x ]
while x <= final - increment
```

```

    x = x + increment
    v = [v, x]
end

```

where `v` is the result. Note that the last element of the result always will be smaller or equal to the value *final*.

See also Section 2.6 for a discussion of the dangers involved in using a colon-expression with fractional parameters.

6.1.3.1.2. `linspace`

The syntax of `linspace` is

```
linspace(initial, final [, length])
```

using a default of 100 for *length*. `linspace` returns a row-vector with *length* entries, which divide the interval (*initial*, *final*) in equal-length sub-intervals. Both endpoints, i.e. *initial* and *final* are always included.

6.1.3.1.3. `logspace`

`logspace` works much like `linspace`, and the following relation holds

$$\text{logspace}(\text{init}, \text{final}) = 10^{\text{linspace}(\text{init}, \text{final})}$$

6.1.3.2. Whole Matrix Construction

All of the functions shown in this section are capable to produce arbitrary matrices including the boundary cases of row-, and column-vectors.

6.1.3.2.1. `zeros`

As the name suggests this function produces a matrix filled with zeros. The two possible instantiations are with two scalar arguments

```

n = 2
m = 5
mat = zeros(n, m)

```

or with one matrix argument

```

mat1 = [ 4 2; ..
        4 5; ..

```

```

      3 5 ]
mat2 = zeros(mat1)

```

The first form produces the n times m matrix `mat` made up of zeros, whereas the second builds the matrix `mat2` which has the same shape as `mat1`, and is also consisting solely of zeros.

Single scalar argument to `zeros`

In the case of a single scalar argument `zeros` returns a 1-times-1 matrix, the sole element being a zero.

Furthermore, note that

```
zeros()
```

is not allowed.

To generate an empty string matrix use `emptystr`, Section 6.1.3.2.6.

6.1.3.2.2. `ones`

The command is functionally equivalent to `zeros`. Instead of returning a matrix filled with `0.0` as `zeros` does, `ones` returns a matrix filled with `1.0`. The only difference is a third form which is permitted for `ones`, and that is calling the function without any arguments:

```

->ones()
ans =
    1.

```

6.1.3.2.3. `eye`

The `eye` function produces a generalized identity matrix, this is a matrix with all elements

$$a_{i,j} = 0 \quad \text{for } i \neq j, \quad \text{and}$$

$$a_{i,j} = 1 \quad \text{for } i = j.$$

This command is functionally equivalent to `zeros`. The only extension is the usage without any argument, where the result automatically takes over the dimensions of the matrix in the subexpression it is used.

```

->a=[2 3 4 3; 4 2 6 7; 8 2 7 4]
a =
!   2.    3.    4.    3. !
!   4.    2.    6.    7. !
!   8.    2.    7.    4. !

```

```
->a = 2*eye()
ans =
! 0.    3.    4.    3. !
! 4.    0.    6.    7. !
! 8.    2.    5.    4. !
```

6.1.3.2.4. diag

Function `diag` has two different working modes depending on the shape of its argument. Given a vector v it constructs a diagonal matrix mat from the vector, with v being mat 's main diagonal, i.e. $mat(i, i) = v(i)$ for all $v(i)$. Given an arbitrary matrix mat , `diag` extracts the diagonal as a column-vector.

```
->diag(2:2:8)
ans =
! 2.    0.    0.    0. !
! 0.    4.    0.    0. !
! 0.    0.    6.    0. !
! 0.    0.    0.    8. !

->m = [2, 3, 8; 7, 6, -6; 0, -5, -8]
m =
! 2.    3.    8. !
! 7.    6.   -6. !
! 0.   -5.   -8. !

->diag(m)
ans =
! 2. !
! 6. !
! -8. !
```

The 2-argument form of the `diag` function

```
diag(v, k)
```

constructs a matrix that has its diagonal k positions away from the main diagonal, the diagonal being made up from v again. Therefore, `diag(v)` is the special case of `diag(v, 0)`. A positive k denotes diagonals above, a negative k diagonals below the main diagonal. As for the 1-argument form, extraction of the k th super-diagonal (positive k , or subdiagonal (negative k) is also implemented.

```
->diag([1 1 1 1]) + diag([2 2 2], 1) + diag([-2 -2 -2], -1)
ans =
! 1.    2.    0.    0. !
! -2.    1.    2.    0. !
! 0.   -2.    1.    2. !
! 0.    0.   -2.    1. !
```

```
->diag(m, -1) // using the same m as above
ans =
!    7. !
! - 5. !
```

Tip: Nesting two calls to `diag` is the building block for an interesting idiom to test whether a matrix `m` is a diagonal matrix.

```
and( abs(diag(diag(m)) - m) <= %eps * abs(m) )
```

The inner call to `diag` extracts `m`'s main diagonal, the outer call taking this column-vector and construction a matrix out of it. The rest of the code simply checks the relative error.

6.1.3.2.5. `rand`

The `rand` function generates pseudo-random scalars and matrices. Again the function shares its two fundamental forms with `zeros`. Moreover, the distribution of the numbers can be chosen from 'uniform' which is the default, and 'normal'. The generator's seed is set and queried with

```
rand('seed', new_seed)

and

current_seed = rand('seed')
```

6.1.3.2.6. `emptystr`

`emptystr()` returns an empty string, `emptystr(m, n)` returns an m -times- n matrix of empty strings, and finally, `emptystr(a)` returns an empty matrix of strings which has the same size as `a`.

Example 6-2. Function `deblank`

```
function tm = deblank(sm)
// Remove leading or trailing blanks from all strings
// in string matrix SM.

tm = emptystr(sm)

for i = 1 : size(sm, "r")
    s = sm(i)

    istart = 1
    while istart <= length(s)
```

```

        if part(s, istart) == " "
            istart = istart + 1
        else
            break
        end
    end
end

istop = length(s)
while istop >= 1
    if part(s, istop) == " "
        istop = istop - 1
    else
        break
    end
end

tm(i) = part(s, istart:istop)
end // for i
endfunction

```

Note that Scilab has a built-in function called `stripblanks` which does exactly the same job than `deblank` does.

To generate an empty matrix of numbers use `zeros`, Section 6.1.3.2.1.

6.1.3.3. Functions Operating on a Matrix as a Whole

Although the section title might imply that the following functions apply to matrices only, Scilab's understanding allows for vectors anywhere a matrix is accepted (but not vice versa).

6.1.3.3.1. `find`

In our opinion one of the most useful functions in the group of whole matrix functions is `find`. It takes a boolean expression of matrices, i.e. an expression which evaluates to a boolean matrix, as argument and in form

```
index = find(expr)
```

returns the indices of the array elements that evaluate to true, i.e. %t in a vector. See also Section 6.1.2.3.

In the form

```
[rowidx, colidx] = find(expr)
```

it returns the row- and column-index vectors separately. Here is a complete example:

```
->a = [ 1 -4 3; 6 2 10 ]
a =
! 1. - 4. 3. !
! 6. 2. 10. !

->index = find( a < 5 )
index =
! 1. 3. 4. 5. !

->a(index)
ans =
! 1. !
! - 4. !
! 2. !
! 3. !

->[rowidx, colidx] = find( a < 5 )
colidx =
! 1. 2. 2. 3. !
rowidx =
! 1. 1. 2. 1. !
```

The expressions *expr* can be arbitrarily complex. They are not at all limited to a single matrix.

```
->b = [1 2 3; 4 5 6]
b =
! 1. 2. 3. !
! 4. 5. 6. !

->a < 5
ans =
! T T T !
! F T F !

->abs(b) >= 4
ans =
! F F F !
! T T T !

->a < 5 & abs(b) >= 4
ans =
! F F F !
! F T F !

->find( a < 5 & abs(b) >= 4 )
ans =
4.
```

Last but not least `find` is perfectly OK on the left-hand side of an assignment. So, replacing all odd elements in `a` with 0 simply is

```
->a( find(modulo(a, 2) == 1) ) = 0
a =
!  0.  - 4.   0.  !
!  6.   2.  10.  !
```

To get the number of elements that match a criterion, just apply `size(idxvec, '*')` to the index vector `idxvec` of the `find` operation.

6.1.3.3.2. max, min

Searching the smallest or the largest entry in a matrix are so common that Scilab has separate functions for these tasks. We discuss `max` only as `min` behaves similarly.

To get the largest value saying

```
max_val = max(a)
```

is enough. The alternate form

```
->[max_val, index] = max(a)
index =
!  2.   3.  !
max_val =
  10.
```

returns the position of the maximum element, too. The form of the index vector is the same as for `size`, i.e. `[row-index, column-index]`. Speaking of `size`, `max` has the forms `max(mat, 'r')`, and `max(mat, 'c')`, too.

```
->[max_val, rowidx] = max(b, 'r')
rowidx =
!  2.   2.   2.  !
max_val =
!  4.   5.   6.  !

->[max_val, colidx] = max(b, 'c')
colidx =
!  3.  !
!  3.  !
max_val =
!  3.  !
!  6.  !
```

These forms return the maximum values of each row or column along with the respective indices of the elements' rows or columns.

The third way of using `max` is with more than one matrix or scalar as arguments. All the matrices must be compatible, scalars are expanded to full matrix size, like `scalmat = scal * ones(mat)`. The return matrix holds the largest elements from all argument matrices.

```
->max(a, b, 3)
ans =
!  3.    3.    3.  !
!  6.    5.   10. !
```

6.1.3.3.3. `and`, `or`

Both, `and` and `or` borrow their syntax from the `size` function: without a second argument, or a star, “*”, as second argument the function is applied to the argument as a whole. A 1 or a “r” applies the function separately to each row, yielding a row-vector as result. Accordingly a 2 or a “c” applies the function separately to each column, yielding a column-vector as result.

The function `and` returns true if all components of the argument are true. Therefore, it is related to Fortran-9x’s `all` function. Similarly function `or` returns true if any component of its argument is true, mimicking Fortran-9x’s `any` function.

One of the fastest ways of testing whether a vector (or matrix) `v` contains any non-zero element uses `or: or(v)`. As demonstrated with the `find` function, the arguments to `and` and `or` can take arbitrarily complex boolean expressions. If we like to test whether all components of the vector `v = [1.0 0.95 1.02]` are within 10% of the value 1, we do not need a loop: `and(abs(v - 1.0) < 0.1)`.

6.1.3.3.4. Operator “&”, Operator “|”

The operators “&”, and “|” perform a component wise logical-and, or logical-or operation. See also Section 4.3.3. The arguments to either operator can be scalars or matrices.

6.1.3.3.5. `sum`, `cumsum`, `prod`, `cumprod`

These are the numeric cousins of the boolean function pair `or` and `and`. Their syntax is identical. The “cum” functions work cumulatively, returning a vector (matrices are processing in their flattened representation).

A fast factorial function?

```
function f = fact(n)

if n < 0 then
    error("fact: domain")
end

if n == 0 then
```

```

    f = 1
else
    f = prod(1 : n)
end

```

\$1000 at 4.5% over 7 years?

```

->1000.0 * cumprod( (1.0 + 0.045) * ones(7, 1) )
ans =
! 1045.      !
! 1092.025   !
! 1141.1661  !
! 1192.5186  !
! 1246.1819  !
! 1302.2601  !
! 1360.8618  !

```

though `1000.0 * (1.0 + 0.045)^(1:7)'` produces the same result and requires less keystrokes.

6.1.3.3.6. gsort

Warning

Do not use `sort`! It is buggy in that it sometimes does not return a permutation of the input data. Use `gsort` instead of `sort`.

The `gsort` function is a versatile sorting function for vectors and matrices of real numbers or strings. It sorts into increasing order or decreasing (default!) order, sorts a matrix's rows or columns separately, and can sort the rows or columns lexicographically. The output of `gsort` not only is the sorted matrix *mat_sorted* but also the permutation vector *permutation* that generates the sorted matrix from the input matrix. The synopsis is

```
[mat_sorted, permutation] = gsort(mat_input, mode, direction)
```

where *mode* can have the values shown in Table 6-2, and *direction* the values displayed in Table 6-3.

Table 6-2. Mode Specifiers for `gsort`

Specifier	Action	Note
'g'	sort flattened matrix	default
'r'	column-by-column	
'c'	row-by-row	
'lr'	rows lexicographically	not 'rl'!
'lc'	columns lexicographically	not 'cl'!

Table 6-3. Direction Specifiers for `gsort`

Specifier	Action	Note
'i'	increasing order or upgrade	
'd'	decreasing order or downgrade	default

Let us look at some simple examples. We use a numeric matrix in the example, but a string matrix would do as well.

```
->mat1 = [11 12; 21 22; 31 32]
mat1 =
!  11.    12.  !
!  21.    22.  !
!  31.    32.  !
```

```
->gsort(mat1)
ans =
!  32.    21.  !
!  31.    12.  !
!  22.    11.  !
```

```
->gsort(mat1, 'r')
ans =
!  31.    32.  !
!  21.    22.  !
!  11.    12.  !
```

```
->gsort(mat1, 'c')
ans =
!  12.    11.  !
!  22.    21.  !
!  32.    31.  !
```

Applied without parameters `gsort` sorts the flattened (see also Section 6.1.2.3) version, here: `mat(:)`, of its argument into decreasing order. The `'r'`- or `'c'`-options tell `gsort` to sort each column or row separately.

Note: `'r'` means column wise, and `'c'` means row wise!

The next example points out the difference between simple row- or column-sorting and lexicographical sorting of columns or rows.

```
->mat2 = [6 72 23; 56 19 23; 66 54 21]
mat2 =
!  6.    72.    23.  !
!  56.    19.    23.  !
```

```

!   66.   54.   21. !

->gsort(mat2, 'r') // col-by-col
ans =
!   66.   72.   23. !
!   56.   54.   23. !
!    6.   19.   21. !

->gsort(mat2, 'lc') // col lexico
ans =
!   72.   23.    6. !
!   19.   23.   56. !
!   54.   21.   66. !

->gsort(mat2, 'c') // row-by-row
ans =
!   72.   23.    6. !
!   56.   23.   19. !
!   66.   54.   21. !

->gsort(mat2, 'lr') // row lexico
ans =
!   66.   54.   21. !
!   56.   19.   23. !
!    6.   72.   23. !

```

Now what is the exact difference between row-by-row sorting and lexicographic row sorting? After row-by-row sorting (in decreasing order) of an m -times- n matrix a the following relation holds:

$$a_{i,j} \geq a_{i,j+1} \quad \text{for } 1 \leq i \leq m \quad \text{and} \quad 1 \leq j \leq n-1.$$

In other words each row is sorted separately by interchanging its columns. After a lexicographic sort the relation between the rows is:

$$a_{i,:} \geq a_{i+1,:} \quad \text{for } 1 \leq i \leq m-1.$$

This time whole rows are compared to each other. Analogous relations hold for column sorting.

In environments not as rich as Scilab `gsort` might be the heart of user-written `min`, `max`, and `median` functions. All three are predefined in Scilab.

6.1.3.3.7. size

The `size` function handles all shape inquiries. It comes in four different guises. Assuming that `mat` is a scalar or matrix, `size` can be used as all-info-at-once function as in

```
[rows, cols] = size(mat)
```

as row-only, or column-only function

```
rows = size(mat, 'r')
cols = size(mat, 'c')
```

and finally as totaling function

```
elements = size(mat, '*')
```

6.1.3.3.8. matrix

A (hyper-)matrix can be reshaped with the `matrix` command. To keep things simple we demonstrate `matrix` with a 6x2-matrix.

```
->a = [1:6; 7:12]
a =
!  1.    2.    3.    4.    5.    6.  !
!  7.    8.    9.   10.   11.   12.  !

->matrix(a, 3, 4)
ans =
!  1.    8.    4.   11.  !
!  7.    3.   10.    6.  !
!  2.    9.    5.   12.  !

->matrix(a, 4, 3)
ans =
!  1.    3.    5.  !
!  7.    9.   11.  !
!  2.    4.    6.  !
!  8.   10.   12.  !
```

In contrary to the Fortran-9x function `RESHAPE`, `matrix` neither allows padding, nor truncation of the reshaped matrix. Put another way, for a m times n matrix a the reshaped dimensions p , and q must obey

$$mn = pq$$

`matrix` works by columnwise “filling” the contents of the original matrix a into an empty template of a p -times- q matrix. (See also Section 6.1.2.3.) If this is too hard to imagine, the second way to think of it is imagining a as a column vector of dimensions $(m * n)$ -times-1 that is broken down column by column into a p -times- q matrix. In fact this is not pure imagination as for many Scilab matrix operations the identity $a(i, j) == a(i + n*(j - 1))$ holds.

```
->a(2,4)
ans =
    10.
```

```
->a(8)
ans =
    10.
```

Moreover, the usual vector subscripting can be used to a matrix.

```
->a(:)
ans =
!   1.   !
!   7.   !
!   2.   !
!   8.   !
!   3.   !
!   9.   !
!   4.   !
!  10.   !
!   5.   !
!  11.   !
!   6.   !
!  12.   !
```

6.1.4. Evaluation of Polynomials

Once upon a time there was a little Scilab newbie who coded an interface to the `optim` routine to make polynomial approximations easier. On the way an evaluation function for polynomials had to be written. The author was very proud of herself because she knew the Right Thing(tm) to do in this case namely the Horner algorithm. Actually she immediately came up with two implementations.

Example 6-3. Naive functions to evaluate a polynomial

```
function yv = pevall(cv, xv)
// Evaluate polynomial given by the vector its
// coefficients cv in ascending order, i.e.
// cv = [p q r] -> p + q*x + r*x^2 at all
// points listed in vector xv and return the
// resulting vector.

yv = cv(1) * ones(xv)
px = xv
for c = cv(2 : $)
    yv = yv + c * px
    px = px .* xv
end
```

```

function yv = peval2(cv, xv)
// same as peval1

yv = cv($);
for i = length(cv)-1 : -1 : 1
    yv = yv .* xv + cv(i)
end

```

So what is wrong with that? This code looks OK and it does the job. But from the performance viewpoint it is not optimal! The fact that Scilab offers a separate type for polynomials has been ignored. Even if we are forced to supply an interface with the coefficients stored in vectors the built-in function `freq` is preferable.

Example 6-4. Less naive functions to evaluate a polynomial

```

function yv = peval3(cv, xv)
// same as peval1, using horner()

p = poly(cv, 't', 'coeff')
yv = horner(p, xv)

function yv = peval4(cv, xv)
// same as peval1, using freq()
// The return value yv _always_ is a row-vector.

p = poly(cv, 't', 'coeff')
unity = poly(1, 't', 'coeff')
yv = freq(p, unity, xv)

```

Table 6-4 shows the speed ratios (each line is normalized separately) for a polynomial of degree 4 that we got on a P5/166 GNU/Linux system.

Table 6-4. Performance comparison of different polynomial evaluation routines

evaluations	peval1	peval2	peval3	peval4
5	3.5	4.2	1	7.0
1000	1.4	2.5	1	2.5

If we now decide to change our interface to take Scilab's built-in polynomial type the evaluation with `freq` can again be accelerated by a factor of more than three.

6.2. Extending Scilab

The brute force way of getting a better performance is rewriting an existing Scilab script in a low-level language as C, Fortran, or even assembler. This option should be chosen with care, because the rapid prototyping facilities of Scilab are lost. On the other hand if the interface of the function has settled, its performance is known to be crucial and it is of use in future projects then the translation into compiled code could be worth the time and the grief.

In the first part of this section we compare different ways of integrating an external function into Scilab. We focus on the ease of integration versus the runtime overhead introduced. The second part deals with writing the low-level functions themselves, especially their interfaces.

6.2.1. Comparison Of The Link Overhead

We revive our matrix mirroring example from Section 6.1.2.

Our Fortran-77 version looks like this:

```

      subroutine mir(n, m, a, dir, b)
*
*   Mirror n*m-matrix a along direction prescribed
*   by dir.  If dir == 'c' then mirror along the
*   columns, i.e., vertically.  Any other value for
*   dir mirrors along the rows, i.e., horizontally.
*   The mirrored matrix is returned in b.
*
      implicit none

*   ARGUMENTS
      integer n, m
      double precision a(n, m)
      character dir*(*)
      double precision b(n, m)

*   LOCAL VARIABLES
      integer i

*   TEXT
      if (dir(1:1) .eq. 'c') then
        do 100, i = 1, m
          call dcopy(n, a(1, m+1-i), 1, b(1, i), 1)
100      continue
        else
          do 200, i = 1, n
            call dcopy(m, a(n+1-i, 1), n, b(i, 1), n)
200      continue
          end if

```

end

The `dcopy(n, x, incx, y, incy,)` is from BLAS level 1, and copies n double precision elements from vector x in increments of $incx$ to y , where it uses increments of $incy$.

The only thing missing is the glue code between Scilab and `mir`.

```
function b = mirf(a, dir)
// interface function for 'mir.f'
// Behavior is the same as mirror()

[n, m] = size(a)
b = zeros(n, m)

if dir == 'r' | dir == 'c' then
    b = fort('mir', ..
            n, 1, 'i', m, 2, 'i', a, 3, 'd', dir, 4, 'c', ..
            'out', ..
            [n, m], 5, 'd')
else
    error('dir must be "r" or "c"')
end
```

OK, let's lock-and-load. We are ready to rock!

```
link('mir.o', 'mir')
getf('mirf.sci')
```

The fast alternative to using `fort`, which dynamically creates an interface to a C or Fortran function is using **intersci**, which creates an interface suitable for static loading.

intersci can create the Fortran glue code for a C or Fortran function to make it callable from the Scilab interpreter. The glue code is compiled (with a Fortran compiler) and linked to Scilab. **intersci** is described very well in the `SCI/doc/Intro.ps`. Anyhow, Example 6-5 shows the description ("`.desc`") file for our current example. Finally it will supply us with a Scilab function called `mirai(a, dir)`.

Example 6-5. Sample interface description ("`.desc`")

```
mirai  a      dir
a      matrix n      m
dir    string 1
b      matrix n      m

mir    n      m      a      dir      b
n      integer
m      integer
a      double
dir    char
```

```

b          double

out        sequence      b
*
```

We do not want to go into detail here, but a `desc`-file has three parts separated by blank lines: The description of the Scilab-level function's signature (here: `mirai`), the same for the low-level function (here: `mir`), and finally the results' structure. The signatures resemble Fortran or K&R-style C function definitions with the parenthesis missing. The process of passing a `desc`-file through **intersci**, compiling the low-level routine and the glue code can be automated. Example 6-6, a snippet of our `Makefile.intersci` shows the relevant rules.

Example 6-6. `makefile` for static Scilab interfaces via **intersci**

```

ifdef SCI
SCIDIR := $(SCI)
else
SCIDIR := /site/X11R6/src/scilab
endif

%.f.pre: %.desc
        $(SCIDIR)/bin/intersci $*
        mv $*.f $*.f.pre

%.f: %.f.pre
        perl -pe 's#SCIDIR#$(SCIDIR)#' $< > $@

%.o: %.f
        $(FC) $(FFLAGS) -c $<
```

Running the automatically generated Fortran code through a filter (here: **perl**) is necessary to fix the lines include `'SCIDIR/routines/stack.h'`. After everything is compiled a single Scilab command makes the new routine available to the user.

```

addinter(['mirai.o', 'mir.o'], // object files
        'mirai',              // name of interface routine
        'mirai')              // name of new Scilab function
```

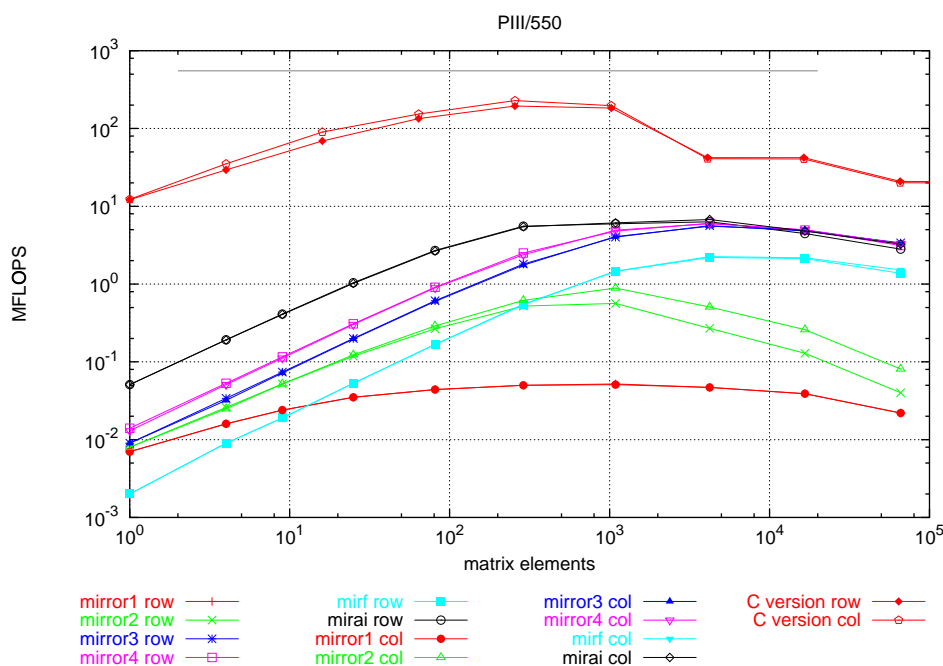
The first argument which almost always is a vector of strings tells Scilab the names of the object files to load. One of them is the interface code made by **intersci**. The rest are the user routines. The second argument specifies name of entry point into the interface routine. The third parameter is the name the new Scilab function will carry.

Entry point of interface function

`addinter`'s second argument must be the name of the *interface routine*, i.e. the one generated by `intersci`. Using the low-level function's entry point here causes Scilab to barf (of course).

Why do we go through that tedious process? After all we are in the performance section, so what we want is speed, high speed, or even better the ultimate speed. Now, we compare all the variants in Figure 6-1.

Figure 6-1. Benchmark results for the `mirror` functions



Performance comparison of `mirror[1-4]`, `mirf`, `mirai`, and a pure C-program doing the same job on a PIII/550 GNU/Linux box. The straight line between (20 elements, 550 MFLOPS) and (20000 elements, 550 MFLOPS) marks the peak performance of the processor.

If we compare the performance of our three Scilab mirror routines `mirror1`, `mirror2`, and `mirror3` together with the two incarnations of the hard-coded routine `mirf`, and `mirai`, we reach at the following conclusions:

- Scilab code that makes heavy use of indexing, like `mirror1`, is extremely slow no matter what problem size. Thumbs down on that one.

- Well written i.e. index-free Scilab code, like `mirror4`, performs well. This is especially true for large vectors or matrices.
- The overhead of the `fort`-call in `mirf` is high; it is hard to amortize for that. `fort` is only justified in situations where a significant amount of time is spent in the low-level user-routine. Usually this will be the case for large problem sizes. Of course the cross-over point must be determined separately in each case.
- A compiled function, integrated with `addinter`, is very fast. `mirai` surpasses all other Scilab-based implementations. For small problem sizes the little overhead in comparison to all the other functions gives this function a factor 10 advantage, though, as the problems size increases `mirai`'s lead is challenged by `mirror4`.
- Of course a carefully hand-optimized C-program outperforms anything. In this figure the plain C-program is meant as a reference what the machine could do, if we pull all registers, or put another way, how much processor power Scilab burns needlessly.

Conclusion: Never underestimate the power of the Emperor^{H^H^H^H^H^H} vectorized Scilab code.

6.2.2. Preparing And Compiling External Subroutines

In this section we will discuss the interfacing of C, C++, Fortran-77, Fortran-9x, or Ada routines with Scilab via `link` command. We restrict ourselves to the simple case of functions that expect exactly one double precision floating point parameter and return a double precision floating point result. Functions with that signature are required e.g. for the integration routine `intg`, or the root finder `fsolve`.

Before we dive into the language specific descriptions, let us point out the main features of Fortran we have to pay attention to when writing an interface in another language.

Function name mangling

A function named `FOO` (`f00`, or whatever capitalization is chosen) in the Fortran source can become a different symbol in the object file. This is compiler dependent. Most often an underscore “`_`” is prepended or appended. Sometimes the name is downcased, sometimes it is upcased.

Tip: The `nm(1)` command provides easy access to the symbols in an object file.

Call-by-reference

Fortran never passes the value of a parameter, but always a pointer to the parameter.

Arrays in column-major order

Arrays are stored so that their leftmost index varies fastest.

6.2.2.1. Fortran-77

*Fortran-77 or how do you want to ruin
your day?*

L. E. van Dijk

Extending Scilab with Fortran-77 is most straightforward. Scilab is written in that language, remember? A Fortran-77 source for function `fals` could look like this:

```
double precision function fals(x)

double precision x

fals = sin(10.0d0 * x)

end
```

After compilation (e.g. `f77 -c fals.f`) the compiled code can be linked to Scilab and called with the integration routine.

```
link('fals.o', 'fals');
[res, aerr, neval, info] = ..
    intals(0.0, 1.0, -0.5, -0.5, 'alg', 'fals')
```

6.2.2.2. Fortran-9x

*Fortran-90? Don't worry, it can't get
much worse.*

Ch. L. Spiel

A bloated, but portable Fortran-90 source for a function could look like this:

```
function fsm(x)
    implicit none
    integer, parameter :: idp = kind(1.0d0)

    ! arguments/return value
    real(kind = idp), intent(in) :: x
    real(kind = idp) :: fsm

    ! text
    fsm = exp(x) / (1.0d0 + x*x)
end function fsm
```

After compilation (e.g. **f90 -c fsm.f90**) the compiled code can be linked to Scilab and called with an integration routine.

```
link('fsm.o', 'fsm');
[ires, ierr, neval] = intsm(0.0, 1.0, 'fsm')
```

6.2.2.3. (ANSI-) C

A simple C function meeting our signature requirements has e.g. this shape:

```
#include <math.h>
#include "machine.h"

double
C2F(fgen)(const double *x)
{
    if (*x > 0.0)
        return 1.0 / sqrt(*x);
    else
        return 0.0;
}
```

After compilation (e.g. **cc -I/site/X11R6/src/scilab/routines -c fgen.c**) the compiled code can be linked to Scilab and called with the integration routine.

```
link('fgen.o', 'fgen', 'c');
[ires, ierr, neval, info] = intgen(0.0, 1.0, 'fgen')
```

There are several ways to get the naming convention differences between Fortran and C right. We show three possible solutions for the case where C uses no decoration at all and Fortran appends one underscore.

```
/* (1) GNU C compiler */
double foo(const double *x) __attribute__((weak, alias ("foo_")));

/* (2) good preprocessor */
#define C2F(name) name##_

/* (3) old preprocessor ;-) */
#define ANOTHERC2F(name) name/**/_
```

None of the above three examples is portable. Therefore, it is prudent to include **SCI/routines/machine.h**, which is automatically generated during the Scilab configuration process and thus knows of the name mangling. Among a lot of other macros it supplies a C-to-Fortran name conversion macro called **C2F**.

6.2.2.4. C++

A C++ source for a function could look like this:

```
#include <math.h>

extern "C"
{
    double C2F(fgk)(const double *x);
}

double
C2F(fgk)(const double *x)
{
    return 2.0 / (2.0 + sin(10.0 * M_PI * (*x)));
}
```

After compilation (e.g. `c++ -I/site/X11R6/src/scilab/routines -c fgen.c`) the compiled code can be linked to Scilab and called with the integration routine.

```
link('fgk.o', 'fgk', 'c');
[ires, ierr, neval, info] = ..
    intgk(0.0, 1.0, 'fgk', 0, %eps, '15-31')
```

See Section 6.2.2.3 for a discussion of the C2F macro.

Further problems arise if the C++ code depends on libraries that have not been linked with Scilab. In the following example `myfct_` is correctly declared, but requires `sqrt` indirectly through a call to `subfct`.

```
// linkcxx.cc
#include <complex>

extern "C" {
    void myfct_(const double *re, const double *im);
}

double_complex subfct(double_complex z);

void
myfct_(const double *re, const double *im)
{
    double_complex u(*re, *im);
    double_complex v(subfct(u));
    // do something with v
}

double_complex
subfct(double_complex z)
{
```

```

    return 1.0 + 0.5 * sqrt(z);
}

```

The problem when linking `myfct_` with Scilab is not the call to `subfct`, but the missing `complex sqrt` function. A listing of the object file's symbols shows the missing function among some functions the (this particular version of `g++`) compiler silently generates due to `inline` expansion.

```

lydia@orion:/home/lydia/tmp $ nm -C linkcxx.o
000000bd t Letext
00000000 ? __FRAME_BEGIN__
00000000 W complex<double> operator/
<double>(complex<double> const &, double)
00000000 W complex<float> operator/
<float>(complex<float> const &, float)
00000000 W complex<long double> operator/
<long double>(complex<long double> const &, long double)
0000008b T main
00000000 T myfct_
                U complex<double> sqrt<double>
(complex<double> const &)
00000046 T subfct(complex<double>)

```

It is up to the programmer to supply *all* necessary libraries – in the correct order – when linking. For the previous example the following call would succeed (on a `libc6` GNU/Linux system):

```

->link("linkcxx.o -lstdc++-2-libc6.1-1-2.9.0")
linking files linkcxx.o -lstdc++-2-libc6.1-1-
2.9.0 to create a shared executable
shared archive loaded
Link done
ans =

    0.

```

In the case that the compiler documentation lacks information about which library defines what symbol, the `nm(1)` command is the most useful tool to find out.

Additional Caveats

The inclusion of C++ modules into a project whose `main()` is not written in C++ call for some additional warnings. See also Section 6.3 for a caveat using compilation switches that break the ABI.

Runtime initialization

When it comes to runtime initialization of his/her code, a C++-programmer depends on the linker as a junkie on his dealer. Either the compiler system does it – and does it right, or you have a very very hard time ahead of you. Sidenote: The GNU linker does the Right Thing(tm)!

exceptions

In brief: Get them – *all*! If the C++ to be linked with Scilab is known to throw exceptions, all interfaced functions of which an exception possibly could escape have to be wrapped in C++-functions that catch these exceptions and translate them into error codes e.g. à la Lapack. Otherwise Scilab is terminated with an `abort()` call.

6.2.2.5. Ada

For GNAT/Ada the package's interface part pulls in the Fortran interface definitions. Is the simplest case the mathematical functions are only instantiated with the type `Double_Precision`. Ada requires to export every function's interface separately, as is clear from the following example.

```
with Interfaces.Fortran;
use Interfaces.Fortran;
with Ada.Numerics.Generic_Elementary_Functions;

package TestFun is
  package Fortran_Elementary_Functions is new
    Ada.Numerics.Generic_Elementary_Functions(Double_Precision);
  use Fortran_Elementary_Functions;

  function foo(x : Double_Precision) return Double_Precision;
  pragma Export(Fortran, foo);
  pragma Export_Function(Internal => foo,
                        External => "foo_",
                        Mechanism => Reference,
                        Result_Mechanism => Value);
end TestFun;
```

According to the interface specification the package body looks like this:

```
package body TestFun is
  function foo(x : Double_Precision) return Double_Precision is
  begin
    return exp(x) / (1.0 + x*x);
  end foo;
end TestFun;
```

The package is compiled as usual **gnatmake -O2 testfun.adb**.

Hint: Make sure that there is a GNAT runtime library `libgnat-3.12p.so`. Your version number may be different, but only the ending ("so") is critical, as `libgnat-3.12p.so.1.7` will not make `dlopen(3)` happy. From now on everything goes downhill, and the function can be linked almost as usual.

```
link('testfun.o -L/site/gnat-3.12p/lib -lgnat-3.12p', 'foo')
```

Again, the path to your gnat-library and the version numbers can differ.

In the case of several functions in the package it is preferable to rely on the extended `dlopen(3)` mechanism, and link the package/library combo with remembering the ID of the shared library.

```
adacode = link('testfun.o -L/site/gnat-3.12p/lib -lgnat-3.12p', 'foo')
```

Linking further functions from the library happens by referencing the number of the library.

```
link(adacode, 'bar')
```

This saves space (Scilab's TRS) and time (to execute the `link`). Speaking about saving, users with a loader e.g. GNU **ld**, capable of incremental linking (e.g. `-i`, `-r`, or `-relocatable`) can of course link `testfun.o` with the (gnat-)library before linking everything to Scilab. To complete the example, here comes the command-line to archive exactly this:

```
ld -i -o testfun-lib.o testfun.o -L/site/gnat-3.12p/lib -lgnat-3.12p
```

In Scilab the arguments to `link` then reduce to

```
link('testfun-lib.o', 'foo')
```

6.2.2.6. Visual C++

by Dave Sidlauskas

This section illustrates the calling of C/C++ routines from the Windows™ version of Scilab using Microsoft™'s Visual C++ compiler. The process is quite simple.

1. Use VC++ create a DLL containing the C functions.
2. In Scilab, use `link()` to load the DLL functions.
3. Use `fort()` to run the functions.

In a little more detail:

1. Use VC++ to create a DLL.

Start VC++, click **FILE**, **NEW**, and select **WIN 32 Dynamic Link-Library**. Give it a name and location and click **OK**. Then select **Empty DLL** and click **Finish**.

Prepare a source file and insert it into the project (**Project**, **Add To Project**). Then build the project (**F7**).

A sample source file is shown below. The declaration `extern "C" declspec(dllexport)` is critical. Using this, the function name is exported correctly with no name mangling. This type of declaration is covered in the VC++ on-line documentation if you wish more details.

Also note that C files that are to be executed by a call to `fort()` are always void, returning no value. Values are returned via pointers in the function parameter list. For example, the parameter `*out` in `matcpy_c` is the return value for that function.

```

extern "C" _declspec(dllexport) void matset_c(double *mat,
                                              const int *nrows,
                                              const int *row,
                                              const int *col,
                                              double *val);

extern "C" _declspec(dllexport) void matcpy_c(const double *in,
                                              const int *nrow,
                                              const int *ncol,
                                              double *out);

// matset

// Set element in mat at row and col to val.
// nrows is number of rows in mat. Shows row
// and col reference in a C function.
// REMEMBER: C row or col = Scilab row or col-1.

void matset_c(double *mat,
              const int *nrows,
              const int *row,
              const int *col,
              double *val)
{
    mat[*row - 1 + (*col - 1)*(*nrows)] = *val;
}

// matcpy

// Function to copy one matrix to another.

void matcpy_c(const double *in,
              const int *nrow,
              const int *ncol,
              double *out)
{
    int row, col;

    for (col = 0; col < *ncol; col++)
        for (row = 0; row < *nrow; row++)
            out[row + col*(*nrow)] = in[row + col*(*nrow)];
}

```

2. In Scilab, use link to load the DLL functions.

```
link("path\filename.dll", "FunctionName", "c")
```

The path is wherever you told VC++ to put your output. It is usually something like ProjectName\debug.

Link uses the Windows™ LoadLibrary function to load your DLL. See the VC++ on-line documentation for details.

3. Use `fort()` to execute your function.

Actually it is probably better to prepare a wrapper function to reduce the clutter of `fort()`. Here is a sample for the `matset` function above.

```
// Wrapper function for calling C language routine matset_c from SciLab

function mat = matset(mat, row, col, val)
m = size(mat);
mat = fort("matset_c",
           mat, 1, "d",
           m(1, 1), 2, "i",
           row, 3, "i",
           col, 4, "i",
           val, 5, "d",
           "out",
           m, 1, "d");
endfunction
```

A sample Scilab session is shown below:

```
->link("d:\vc\sci\debug\sci.dll", "matset_c", "c")
Linking matset_c
Link done
ans =
    0.

->getf('E:\scilab\source\ctest.sci');
->mat = zeros(5, 5);
->matset(mat, 3, 3, 16.71)
ans =
!   0.   0.   0.   0.   0. !
!   0.   0.   0.   0.   0. !
!   0.   0.  16.71  0.   0. !
!   0.   0.   0.   0.   0. !
!   0.   0.   0.   0.   0. !
```

6.2.2.7. Borland C 5.01

by Enrico Segre

These are the steps for creating a DLL with functions, which is callable from Scilab, using Borland® C 5.01 and to link them into Scilab. The process of creating DLL `foo.dll` from source `foo.c`, which defines function `foo` is also simple. The steps are:

1. In BCW, create a new DLL project with `File/New/Project/Target_type→DLL`. Some relevant options are:

Options/Project/16bitCompiler/entry-exit_code/Windows_DLL_all_functions
Options/Project/32bitCompiler/callingConvention/C

2. To this project add file `foo.c`. There is a button for that action in the icon bar.
3. File `foo.c` must contain the following code.

```
#define STRICT
#include <windows.h>

BOOL WINAPI DllEntryPoint(HINSTANCE hinstdll,
                          DWORD fdwReason,
                          LPVOID lpvReserved)
{
    return 1;
}
```

and, to define function `foo` as for example

```
void _export
foo(const double *a, const double *b, double *c)
{
    *c = *a + *b;
}
```

with the keyword `_export` in front of the function's head. File `foo.c` can contain more than one exported function, as well as other functions which are not defined with `_export`, and thus are not entry points for Scilab.

4. Make the DLL (**F9**).

In Scilab, link the function with

```
link('foo.dll', '_foo', 'C')
```

Note the leading underscore! Execute the function with `fort('_foo', ...)` or `call('_foo', ...)`, or even better, define a convenient wrapper function.

```
function foo(a, b, c)
    c = call('_foo',
            a, 1, 'd', b, 2, 'd',
            'out', [1, 1], 3, 'd')
endfunction
```

6.2.3. Pushing It Further

What? What are you doing in this section? Still not satisfied with your functions' performance?—Sorry, but there are no conventional ways to get more out of Scilab. Tinkering with the interface routines is not worth the effort. Some completely new approach is necessary.

6.2.3.1. Scilab as Prototyping Environment

If a problem is too tough, Scilab still can serve as a rapid prototyping environment. One sister program of Scilab, namely *Tela* has been written for exactly this purpose. Prototyping with an interpreted language is currently going through a big revival with C (and C++) developers discovering Python.

As whenever optimization is the final goal, an extensive test suite is the base for success. So one way to proceed could be to develop test routines and reference implementation completely in Scilab. The next step is rewriting the routines *still* in Scilab to match the signatures of for example BLAS/Lapack routines as closely as possible. The test suite can remain untouched in this step. The final step is to migrate the Scilab code to Fortran, C, or whatever, while making extensive use of BLAS/Lapack. Ideally the test suite remains under Scilab and can be used to exercise the new standalone code.

6.2.3.2. Scilab to Fortran-77 Compiler

FIXME: write it!

6.3. Building an Optimized Scilab

One relatively easy way to increase Scilab's performance is recompiling it with a good compiler and an optimized BLAS library².

-
2. Simply linking with an optimized BLAS library generally is not enough. Patches (e.g. “fast-blas”, and “big patch”) to fix

Our experience only suffices to explain the compilation on IA32 GNU/Linux systems. Here, *gcc* or *pgcc* are the compilers of choice.

The following options are a good starting point for further exploration. They apply to compiling Fortran as well as C code.

`-march=arch`

This option instructs *gcc* to generate code specifically for architecture *arch*. Among other things it sets `-mcpu=arch`. Furthermore, it forces `-malign-loops`, `-malign-jumps`, `-malign-functions`, and `-mpreferred-stack-boundary` to their optimum values for the selected architecture *without* braking the ABI. Therefore, it can be considered an optimization switch.

`-malign-double`

For systems with an original Intel®³ Pentium® or above processor this option is an absolute *must*. It forces the alignment of 64 bit floating point numbers (also known as double, double precision, and IEEE754) to a 64 bit boundary. Though *it breaks the ABI*, the gain in speed due to avoiding the misalignment penalty on each memory access is tremendous, even on PentiumPro® and later systems with all write back caches enabled.

Warning

`-malign-double` breaks the ABI!

Code using double compiled with [p]g++-2.95 and `-malign-double` is known to cause segmentation faults under some circumstances.

`-O2`

The workhorse optimization switch, `-O2`, activates a lot of optimizations. See node “Optimize Options” in *gcc*’s info file, e.g. **info -f /usr/info/gcc.info.gz -n "Optimize Options"**

The optimizations toggled on by `-O2` are well tested and do not produce excessively long text.

`-funroll-all-loops`

This switch increases the text size by unrolling as many loops as possible, thereby speeding them up. YMMV.

part of this problem exist. Check out *Hammersmith Consulting’s Scilab patches page* .

3. Intel®, Pentium®, and PentiumPro® are registered trademarks of Intel Corp.

`-fschedule-insns2`

Although the gcc info page states that this optimization is switched on by `-O2`, this might not be true for all versions of gcc floating around. The switch should be particularly helpful on machines with a relatively small number of registers and where memory load instructions take more than one cycle.

Chapter 7. Scilab Core

Aerosmith video “Love In An Elevator”, “Pump” (1989).

Good morning Mister Tyler! Going down?

We are going down all the way right to the core, the core of Scilab. Though this is the most technical and most complex chapter, it is by no means true that writing a native Scilab function is unmanageable by for ordinary mortals. A strict programming discipline, patience, persistence, and a thorough knowledge of what makes up the stack-structures involved, let us overcome the difficulties.

To be able to exactly specify the interface Scilab provides for extensions, we use Ada-like syntax, which is introduced in Section 7.1. Equipped with this explanatory aid of a strongly typed language, we proceed in Section 7.2 by explaining the internal data structures like e.g. the stack. The real meat of the chapter starts in Section 7.3, with an extensive discussion of a native Scilab functions, functionals, and dispatch tables. Closely linked to writing a native function is taking care of the errors on a low-level (We do not mean ignoring them!), a topic that is discussed in Section 7.4.

The lack of a comprehensive and tabular documentation of the Scilab is taken care of in Section 7.5 and Section 7.6, which close the chapter.

7.1. Introduction To Pseudo-Ada

Instead of simply repeating the Fortran-77 and C statements that make up the Scilab stack, the API, etc., we introduce a new language that is better suited for this job: a pseudo-form of Ada¹, called pAda from hereon, which is much more expressive. The syntax follows Ada, and the pAda types are mapped onto Fortran-77 and C types as listed in Table 7-1, Table 7-2, and Table 7-3. What might look like an artificial complication, the introduction of new types, actually is a major simplification (Three cheers for Ada!):

1. The name of the type now makes clear exactly what it is used for.
2. Distinct types designate distinct things, i.e. stuff that never should be mixed up.
3. The valid ranges of the sub-types are explicitly mentioned in the types’ definition.
4. The description of the Fortran-77 interface (Section 7.5) and the C interface (Section 7.6) can be uniformly documented.

1. We apologize to all Ada programmers for the abuse of this wonderful language, but Ada’s expressiveness and clarity are unmatched for the job we have in mind.

Table 7-1. pAda to Fortran-77 and C type mappings – elementary types

pAda	Fortran-77	C
Integer	INTEGER	int
Float	DOUBLE PRECISION, REAL*8	double
Boolean	LOGICAL	n/a, substitute: int, 0 meaning false, everything else meaning true;
Character	CHARACTER	char
type String is array (1..N) of Character	CHARACTER*N	char[N + 1]
subtype Natural is Integer range 0..Integer'Last	INTEGER with the restriction to non-negative values, i.e., allowed are 0, 1, ...	int with the restriction to non-negative values, i.e., allowed are 0, 1, ...

Table 7-2. pAda type mappings – Scilab Fortran-77 interface

pAda	Fortran-77
type ComplexFlag is (RealVariable, ComplexVariable)	INTEGER = 0, 1
type ParameterStackAddress is new Integer range 1..Integer'Last	INTEGER = 1, 2, ...
type DataStackIndex is new Integer range 1..Integer'Last	INTEGER = 0, 1, ...

Table 7-3. pAda type mappings – Scilab C interface

pAda	C
type AccessNatural is access all Natural;	int*, pointer to modifiable integer
type ConstAccessNatural is access constant Natural;	const int*, pointer to read-only integer
type AccessString is access all String	char*, pointer to modifiable character
type ConstAccessString is access constant String	const char*, pointer to read-only character
type TypeString is String (1 .. 1);	char[2] (see also Table 7-4)

pAda		C
-------------	--	----------

```

subtype ParameterStackIndex is      int
Integer range 1..Integer'Last;
type AccessDataStackIndex is access int*
DataStackIndex;

```

7.2. Internal Data Structure

FIXME: explain the parameter stack, data stack, etc.

7.2.1. Parameter Stack And Data Stack

FIXME: follow the documentation in “Internals”.

7.2.2. Storage of Complex Matrices

Many programming languages store scalar complex variables z in Euclidean representation,

$$z = x + iy,$$

where x , and y are real numbers and i denotes the imaginary unit. A complex number is stored in memory as a record.

```

type Complex is record
    RealPart : Float;
    ImagPart : Float;
end record;

```

Fortran chooses to store complex matrices as sequences of Complex, and almost all other programming languages follow this convention.

```

declare
    CpxVec : array (1 .. 10) of Complex;

```

Thus, the memory image of CpxVec, broken into pieces, is

— address — — contents —

```

addr +      0 : CpxVec(1).RealPart
addr + Float'Size : CpxVec(1).ImagPart
addr + 2*Float'Size : CpxVec(2).RealPart
addr + 3*Float'Size : CpxVec(2).ImagPart

```

```

addr + 4*Float'Size : CpxVec(3).RealPart
addr + 5*Float'Size : CpxVec(3).ImagPart
...

```

where `addr` is the start address of the complex vector `CpxVec` in memory. The obvious advantage of this storage scheme is that it can be viewed as a vector of Complex scalars.

```

— address —      — contents —

addr +      0 :   CpxVec(1)
addr + Complex'Size :   CpxVec(2)
addr + 2*Complex'Size :   CpxVec(3)
...

```

Scilab does *not* follow this convention for storing complex numbers, if it did, we would not have to write this section. Instead of storing real and imaginary parts of a complex vector in turn, Scilab separately stores the vector of the real parts, and the vector of the imaginary parts.

Our example vector `CpxVec` from above, gets stored by Scilab in the following way:

```

— address —      — contents —

real_addr +      0 : CpxVec(1).RealPart
real_addr + Float'Size : CpxVec(2).ImagPart
real_addr + 2*Float'Size : CpxVec(3).RealPart
...

imag_addr +      0 : CpxVec(1).ImagPart
imag_addr + Float'Size : CpxVec(2).ImagPart
imag_addr + 2*Float'Size : CpxVec(3).ImagPart
...

```

where `real_addr` and `imag_addr` are the start addresses of the two vectors. Nothing should be assumed of their relation; e.g. `imag_addr` might not point to the first memory cell after the last cell in the vector of the real parts.

The consequence for a Scilab programmer who wants to interface routines that use the conventional (Fortran) storage scheme for complex matrices is that she has to splice real and imaginary parts before calling the routine, and to store them separately after completion. See Example 7-2 for a demonstration of this technique.

Example 7-1 re-implements the multiplication of two complex matrices, `wmmul` in Scilab. For conventional storage the function would be much shorter, for we could use `zgemm` from BLAS to compute the product `C` of two matrices `A` and `B`. `dgemm` and `zgemm` compute `C := Alpha*A*B + Beta*C`. `Alpha` and `Beta` are scalars.

```

type OrientationType is new Character;2

```

```

procedure dgemm
  (OrientationA : in    OrientationType;
   OrientationB : in    OrientationType;
   M            : in    Natural;
   N            : in    Natural;
   K            : in    Natural;
   Alpha        : in    Float;
   A            : in    FloatMatrix;
   LdA          : in    Natural;
   B            : in    FloatMatrix;
   LdB          : in    Natural;
   Beta         : in    Float;
   C            :      out FloatMatrix;
   LdC          : in    Natural);

procedure zgemm
  (OrientationA : in    OrientationType;
   OrientationB : in    OrientationType;
   M            : in    Natural;
   N            : in    Natural;
   K            : in    Natural;
   Alpha        : in    Complex;
   A            : in    ComplexMatrix;
   LdA          : in    Natural;
   B            : in    ComplexMatrix;
   LdB          : in    Natural;
   Beta         : in    Complex;
   C            :      out ComplexMatrix;
   LdC          : in    Natural);

  subroutine wmmul(a, na, b, nb, c, nc, l, m, n)

  call zgemm('n', 'n', l, n, m, 1.0d0, a, na, b, nb,
$      0.0d0, c, l)

```

-
2. A serious Ada interface would not define OrientationType,
but introduce the two types

```

type RealOrientationType is (NoTranspose, Transpose);
type ComplexOrientationType is (NoTranspose, ConjugateTranspose);

```

to let the compiler do the type checking. The BLAS routines even accept *strings* where we use OrientationType. However, a BLAS routine is supposed to look only at the first character. The valid strings are: "No transpose", "Transpose", and "Conjugate tranpose".

end

But as Scilab stores real and imaginary part of a complex matrix separately, we use a Karatsuba multiplication scheme with only three multiplications instead of the four as the naive algorithm does. Expressed in Scilab, we have

```
function [cr, ci] = mul_karatsuba(ar, ai, br, bi)

// fast multiplication of two complex numbers
// z1 := ar + i*ai
// z2 := br + i*bi
// cr + i*ci := z3 = z1 * z2

p1 = ar * br
p2 = ai * bi
cr = p1 - p2

s1 = ar + ai
s2 = br + bi
p3 = s1 * s2
ci = p3 - p1 - p2
```

The actual implementation of `wmmul` is a more space saving version of the above.

```
function [cr, ci] = mul_karatsuba_final(ar, ai, br, bi)

p1 = ar * br
p2 = ai * bi

s1 = ar + ai
s2 = br + bi
ci = s1 * s2
ci = ci - p1 - p2
cr = p1 - p2
```

It is fairly obvious, how big the effort is, even for expressing the algorithm in Scilab. The Fortran function `wmmul` is even more convoluted because of several explicit `do`-loops.

Example 7-1. Multiplication of complex matrices

```
subroutine wmmul(ar, ai, na, br, bi, nb, cr, ci, nc, l, m, n)
*
*   name      : wmmul.f - multiplication of two complex matrices;
*                   c := a * b
*   author    : L. van Dijk
*   last. rev. : Sun Jan 16 22:41:27 UTC 2000
*   Scilab ver.: 2.5
*   compiler   : g77 version 2.95.1 19990816 (release)
```

```

*      Copyright (C) 2000 Lydia van Dijk

*      PARAMETERS
*      ai,ar, bi,br, ci,cr: real and imaginary parts of the respective
*                          matrices
*      na, nb, nc: number of rows of resp. matrix in calling routine
*      l: number of rows in a and c
*      m: number of columns in a, and number of rows in b
*      n: number of columns in b and c
*      implicit none
*      double precision ar(*), ai(*), br(*), bi(*), cr(*),ci(*)
*      integer na, nb, nc, l, m, n

*      LOCAL VARIABLES
*      integer i, j
*      integer ia, ib, ic
*      double precision p1(l, n), p2(l, n)
*      double precision s1(l, m), s2(m, n)

*      TEXT
*      call dgemm('n', 'n', l, n, m, 1.0d0, ar, na, br, nb,
$          0.0d0, p1, l)          - p1 = ar * br
*      call dgemm('n', 'n', l, n, m, 1.0d0, ai, na, bi, nb,
$          0.0d0, p2, l)          - p2 = ai * bi
*      ia = 0
*      do 20 j = 1, m              - s1 = ar + ai
*          do 10 i = 1, l
*              s1(i, j) = ar(ia+i) + ai(ia+i)
10      continue
*          ia = ia + na
20      continue
*      ib = 0
*      do 40 j = 1, n              - s2 = br + bi
*          do 30 i = 1, m
*              s2(i, j) = br(ib+i) + bi(ib+i)
30      continue
*          ib = ib + nb
40      continue
*      call dgemm('n', 'n', l, n, m, 1.0d0, s1, l, s2, m,
$          0.0d0, ci, nc)          - ci = s1 * s2
*      ic = 0
*      do 60 j = 1, n              - ci = ci - p1 - p2
*                                  - cr = p1 - p2
*          do 50 i = 1, l
*              ci(ic+i) = ci(ic+i) - p1(i, j) - p2(i, j)
*              cr(ic+i) = p1(i, j) - p2(i, j)
50      continue
*          ic = ic + nc
60      continue

```

end

7.3. Writing Native Scilab Functions

In the following two sections we shall treat the “anatomy” of native, i.e. low-level Scilab functions. This will confront us with all the gory details of the stack, the low-level API, and the calling conventions. Having the “Guide for Developers”, `Internals.ps` (see also Section 8.2) ready is a good idea. Where the developer guide is at the end of its wits, a study of the source code is appropriate, especially the file `SCI/routines/interf/stack1.f` is of interest for us.

We start out discussing simple functions in Section 7.3.1. Simple in the sense that they are self-contained and only take non-function parameters as their arguments. In the second part, Section 7.3.2, we shall consider functions that take other functions (either Scilab functions or externals) as arguments. These functionals all rely on correctly set up deisplatch tables, which are treated in Section 7.3.3.1.

7.3.1. Simple Functions

A typical native Scilab function proceeds as follows:

1. Check the number of input and output parameters.
2. Get the “pointers” to actual input parameters; supply default values for optional parameters; issue warnings or errors as appropriate if too many or too few parameters are supplied.
3. Allocate space for temporary variables, “workspace(s)”, etc.
4. It might be necessary to translate the input variables which are in Scilab format into the appropriate format for the worker routine. This happens for example if the worker routine uses Fortran-77’s `double complex` (or equivalently `complex*16`) variables. See Section 7.2.2 for details.
5. Perform the calculations or transformations that *really* make up the procedure.
6. As in Step 4, it might be necessary to transform the results, now from the worker routine’s format back into Scilab format.
7. If necessary, allocate space for the return value(s) on the Scilab stack, and copy result(s) to this space.

Now that the general outline is clear, we are ready to dissect a simple function: `ortho`. It takes exactly one argument `a`, that is a real or complex m times n matrix. The single output parameter is a matrix of the same shape and type as is the input matrix. The duty of `ortho` is to transform the

columns of the input matrix into orthonormal form; to achieve this we employ the following Lapack functions:

```

type Complex is record
    RealPart : Float := 0.0;
    ImagPart : Float := 0.0;
end record;

type FloatVector is array (Positive range <>) of Float;
type ComplexVector is array (Positive range <>) of Complex;
type FloatMatrix is array (1..Lda, Positive range <>) of Float;
type ComplexMatrix is array (1..Lda, Positive range <>) of Complex;

procedure dgeqrf
    (M      : in      Natural;          - number of rows of A
     N      : in      Natural;          - number of cols of A
     A      : in out FloatMatrix;       - M-by-N matrix
     Lda    : in      Natural;          - leading dimension of A
     Tau    : out FloatVector;          -
    scalar factors of elementary reflectors
     Work    : out FloatVector;          - workspace
     Lwork   : in      Integer;          - size of workspace Work
     Info    : out Integer);            - error indicator

procedure dorgqr
    (M      : in      Natural;          - number of rows of A
     N      : in      Natural;          - number of cols of A
     K      : in      Natural;          - number of elementary reflectors
     A      : in out FloatMatrix;       - M-by-N matrix
     Lda    : in      Natural;          - leading dimension of A
     Tau    : out FloatVector;          -
    scalar factors of elementary reflectors
     Work    : out FloatVector;          - workspace
     Lwork   : in      Integer;          - size of workspace Work
     Info    : out Integer);            - error indicator

procedure zgeqrf
    (M      : in      Natural;          - number of rows of A
     N      : in      Natural;          - number of cols of A
     A      : in out ComplexMatrix;     - M-by-N matrix
     Lda    : in      Natural;          - leading dimension of A
     Tau    : out ComplexVector;        -
    scalar factors of elementary reflectors
     Work    : out ComplexVector;        - workspace
     Lwork   : in      Integer;          - size of workspace Work
     Info    : out Integer);            - error indicator

procedure zungqr
    (M      : in      Natural;          - number of rows of A

```

```

    N      : in      Natural;          - number of cols of A
    K      : in      Natural;          - number of elementary reflectors
    A      : in out ComplexMatrix;     - M-by-N matrix
    Lda    : in      Natural;          - leading dimension of A
    Tau    : out ComplexVector;        -
scalar factors of elementary reflectors
    Work   : out ComplexVector;        - workspace
    Lwork  : in      Integer;          - size of workspace Work
    Info   : out Integer;              - error indicator

procedure dcopy
    (N      : in      Natural;          - number of elements to copy
    X      : in      FloatVector;      - input vector
    IncX   : in      Integer;          - input stride
    Y      : out FloatVector;          - output vector
    IncY   : in      Integer);         - output stride

```

The `dgeqrf`- and `zgeqrf`-functions compute a QR-factorization of a real or complex m -by- n matrix a , while the `dorgqr`-, and `zungqr`-functions generate an m -by- n real or complex matrix q with orthonormal columns, relying on the QR-factorization of `dgeqrf` or `zgeqrf`. Function `dcopy` copies N elements (of type `Float`) of the vector X in increments of $IncX$ to the vector Y using increments of $IncY$ on that side. For a detailed description please consult the Lapack User Guide, or the appropriate manual pages. For the mathematics behind the operation, the reader is referred to [Golub:1996].

Example 7-2 is one of the longest examples in the running text, but do not be scared as we will explain line-by-line and variable-by-variable what is there and why.

Example 7-2. Simple native Scilab function

```

    subroutine ortho                                     -
Native functions are parameterless

    implicit none                                       -
Switch into weeny mode :-)

    *          CONSTANTS
    integer reatype
    parameter (realtype = 0)                           - See Table 7-
    2 for type association

    *          LOCAL VARIABLES
    character*6 fname                                   -
name of the routine as string

    logical checklhs, checkrhs, cremat, getmat          - Scilab API functions

    integer topk
    integer n, m, mattyp

```

```

integer tausz, worksz, info
integer areadr, aimadr, badr, tauadr
integer wrkadr, readr, rimadr, dumadr

*      EXTERNAL FUNCTIONS/SUBROUTINES
external checklhs, checkrhs, cremat, getmat          – Scilab API functions
external error

external dcopy, dgeqrf, dorgqr, zgeqrf, zungqr      –
LAPACK/BLAS worker subroutines

*      HEADER
include '/site/X11R6/src/scilab/routines/stack.h'   – Scilab API header

*      TEXT
fname = 'ortho'                                     –
Function name (for error messages)
topk = top                                           –
top is defined in stack.h
rhs = max(0, rhs)

if (.not. checkrhs(fname, 1, 1)) return              ❶
if (.not. checklhs(fname, 1, 1)) return

*      fetch input parameters                        ❷
if (.not. getmat(fname, topk, top - rhs + 1,
$          mattyp, m, n, areadr, aimadr)) return

if (n * m .eq. 0) return                             –
Quick return on empty matrix

tausz = min(m, n)                                     – Prescribed by man-
page
worksz = max(1, n)                                    – ... same here

if (mattyp .eq. realtype) then
*      real case

*      allocate temporary variables; all are real  ❸
if (.not. cremat(fname, top + 1, realtype, tausz, 1,
$          tauadr, dumadr)) return
if (.not. cremat(fname, top + 2, realtype, worksz, 1,
$          wrkadr, dumadr)) return
if (.not. cremat(fname, top + 3, realtype, m, n,
$          badr, dumadr)) return

*      prepare worker routines' input parameters  ❹
call dcopy(n * m, stk(areadr), 1, stk(badr), 1)

```

```

*      call worker routines ⑤
      call dgeqrf(m, n, stk(badr), m, stk(tauadr),
$      stk(wrkadr), worksz, info)
      if (info .ne. 0) then —
Any error is considered fatal
      buf = fname // ' dgeqrf failed'
      call error(999)
      return
    endif

      call dorgqr(m, n, tausz, stk(badr), m, stk(tauadr),
$      stk(wrkadr), worksz, info)
      if (info .ne. 0) then —
Any error is considered fatal
      buf = fname // ' dorgqr failed'
      call error(999)
      return
    endif

    else
*      complex case; mattyp != realtype

*      allocate temporary variables,
*      use two REAL*8 for one COMPLEX*16 ⑥
      if (.not. cremat(fname, top + 1, realtype, 2 * tausz, 1,
$      tauadr, dumadr)) return
      if (.not. cremat(fname, top + 2, realtype, 2 * worksz, 1,
$      wrkadr, dumadr)) return
      if (.not. cremat(fname, top + 3, realtype, 2 * m, 2 * n,
$      badr, dumadr)) return

*      prepare worker routines' input parameters, joining
*      two REAL*8 arrays into one COMPLEX*16 array ⑦
      call dcopy(n * m, stk(areadr), 1, stk(badr), 2)
      call dcopy(n * m, stk(aimadr), 1, stk(badr + 1), 2)

*      call worker routines ⑧
      call zgeqrf(m, n, stk(badr), m, stk(tauadr),
$      stk(wrkadr), worksz, info)
      if (info .ne. 0) then —
Any error is considered fatal
      buf = fname // ' zgeqrf failed'
      call error(999)
      return
    endif

      call zungqr(m, n, tausz, stk(badr), m, stk(tauadr),
$      stk(wrkadr), worksz, info)

```

```

        if (info .ne. 0) then
Any error is considered fatal
            buf = fname // ' zorgqr failed'
            call error(999)
            return
        endif

    endif

*   get ready to exit
    if (lhs .ge. 1) then
        if (.not. cremat(fname, top, mattyp, m, n,
$       rreadr, rimadr)) return
        if (mattyp .eq. realtype) then
            call dcopy(m * n, stk(badr), 1, stk(rreadr), 1)
        else
            call dcopy(m * n, stk(badr), 2, stk(rreadr), 1)
            call dcopy(m * n, stk(badr + 1), 2, stk(rimadr), 1)
        endif
    endif

end
end

```

- ❶ Check the number of input and output parameters. The task is easy as we receive one and return exactly one parameter. This line and the next correspond to Step 1.
- ❷ Get the addresses – as mentioned in Step 2 – of the real, and imaginary part of the matrix passed as (only) parameter to `ortho`. Note that `getmat` will return `False` if the parameter at the given parameter stack position is not a matrix of numbers.

Function `getmat` is called with the second parameter, `topk`, holding the value of the parameter stack pointer when the control flow entered `ortho`. This as well as the function name passed in `fname` is necessary for the cleanup and messaging in case of an error.

The only parameter we use sits on top of the parameter stack for `top - rhs + 1` equals `top` in our case.

On successful return `getmat` not only sets the data stack addresses `areadr`, and `aimadr` of the real and imaginary parts, but also tells us via `mattyp` whether the matrix is real complex, and via `m`, and `n` how large the matrix is.

The following lines directly depend on the sizes passed back from the core interface, calculating the necessary space for two scratch arrays.

- ❸ Allocating space for the temporary variables `tau`, `work`, and `b` on the data stack is a realization of Step 3. The variables `tau` and `work` are necessary because of the Lapack routines used; `b` is a copy of `a` as the Lapack routines, `dgeqrf`, and `zgeqrf`, modify the matrix in place, i.e. would mangle the input variable `a`. The temporaries are accessed the same way parameters are

accessed: through indices into the data stack. These indices are `tauadr`, `wrkadr`, and `badr`. Their positions on the parameter stack are `top + 1`, `top + 2`, and `top + 3`, respectively.

We request a purely real storage for each of the three temporary variables, the third parameter being `realttype = 0`. Therefore, the index for the imaginary part is a dummy index, called `dumadr`.

- ④ There is no “translation” to do in the real case. So Step 4 is quite simple. The input variable – of which we definitely know that it is real – is simply copied to the scratch space that we have allocated on the data stack.

Note how the *address* of the matrices is passed. The idiom is `stk(index)`, where *index* has been obtained through one of the `get*`-, or `cre*`-functions. The mnemonic “stk” means data stack.

- ⑤ Everything is set up correctly and initialized. We have reached Step 5. The worker routines can take over now.
- ⑥ In the complex case the allocation of the temporaries variables requires a bit more thought, although it is again just Step 3. We know that the Lapack routines need the complex vectors/matrices in packed form. Thus, we allocate *one* real (double precision) vector/matrix of twice the size each time thereby accommodating the storage requirement of complex (double complex, or `complex*16`) variables. Otherwise this step proceeds as in the real case.
- ⑦ Because of the different handling of complex variables in Scilab and Lapack, Step 4 requires two calls to the copy function.

```
call dcopy(n * m, stk(areadr), 1, stk(badr), 2)
call dcopy(n * m, stk(aimadr), 1, stk(badr + 1), 2)
```

The first line says: “Copy *m* times *n* elements from the first position of the double precision variable `stk(areadr)` taking each entry (3rd parameter, read stride = 1) to the double complex output variable `stk(badr)` filling every other entry (5th parameter, write stride = 2).” The second line does almost the same, but starts off writing at the second element `stk(badr + 1)`, therefore filling in the imaginary parts of `stk(badr)`. This corresponds to Step 4.

- ⑧ Again we have reached Step 5; everything is set up correctly and initialized. The worker routines can take over.
- ⑨ If there is an output variable, we copy the results into it. Otherwise, we skip the expensive copy operation.
- (10) At this point a purely real result, `stk(badr)`, can simply be copied to the output parameter, `stk(rreadr)`.

The situation is a bit more complicated for a complex result, as we have to de-splice the double complex result from Lapack into two double precision matrices. Here are the crucial lines again:

```
call dcopy(m * n, stk(badr), 2, stk(rreadr), 1)
call dcopy(m * n, stk(badr + 1), 2, stk(rimadr), 1)
```

The first line says: “Copy m times n elements from the first position in the double complex result `stk(badr)` taking every other entry (3rd parameter, read stride = 2) into the double precision output variable `stk(rreadr)` filling each entry (5th parameter, write stride = 1).” The second line does almost the same, but starts off at the second element, `stk(badr + 1)`, therefore copying the imaginary parts into `stk(rimadr)`. This way we are merging Step 6, and Step 7 into one.

7.3.2. Functionals

Func what? What are you talking about? Functionals – what is this? Glad you asked! Functions operate on numbers or variables, which themselves are not functions. The square root function for example is usually applied to numbers (like: `sqrt(2)`) or more generally to variables (like: `sqrt(x)` for any real x). Functionals operate on other functions. Prominent examples are differentiation

$$\frac{df}{dx}(x_0) := \lim_{x \rightarrow x_0} \frac{f(x) - f(x_0)}{x - x_0},$$

where f has to fulfill certain continuity requirements at the point x_0 ; integration:

$$\begin{aligned} \int \alpha f(x) + \beta g(x) dx &= \alpha \int f(x) dx + \beta \int g(x) dx && \text{linear} \\ f(x) \leq g(x) \quad \text{for all } x &\Rightarrow \int f(x) dx \leq \int g(x) dx && \text{monotonic} \\ \int f(x - x_0) dx &= \int f(x) dx && \text{invariant under translation} \\ \int_0^1 1 dx &= 1 && \text{normalized,} \end{aligned}$$

where again f has to fulfill certain (interesting) requirements; and Fourier transformation:

$$F[f(x)](p) := \frac{1}{\sqrt{2\pi}} \mathcal{I}[f(x) \exp(2\pi xp)](p)$$

for suitable functions f , and integrals \mathcal{I} .

The question how to write native Scilab functions that take arbitrary non-function parameters as their arguments has been discussed in the previous section. Now we focus on Scilab functions that take other Scilab functions as their arguments. If the reader does not feel familiar with native Scilab functions, she should reconsider Section 7.3.1.

In a similar manner as in the last section, we introduce an example. The example is taken from our Scilab/Quadpack interface available on the web. Among others it features the integrator `dqng` for sufficiently smooth functions, which has the following signature:

```
type SimpleFunctionType is access
  function(X : in Float) return Float;

procedure dqng
  (Function          : in      SimpleFunctionType;
   LowerIntervalEnd  : in      Float;
   HigherIntervalEnd : in      Float;
   EpsilonAbsolute   : in      Float;
   EpsilonRelative   : in      Float;
   Result            : out     Float;
   ErrorAbsolute     : out     Float;
   NumberOfEvaluations : out   Natural;
   ErrorIndicator    : out     Natural);
```

Here comes the complete example.

Example 7-3. Scilab functional

```
subroutine intsm
*
*   name:          intsm.f - Scilab/F77 interface to QUADPACK's dqng
*   author:        Lydia van Dijk
*   last rev.:     Wed Mar 15 23:49:45 UTC 2000
*   scilab ver.:   2.5
*   compiler:      g77-0.5.25 (Linux 2.3.49)
*
*   Scilab signature:
*       [res, abs_err, n_eval] = intsm(a, b, f, eps_abs, eps_rel)
*
*   Return Values:
*       res:        value of the integral
*       abs_err:    estimate of the absolute error
*       n_eval:     number of function evaluations
*
*   Arguments (mandatory):
*       a:          lower bound of integral
*       b:          upper bound of integral
*       f:          function to integrate with signature y = f(x),
*                  x, y real scalars
*
*   Arguments (optional):
*       eps_abs:    desired absolute error; default: 0.0
*       eps_rel:    desired relative error; default: 1e-8
```

```

        implicit none
Switch into weeny mode :-)
```

—

```

        include 'stack.h'

        common /cintg/ namef
Name of integrand function
```

—

```

        external bintg, fintg
gateways, see Section 7.3.3.1
        external setfintg
```

—

```

*      LOCAL VARIABLES
        character*6 namef
Name of the routine as string
        character*6 fname
Name of function to be integrated
        character*8 errstr
```

—

```

        logical getexternal, getscalar
        logical type, cremat

        integer iadr, sadr, neval, ifail, l, idxf, idxa
        integer topk, lr, lra, lrb, iipal, dummy

        double precision epsa, epsr, a, b, val, abserr
```

—

```

        include 'errnum.inc'
Error numbers are defined here
```

—

```

*      STATEMENT FUNCTIONS
        iadr(1) = 1 + 1 - 1
Accessor for integers on real*8 stack
        sadr(1) = 1/2 + 1
Accessor for real* on integer stack
```

—

```

*      TEXT
        fname = 'intsm'
Name of this function
        if(rhs .lt. 3 .or. rhs .gt. 5) then
            call error(39)
            return
        endif
        topk = top
```

—

Remember stack position

```

*      pop optional parameters off the stack
        if(rhs .eq. 5) then
            if (.not. getscalar(fname, topk, top, lr)) return
```

—

```

        epsr = stk(lr)
        top = top - 1
    else
        epsr = 1.0d-8
    endif
    endif

    if (rhs .ge. 4) then
        if (.not. getscalar(fname, topk, top, lr)) return
        epsa = stk(lr)
        top = top - 1
    else
        epsa = 0.0d0
    endif

*      pop mandatory parameters off the stack
    namef = '          '
string with 6 spaces
    type = .false.
    if (.not. getexternal(fname, topk, top, namef, type, setfintg)) ❸
$      return
    idxf = top
Remember stack position of function f
    top = top - 1

    if (.not. getscalar(fname, topk, top, lrb)) return
    b = stk(lrb)
    top = top - 1

    if (.not. getscalar(fname, topk, top, lra)) return
    a = stk(lra)
    idxa = top
Remember stack position of argument a
    top = topk + 1

*      call integration routine
    if (type) then
*      compiled external
        call dqng(fintg, a, b, epsa, epsr, val, abserr, neval, ifail)
    else
*      Scilab macro
        iipal = iadr(lstk(top))
Start building a variable description
        istk(iipal) = 1
        istk(iipal + 1) = iipal + 2
        istk(iipal + 2) = idxf
        istk(iipal + 3) = idxa
        lstk(top + 1) = sadr(iipal + 4)
        call dqng(bintg, a, b, epsa, epsr, val, abserr, neval, ifail)
    endif

```

– Scilab default

– Scilab default

– Fill name-

–

–

– Reset stack index

❹

❺

– ?

– ?

– ?

– ?

– ?

```

        if (ifail .eq. 1) then
            buf = fname // ': max. number of steps reached; '
        $
            // 'integral too difficult for int_sm'
            call error(emaxdiv)
            return
        endif
        if (ifail .eq. 6) then
            buf = fname // ': invalid error bounds'
            call error(ebounds)
            return
        endif
        if (ifail .ne. 0) then
    *
        catch all other errors
            write(errstr, '(I10)') ifail
            buf = fname // ': unknown error ' // errstr
            call error(eunknown)
            return
        endif

    *
        return values #1, and #2 (val, abserr) replace arguments #1, and
    *
        #2 (a, b).
        top = topk - rhs + 1
        stk(lra) = val
        if (lhs .ge. 2) then
            top = top + 1
            stk(lrb) = abserr
        endif

    *
        return value #3, neval, needs space on the stack
        if (lhs .ge. 3) then
            top = top + 1
            if (.not. cremat(fname, top, 0, 1, 1, lrb, dummy)) return
            stk(lrb) = dble(neval)
        endif
        neval is int, stk() is double precision
    *
        endif

    end

```

- ❶ Here, we do not rely on the predefined number-of-arguments checking functions, `checklhs` and `checkrhs`, but set up our own scheme. `intsm` will require three mandatory arguments, `a`, `b`, `f`, and two optional ones, `eps_abs`, `eps_rel`, making a total of five.
- ❷ Take care of the optional parameters: fetch them from the stack, or use a default value if the actual parameter is omitted.
- ❸ Fetch mandatory parameters from the stack. The stack index, `top` is decremented with each parameter. This is a slight variation of the code shown in Example 7-2, where we keep the stack index fixed and add an appropriate offset when fetching the parameter from the stack.

- ④ `getexternal` returns the type of the external after a successful call. An external, i.e. object code linked to Scilab, sets `type = 1`, a macro – defined via `deff`, or function – sets `type = 0`.

The case of an external is easy to handle as `getexternal` has already taken care of initializing the address to be called `fintg`. A call to `setfintg` accomplishes this.

- ⑤ Calling a Scilab macro is much more involved as it requires to manually set up a function activation record (“calling frame”).

FIXME: add text here.

- ⑥ The return code from the integration routine `dqng` is checked, and errors are handled as described in Section 7.4.
- ⑦ For `intsm` returns a scalar *and* the first argument is a mandatory scalar too, we do not need to reserve space for the value of the integral, `val`. The result is simply copied into the argument’s stack position.

Almost the same holds for the second return value, `abserr`, though we only can use its slot if there actually is a return variable.

- ⑧ The third return value is a scalar, but the third argument is a function, so we cannot apply our previous technique again. `cremat` reserves the space for `neval`.

7.3.3. Library Interfaces

When finally a Fortran-77 or C function to extend Scilab has been completed, it must be registered with the interpreter. The registration is done from the command line with function `addinter`, and happens indirectly via so called dispatch tables or “gateways” in INRIA parlance.

Dispatch tables are either set up manually, a technique that is explained in Section 7.3.3.1, or they are generated automatically – which is of course much easier – with function `ilib_build`. The auto-generation of gateways will be explained in Section 7.3.3.2.

7.3.3.1. Dispatch Tables

A dispatch table or “gateway” connects the identifiers of the extension functions (entry points of the functions) with the names the functions will carry in the Scilab environment. For example, Scilab must be taught that the C-function `my_fun`, is called `func1` at the prompt.

Dispatch tables are implemented by arrays of struct `GenericTable`. The structure is defined in `mex.h`.

```
typedef int (*GatefuncH) (int nlhs, Matrix *plhs[], int nrhs, Matrix *prhs[]);
typedef int (*Myinterfun) (char*, GatefuncH F);

typedef struct
```

```

{
    Myinterfun f;      – interface
    GatefuncH F;      – function identifier
    char *name;       – function name in Scilab
} GenericTable;

```

A typical interface looks like the following piece of C-code.

```

#include <mex.h>

extern Gatefunc C2F(my_fun);
– more functions can be declared here

static GenericTable Tab[] =
{
    { (Myinterfun)sci_gateway, C2F(my_fun), "func1" }
    – more functions can be registered here
};

int
C2F(lib_my_lib)()
{
    Rhs = Max(0, Rhs);
    (*(Tab[Fin-1].f))(Tab[Fin - 1].name, Tab[Fin - 1].F);
    return 0;
}

```

The interface is activated at the command line by `addinter(["libmylib.so", "my_fun.so"], "lib_my_lib", ["func1"])`, where the square brackets could take more files or function names. It has been assumed that the gateway has been compiled into the file `libmylib.so`, and the user's function into the file `my_fun.so`. The second argument to `addinter` is the name of the dispatching function itself (as a string). The third and last parameter to `addinter` lists all function names to be registered.

Important: The order of the functions to be registered must be the same in `Tab`, this is, in the C-file and in the `addinter` call, this is, at interpreter level!

Here is another example of a gateway.

```

/*
 * name:      quadqack-gw.c  -  gateway for all QUADPACK
 *                               interface routines
 * author:    Lydia van Dijk
 * last rev.: Wed Mar 15 02:22:02 UTC 2000
 * compiler:  pgcc-2.95.2 19991024 (Linux 2.3.49)
 */

#include <stack-c.h> /* lives in $SCI/routines */

```

```
typedef void (*gatef_t) (void);
```

```
extern void C2F(intals)(void);
extern void C2F(intcau)(void);
extern void C2F(intexc)(void);
extern void C2F(intfou)(void);
extern void C2F(intgen)(void);
extern void C2F(intgk)(void);
extern void C2F(intinf)(void);
extern void C2F(intosc)(void);
extern void C2F(intsm)(void);
```

```
static gatef_t gftab[] = {
    C2F(intals),
    C2F(intcau),
    C2F(intexc),
    C2F(intfou),
    C2F(intgen),
    C2F(intgk),
    C2F(intinf),
    C2F(intosc),
    C2F(intsm)
};
```

```
int
C2F(quadpack_gw)(void)
{
    (*gftab[Fin - 1])();
    return 0;
}
```

Scilab script part ...

```
quadpacklibs = ['/site/src/netlib/quadpack/libquadpack-1.0.so', ..
                '/site/src/netlib/quadpack/intersci/libqpif-1.0.so']
gateway = 'quadpack_gw' // name of the gateway function
interfaces = ['intals', 'intcau', 'intexc', 'intfou', ..
              'intgen', 'intgk', 'intinf', 'intosc', ..
              'intsm']
```

```
addinter(quadpacklibs, gateway, interfaces)
```

The complete example can be found in Section 10.7.

7.3.3.2. Interface Generator

The previous section has shown that manually setting up an interface is a rather complicated process. However, Scilab can take over most of the tedious and error-prone part. The Scilab function that does all the magic is `ilib_build`; it accepts four or five parameters:

```
ilib_build
    (library_name,
     function_table,
     function_files,
     extra_libraries,
     [makefile_name])
```

`library_name` is the name of the library. The resulting interface (shared) object file will be `library_name.o` or `library_name.so`. `library_name` is a n -times-2 string matrix that lists the C- or F77-function name/Scilab function name pairs. For the remaining arguments the the man-page.

Usually, the call to `ilib_build` is wrapped in the Scilab script `builder.sce`. However, any other name is possible, too.

```
// builder.sce

func_files = ["rot90.o"];
func_table = ..
[ ..
  "rot90", "C2F(rot90)" ..
];

ilib_build("librot90", func_table, func_files, []);

exit;
```

Note: Do not forget the `exit` command at the end of `builder.sce` if you want to use the script non-interactively.

With the help of `ilib_build` a Makefile for a Scilab-extension condenses into a few simple rules as is done in the Makefile below.

```
# Makefile for Scilab extension 'rot90'

include Makelib

.phony: all
all:: librot90.so
```

```
.phony: test
test:: all
    scilab -nw -f tester.sce | tail +12

.phony: clean
clean::
    rm -f librot90.* *.lo loader.sce Makelib

.phony: distclean
distclean:: clean
    rm -f *~ core
```

```
Makelib:    # empty rule; "Make-
lib" is made by ilib_build() in "builder.sce"
```

```
librot90.so: rot90.c
    scilab -nw -f builder.sce | tail +12
```

In fact the example is so simple that `builder.sce` is not even necessary, and the `ilib_build` call can be fed directly into Scilab.

```
librot90.so: rot90.c
    echo 'ilib_build("librot90", ["rot90", "C2F(rot90)"], "rot90.o", []);' \
    | scilab -nw -f | tail +12
```

Given the above files, `Makefile`, `builder.sce`, and the C-source `rot90.c` and the test script `tester.sce`, building the extension could proceed as follows.

```
$ ls -l
Makefile
builder.sce
rot90.c
tester.sce

$ make
Makefile:4: Makelib: No such file or directory
scilab -nw -f builder.sce | tail +12
    generate a gateway file
    generate a loader file
    generate a Makefile: Makelib
    running the makefile

$ ls -l
Makefile
Makelib
builder.sce
librot90.a
```

```

librot90.c
librot90.la
librot90.lo
librot90.o
librot90.so
loader.sce
rot90.c
rot90.lo
rot90.o
tester.sce

$ make test
scilab -nw -f tester.sce | tail +12
Loading shared executable ./librot90.so
shared archive loaded
Linking librot90
Interface 0 librot90

passed 18 tests.

```

7.4. Error Handling

We briefly discuss how to produce the three possible classes of errors: fatal, warning, and message in Scilab.

7.4.1. Fatal Errors

To signal a fatal error condition in an interface procedure, call `error` with the appropriate code. The codes can be looked up in `SCI/routines/system/error.f`.

Here is a code snippet that does this.

```

if (ifail .eq. 2) then
    call error(1232)
    return
endif

```

If there is no suitable error message, place your own message (length ≤ 80 chars) in the global variable `buf`, and call `error` afterwards.

Warning

The string placed in `buf` must not be longer than 80 characters.

```

if (ier .eq. 6) then
    buf = 'invalid limits'
    call error(32253)
    return
endif

```

Sideeffect of calling `error`: The Scilab stack is cleaned up, i.e. put back in the state it was just before the user routine has been entered.

On the Scilab interpreter level an error terminates the evaluation of whatever is currently evaluated (expression, file, or string), unless the trapping of errors has been modified by `errcatch`. See also the interpreter functions `errclear`, and `iserror`.

7.4.2. Warnings

To signal non-fatal error conditions (also known as soft-errors, or warnings), place a negative integer in `err2`, and call `out` to display the warning message. Depending on the situation a `return` may be issued after that. The Scilab stack is *not* cleaned up, which means all return values from the interface routine are passed back normally. This is the solution of choice if the user can decide how to proceed, based on the return values.

Again, here is a small piece of code for demonstration.

```

if (fail .eq. 1) then
    err2 = -6343
    call out('reached table limit')
    return
endif

```

On interpreter level it is now mandatory to call `iserror` after a call to a routine that issues warnings like this. In the user-level error handler the error code *must* be reset by `errclear` to allow for further warnings to be received.

A typical way of coping with these soft-errors in the interpreter level is shown in Example 7-4.

Example 7-4. Handling of warnings in Scilab

```

[z, n, info] = abraxas(a, b, foo, limit)
if iserror(-19) then
    errclear(-19)
    limit = limit / 2    // make it easier
    [z, n, info] = abraxas(a, b, foo, limit)
    if iserror(-19) then
        errclear(-19)
        error(failed even with easy limit');
    end
end

```

7.4.3. Messages

Messages are the least severe class of errors. Sometimes they are not really errors, but just additional information that something unexpected is going on. No news is good news.

We have already seen the appropriate subroutine in action. It is out.

```
if (iter .gt. 1000) then
    call out('iterating excessively')
endif
```

7.5. Fortran Interface to Scilab's Core

The interface to Scilab's core is widely undocumented. What is missing from the official documentation will be described in the following sections.

There are two levels of interface functions, a lower-level Fortran77-derived, and an interface that resembles Scilab's C-interface (see also Section 7.6). Up to Scilab-2.5 (official release) the lower-level API was defined in `SCI/routines/interf/stack1.f`, but from Scilab-2.5.1 (alpha version) on it is defined in `SCI/routines/interf/stack1.h` and `SCI/routines/interf/stack1.c`. This means that the implementation has been ported from Fortran77 to C. The higher-level API is defined in `SCI/routines/interf/stack2.h` and `SCI/routines/interf/stack2.c`.

All lower-level functions expect the user-function name in the first parameter, whereas the higher-level functions need a variable of type `ParameterStackIndex`.

To save the reader frequent lookups in the defining files, we have compiled the most important ones in the following sections: query, access, creation of objects, and miscellaneous functions.

7.5.1. Query

The functions in this group retrieve information about the parameters a function has been called with, and about the properties of objects on the stack.

7.5.1.1. `checkrhs`

Synopsis

```
function CheckRhs
    (SelfName      : in String;
     MinNumParameter : in Natural;
     MaxNumParameter : in Natural)
    return Boolean;
```

Description

Check the number of actual parameters on the right-hand side to be in the range *MinNumParameter* : *MaxNumParameter*. Return `True` if it is in the range, otherwise raise error 77 associated with *SelfName* and return `False`.

Example

Ensure that at least 2, but not more than 5 parameters are passed to the function:

```
if (.not. checkrhs(fname, 2, 5)) return
```

We have assumed that *fname* is set to the function's name.

See also

`CheckLhs`, `Lhs`, `Rhs`

7.5.1.2. checklhs

Synopsis

```
function CheckLhs
  (SelfName      : in String;
   MinNumParameter : in Natural;
   MaxNumParameter : in Natural)
  return Boolean;
```

Description

Check the number of output variables, i.e. arguments on the right-hand side to be in the range *MinNumParameter* : *MaxNumParameter*. Return `True` if it is in the range, otherwise raise error 78 associated with *SelfName* and return `False`.

Note that it is no error to supply less output parameters than the function actually returns. The extra values are silently discarded. This is true for the case of zero output values, too; then `ans` gets the first output value. Thus, a function called without any output parameters is assigned an `Lhs` of 1.

Example

Ensure that there are not more than 2 output parameters, when the function is called:

```
if (.not. checklhs(fname, 1, 2)) return
```

We have assumed that `fname` is set to the function's name.

See also

`CheckRhs`, `Rhs`, `Lhs`

7.5.1.3. `lhs`

Synopsis

```
Lhs : Natural;
```

Description

The number of actual output parameters, i.e. those on the left-hand side of the assignment operator, is stored in the global variable `Lhs`. Inside a user function, `Lhs` should be used as a constant.

See also

`CheckLhs`, `CheckRhs`, `Rhs`

7.5.1.4. `rhs`

Synopsis

```
Rhs : Natural;
```

Description

The number of actual input parameters, i.e. those on the right-hand side of the assignment operator, is stored in the global variable `Rhs`. Inside a user function, `Lhs` should be used as a constant.

See also

`CheckRhs`, `CheckLhs`, `Lhs`

7.5.2. Access Object

The functions in this section grant the programmer access to parameters that are stored on the Scilab stack. In general all of these functions work alike: An index to the current (i.e. as on entry of the function) top of the parameter stack, “BasePointer”, and an index to the desired argument, “StackPointer”, are passed to the API. On return the user gets all necessary information about the argument like sub-type, dimension as well as the indices, “FooIndex” that index into the Scilab heap. The indices act like pointers to the actual contents. This way only meta-data is passed, saving time-consuming copy operations.

7.5.2.1. getmat

Synopsis

```
function GetMat
    (SelfName      : in      String;
     BasePointer   : in      ParameterStackAddress;
     StackPointer  : in      ParameterStackAddress;
     IsComplex     : out     ComplexFlag;
     Rows          : out     Natural;
     Columns       : out     Natural;
     RealIndex     : out     DataStackIndex;
     ImaginaryIndex : out     DataStackIndex)
return Boolean;
```

Description

Retrieve the address(es) and dimensions of a real or complex matrix from the parameter stack. The *BasePointer* must be set to the parameter stack pointer’s value on entry of the *calling* function. *StackPointer* points to the desired parameter on the parameter stack. If successful, GetMat returns True, and *IsComplex*, *Rows*, *Columns*, and *RealIndex* are valid. If *IsComplex* = ComplexVariable then *ImaginaryIndex* is valid, too. If the parameter indexed by *StackPointer* is not a matrix GetMat returns False.

The output parameter *IsComplex* indicates whether the matrix on the data stack is purely real or complex. In the first case *RealIndex* points to the matrix, in the second case *RealIndex* points to the real part of matrix, and *ImaginaryIndex* points to the imaginary part. In any case *Rows* and *Columns* are the number of rows and columns of the matrix.

Example

Fetch the addresses of a possibly complex m-times-n matrix from position `top` of the parameter stack.

```

        if (.not. getmat(fname, topk, top, is-
cmpx, m, n, are, aim)) return

```

It is assumed that `fname` has been set to the function's name, and `topk` carries the position of the stack on entry to the calling function.

See also

`GetRMat`, `GetRVect`, `GetVect`

7.5.2.2. `getrmat`

Synopsis

```

function GetRMat
  (SelfName      : in      String;
   BasePointer   : in      ParameterStackAddress;
   StackPointer  : in      ParameterStackAddress;
   Rows          :         out Natural;
   Columns       :         out Natural;
   RealIndex     :         out DataStackIndex)
return Boolean;

```

Description

Function `GetRMat` works like function `GetMat`, but restricts the accepted matrices to purely real ones.

See also

`GetMat`, `GetRVect`

7.5.2.3. `getrvect`

Synopsis

```

function GetRVect
  (SelfName      : in      String;
   BasePointer   : in      ParameterStackAddress;
   StackPointer  : in      ParameterStackAddress;
   Rows          :         out Natural;
   Columns       :         out Natural;

```

```

    RealIndex      :      out DataStackIndex)
return Boolean;

```

Description

Function `GetRVect` works like function `GetRMat`, but restricts the accepted matrices to either single rowed (1-times- N) or single columned (N -times-1).

See also

`GetVect`, `GetRMat`

7.5.2.4. `getvect`

Synopsis

```

function GetVect
(SelfName      : in      String;
 BasePointer   : in      ParameterStackAddress;
 StackPointer  : in      ParameterStackAddress;
 IsComplex     :      out ComplexFlag;
 Rows          :      out Natural;
 Columns       :      out Natural;
 RealIndex     :      out DataStackIndex;
 ImaginaryIndex :      out DataStackIndex)
return Boolean;

```

Description

Function `GetVect` works like function `GetMat`, but restricts the accepted matrices to either single rowed (1-times- N) or single columned (N -times-1).

See also

`GetMat`, `GetRVect`

7.5.2.5. `getscalar`

Synopsis

```

function GetScalar
(SelfName      : in      String;
 BasePointer   : in      ParameterStackAddress;
 StackPointer  : in      ParameterStackAddress;

```

```

    Index      :      out DataStackIndex)
return Boolean;

```

Description

Retrieve the address and dimensions of a real or complex scalar from the parameter stack. The *BasePointer* must be set to the parameter stack pointer's value on entry of the *calling* function. *StackPointer* points to the desired parameter on the parameter stack. If successful, *GetScalar* returns *True*, and *Index* is valid. If the parameter indexed by *StackPointer* is not a scalar *GetScalar* returns *False*.

See also

GetVect, *GetMat*

7.5.2.6. getexternal

Synopsis

```

type FortranIdentifier is
    array (1 .. 6) of Character;  - Fortran's 6 char limit

- SimpleFunctionType is just an example
type SimpleFunctionType is access
    function(X : in Float) return Float;

type InstallerProcedureType is access
    procedure(FunctionName      : in      FortranIdentifier;
               FunctionEntryPoint : in      SimpleFunctionType);

function GetExternal
    (SelfName      : in      String;
     BasePointer   : in      ParameterStackAddress;
     StackPointer  : in      ParameterStackAddress;
     FunctionName  : in out FortranIdentifier;
     IsExternal    :      out Boolean;
     Installer     : in      InstallerProcedureType)
return Boolean;

```

Description

The first three parameters *SelfName*, *BasePointer*, and *StackPointer* work exactly the same as in the other functions, so does the return value. Their explanations are not repeated here, see e.g. Section 7.5.2.1.

The fourth parameter, *FunctionName* is the name of the function to be called. This parameter can designate an external function, i.e. code that has been compiled separately and then linked to Scilab via, for example `link`, or the name of a Scilab function that has been defined with `function` or `deff`. In any case it is simply the name of the function. On return the *IsExternal* parameter signals `True` if the function is a external and `False` otherwise.

The last parameter is very special. It specifies the installation procedure *Installer* that manipulates a dispatcher function *DispatchFunction*. After a call to *Installer* the dispatcher points to the user function given by *FunctionName*.

Note: The programmer *should not* issue such a call; it is already done by `GetExternal`.

Examples of dispatcher functions, the associated hooks and dispatch tables are e.g. found in `SCI/routines/default/FTables.{h,c}`. We will discuss dispatch tables in Section 7.3.3.1.

Support Functions

The `GetExternal` function relies heavily on various support functions. The supporting function have to be set up previous to the first call. Usually they are installed by the user's extension package unless she/he decides to (ab)use existing dispatcher tables and functions.

```
package ExternalSupport is
  procedure InstallProcedure
    (FunctionName      : in      FortranIdentifier;
     FunctionEntryPoint : in      SimpleFunctionType);
end ExternalSupport;

package body ExternalSupport is

  with Ada.Characters.Latin_1;

  type FunctionTableEntry is
    record
      FunctionName      : FortranIdentifier;
      FunctionAddress   : SimpleFunctionType;
    end record;

  type FunctionTableType is
    array (Positive range <>) of FunctionTableEntry;

  - SimpleExample is of type SimpleFunction
  function SimpleExample(X : Float) return Float;
begin
  return X;
```

```

end Hook;

FunctionTable : FunctionTableType(1 .. 2) :=
  (1 => (FunctionName    => "exampl",
        FunctionAddress => SimpleExample'Access),
   2 => (FunctionName    => (oth-
ers => Ada.Characters.Latin_1.NUL),
        FunctionAddress => null));

DispatchFunction : SimpleFunctionType;

- function Hook is the hard-coded target for all
- internal calls

function Hook(X : Float) return Float;
begin
  return DispatchFunction.all(X);
end Hook;

- bend hook function to point to FunctionEntryPoint

procedure InstallProcedure
  (FunctionName      : in      FortranIdentifier;
   FunctionEntryPoint : in      SimpleFunctionType);
begin
  DispatchFunction := SetFunction(FunctionName,
                                  FunctionEntryPoint,
                                  FunctionTable);
end InstallProcedure;

end ExternalSupport;

```

Example

For a self-contained example, see Section 7.3.2.

See also

Functionals, Dispatch Tables

7.5.3. Create Object

The object creation functions are mainly used to setup temporary variables for the current procedure or the procedures to be called; they bear a lot of resemblance with the object access functions (see also Section 7.5.2). The difference is that a new object is created and therefore stack space is used.

7.5.3.1. Cremat

Synopsis

```
function Cremat
    (SelfName      : in      String;
     StackPointer  : in      ParameterStackAddress;
     WantComplex   : in      ComplexFlag;
     Rows          : in      Natural;
     Columns       : in      Natural;
     RealIndex     :          out DataStackIndex;
     ImaginaryIndex :          out DataStackIndex)
    return Boolean;
```

Description

FIXME: write it

7.5.4. Miscellaneous

FIXME: Write it!

7.6. C Interface to Scilab's Core

Analogously to the Fortran-77 section, Section 7.5, following provides a reference for the C-interface to Scilab split into four sub-sections: query, access, creation of objects, and miscellaneous functions.

All C-interface functions are introduced in `SCI/routines/stack-c.h`.

7.6.1. Query

The functions in this group retrieve information about the parameters a function has been called with, and about the properties of objects on the stack.

7.6.1.1. CheckRhs

Synopsis

```
function CheckRhs
  (MinNumParameter : in Natural;
   MaxNumParameter : in Natural)
  return Integer;
```

Description

Check the number of actual parameters on the right-hand side to be in the range *MinNumParameter* : *MaxNumParameter*. Return 1 if it is in the range, otherwise raise error 77 associated with the name of the C-function from which *CheckRhs* is called.

The semantics of *CheckRhs* is slightly goofy. If the number of actual input parameters is in the specified range, *CheckRhs* returns 1, but it never returns 0 as it raises an error in this case. Therefore, the return value can safely be ignored, as we do in the Example.

Example

```
int
myfun(const char *fname)
{
    /* local variables */

    CheckRhs(1, 1);                                     – check for exactly one argument

    /* more code goes here */
}
```

See also

CheckLhs, Lhs, Rhs

7.6.1.2. CheckLhs

Synopsis

```
function CheckLhs
  (MinNumParameter : in Natural;
   MaxNumParameter : in Natural)
  return Integer;
```

Description

Check the number of actual parameters on the left-hand side to be in the range *MinNumParameter* : *MaxNumParameter*. Return 1 if it is in the range, otherwise raise error 78 associated with the name of the C-function from which `CheckLhs` is called.

The semantics of `CheckLhs` is slightly goofy. If the number of actual output parameters is in the specified range, `CheckLhs` returns 1, but it never returns 0 as it raises an error in this case. Therefore, the return value can safely be ignored, as we do in the Example.

Note that a seemingly empty left-hand side as in

```
->ones(2,3)
ans =
!  1.    1.    1. !
!  1.    1.    1. !
```

implies the variable `ans`, and accordingly the number of left-hand side parameters in this case is 1.

Example

```
int
myfun(const char *fname)
{
    /* local variables */

    CheckLhs(1, 1);
    allow a maximum of one return value

    /* more code goes here */
}
```

See also

`CheckRhs`, `Lhs`, `Rhs`

7.6.1.3. Lhs

Synopsis

```
function Lhs : Natural;
```

Description

The number of actual output parameters, i.e. those on the left-hand side of the assignment operator, is available through `Lhs`.

Note that a seemingly empty left-hand side as in

```
->ones(2,3)
ans =
!  1.    1.    1.  !
!  1.    1.    1.  !
```

implies the variable `ans`, and accordingly the number of left-hand side parameters, `Lhs`, in this case is 1.

See also

`CheckLhs`, `CheckRhs`, `Rhs`

7.6.1.4. Rhs**Synopsis**

```
function Rhs : Natural;
```

Description

The number of actual input parameters, i.e. those on the right-hand side of the assignment operator, is available through `Rhs`.

See also

`CheckRhs`, `CheckLhs`, `Lhs`

7.6.1.5. GetType**Synopsis**

```
function GetType
  (ParameterNumber : in ParameterStackIndex)
  return TypeCode;
```

Description

Inquire the Scilab type code of argument *ParameterNumber*. The type codes are the same as returned by `type`. The first parameter has index 1. See Table 4-3 for a complete listing of all available type codes.

Example

```
int
vandermonde(const char *fname)
{
    /* local variables */

    CheckRhs(1, 1);                                – assert one argument
    if (GetType(1) != 1) {                            – test for a float
        Scierror(814, "%s: expecting floating point entity", fname);
        return 1;
    }

    /* more code goes here */
}
```

See also

GetRhsVar

7.6.2. Access Object

The following functions allow for the access of the Scilab data stack, and the mapping of C-pointers to and from stack indices.

Please remember that variables of type `ParameterStackAddress`, within the user function, point to *unique* stack positions. Thus, after the calls

```
GetRhsVar(1, "d", &rows1, &cols1, &idx1);
CreateVar(1, "d", &rows2, &cols2, &idx2); /* Ouch! */
```

`stk(idx1)` and `stk(idx2)` point to the same memory location. Sometimes – if and only if the “old” variable is not accessed anymore – this is desired.

The type of a variable on the stack, whether it is the actual type of an argument passed in, or the requested type of a local variable, is defined with a string of length 1, the `TypeString`. All valid type strings are compiled in Table 7-4.

Table 7-4. TypeString Identifiers

TypeString	C or Scilab type
S	string
c	single precision complex, struct complex { float re, im; } (see also Section 7.2.2)
d	double
f	float
i	int
l	list
p	void*
r	reference?
t	tlist
z	double precision complex, struct double_complex { double re, im; } (see also Section 7.2.2)

7.6.2.1. GetRhsVar

Synopsis

```

procedure GetRhsVar
  (ParameterNumber : in    ParameterStackIndex;
   VariableType    : in    TypeString;
   Rows            :      out AccessNatural;
   Columns         :      out AccessNatural;
   StackIntPtr     :      out AccessDataStackIndex);

```

Description

Return size and Scilab-stack address of function argument number *ParameterNumber*. The argument must have type *VariableType*. See Table 7-4 for valid type-strings.

On success *Rows* and *Columns* hold the dimensions of the parameter (twice a one for a scalar) and *StackIntPtr* points to the index of start address of the parameter on the Scilab stack *stk*.

GetRhsVar performs weak type checking with respect to *VariableType*. If *VariableType* = "d" and a (double) complex is passed to the function, *no* error is raised! Integers, strings, etc. do raise an error in this case. Therefore, the type of an argument always should be inquired with GetType before calling GetRhsVar.

Example

```

int
vandermonde(const char *fname)
{
    int rows, cols, vec_idx;
    double *input_vec;

    /* more code */

    GetRhsVar(1,                                - get first argument
              "d",                               - expect double
              &rows,                             - number of rows
              &cols,                             - number of columns
              &vec_idx);                        - stack index
    if (rows != 1 && cols != 1) {
        Scierror(815, "%s: expecting vector", fname);
        return 1;
    }
    input_vec = stk(vec_idx);                    - convert index to pointer

    /* more code */
}

```

See also

GetType, GetMatrixptr, LhsVar

7.6.2.2. GetMatrixptr

Synopsis

```

procedure GetMatrixptr
(
);

```

Description

FIXME: Write it!

Example

...

See also

...

7.6.2.3. GetMatrixDims

Synopsis

```
procedure GetMatrixDims
(
);
```

Description

FIXME: Write it!

Example

...

See also

...

7.6.2.4. GetRhsCVar

Synopsis

```
procedure GetRhsCVar
(
);
```

Description

FIXME: Write it!

Example

...

See also

...

7.6.2.5. GetListRhsVar

Synopsis

```
procedure GetListRhsVar
(
) ;
```

Description

FIXME: Write it!

Example

...

See also

...

7.6.2.6. GetListRhsCVar

Synopsis

```
procedure GetListRhsCVar
(
) ;
```

Description

FIXME: Write it!

Example

...

See also

...

7.6.2.7. GetFuncPtr

Synopsis

```

procedure GetFuncPtr
(
);

```

Description

FIXME: Write it!

Example

...

See also

...

7.6.2.8. LhsVar

Synopsis

```
LhsVar : array (ParameterStackIndex) of ParameterStackIndex;
```

Description

LhsVar is an array of the return values of a function. Assigning to LhsVar (N) means assigning to return-value number N . The values put into LhsVar are the numbers of local variables, which have previously been created with CreateVar or GetRhsVar.

Example

```

int
vandermonde(const char *fname)
{
    int rows, cols, vec_idx;
    int n, vdm_idx;
    double *input_vec;
    double *vdm_matrix;

    /* some code here */

    GetRhsVar(1,
              "d",
              &rows,
              &cols,
              &vec_idx);
    input_vec = stk(vec_idx);

```

– argument 1 at stack position 1

```

n = (rows > cols) ? rows : cols;

/* allocate matrix */
CreateVar(2,                                     – local variable 1 at stack position 2
          "d",
          &n,
          &n,
          &vdm_idx);

/* compute Vandermonde matrix */

LhsVar(1) = 2;                                   – stack position 2 goes to return-
value 1
return 0;
}

```

The line `LhsVar(1) = 2;` requires further explanation. The left-hand side of the assignment specifies the first element in the array of all return values of the function. The right hand side of the assignment denotes the first local variable (created with `CreateVar` in the example) within the function. It has stack index 2, because this is the first free stack position after all parameters (here: 1).

See also

`CreateVar`, `CheckLhs Lhs`

7.6.3. Create Object

The functions in the create group of the core interface allocate new Scilab variables on the Scilab data stack, which are accessed through their index on the parameter stack.

7.6.3.1. CreateVar

Synopsis

```

procedure CreateVar
  (VariableNumber : in      ParameterStackIndex;
   VariableType   : in      TypeString;
   Rows           : in      AccessNatural;
   Cols           : in      AccessNatural;
   StackIndex     :         out AccessDataStackIndex);

```

Description

Create a new local Scilab variable on the Scilab stack. The variable is later accessed with its “handle”, *VariableNumber*. The type of the variable is selected with *VariableType*. See Table 7-4 for a complete listing of all type strings. The size of the new scalar, vector, or matrix is determined by *Rows*, and *Cols*. Scalars have *Rows* := 1; *Cols* := 1.

On return the *CreateVar* sets *StackIndex* to the element in the data stack *stk* that points to the start address of the newly created variable.

Example

```
int
vandermonde(const char *fname)
{
    int vdm_idx;
    const int n = 4;
    double *vdm_matrix;

    /* some code here */

    CreateVar(2,                                – first local variable
              "d",                              – double precision
              &n,                                – n rows
              &n,                                – n columns
              &vdm_idx);                        – index into stack
    vdm_matrix = stk(vdm_idx);                   – convert to pointer

    /* more code here */
}
```

See also

LhsVar

7.6.3.2. CreateVarFromPtr

Synopsis

```
procedure CreateVarFromPtr
(
    );
```

Description

FIXME: Write it!

Example

...

See also

FreePtr

7.6.3.3. FreePtr

Synopsis

```
procedure FreePtr
(
);
```

Description

FIXME: Write it!

Example

...

See also

CreateVarFromPtr

7.6.3.4. CreateCVar

Synopsis

```
procedure CreateCVar
(
);
```

Description

FIXME: Write it!

See also

...

7.6.3.5. CreateCVarFromPtr

Synopsis

```
procedure CreateCVarFromPtr
(
);
```

Description

FIXME: Write it!

See also

...

7.6.3.6. CreateData

Synopsis

```
procedure CreateData
(
);
```

Description

FIXME: Write it!

See also

...

7.6.3.7. CreateListCVarFromPtr

Synopsis

```
procedure CreateListCVarFromPtr
(
```

```
) ;
```

Description

FIXME: Write it!

See also

...

7.6.3.8. CreateListVarFromPtr

Synopsis

```
procedure CreateListVarFromPtr
(
) ;
```

Description

FIXME: Write it!

See also

...

7.6.3.9. Createlist

Synopsis

```
procedure Createlist
(
) ;
```

Description

FIXME: Write it!

See also

...

7.6.3.10. WriteMatrix

Synopsis

```

procedure WriteMatrix
  (ScilabVariableName : in String;
   Rows : in ConstAccessNatural;
   Cols : in ConstAccessNatural;
   Carray : in ConstAccessAny);

```

Description

FIXME: Write it!

send array C[] to Scilab as variable C; see: intex14c.c

Example

...

See also

...

7.6.3.11. WriteString

Synopsis

```

procedure WriteString
  (ScilabVariableName : in ConstAccessString;
   CStringLength : in ConstAccessNatural;
   CString : in ConstAccessString);

```

Description

FIXME: Write it!

create the Scilab variable Str from str; see: intex16c.c

Example

...

See also

...

7.6.4. Miscellaneous

The dreaded miscellaneous... All functions that do not fit in any of the above categories go here.

7.6.4.1. `scierror`

Synopsis

```
procedure Scierror
    (ErrorNumber : Natural;
     FormatString : String;
     FunctionName : String);
```

Description

Raise error *ErrorNumber*, associated with the errors message *FormatString*, which is preceeded by the function's (the one we are currently in) name *FunctionName*.

Example

```
int
hrtimer(const char *fname)
{
    /* code left out here */

    if (t < 0.0) {
        Scierror(5771, "%s: internal error\n", fname);
        return 1;
    }

    /* code left out here */
}
```

7.6.4.2. `PExecSciFunction`

Synopsis

```

procedure PExecSciFunction
(
);

```

Description

FIXME: Write it!

From the example file: Executes the Scilab function (f) pointed to by sci_f. We provide a rhs = 2 and expect lhs = 1; PExecSciFunction(5, &sci_f, &lhs, &rhs, "ArgFex", ex17cenv);

Example

See also

intex17c.c ...

7.6.4.3. ReadString

Synopsis

```

procedure ReadString
(Identifier      : in      ConstAccessString;
 CStringLength  : in out  AccessNatural;
 CString        :      out AccessString);

```

Description

Form the example file: We search a Scilab object named Mystr check that it is a string and store the string in str. strl is used on entry to give the maximum number of characters which can be stored in str. After the call strl contains the number of copied characters.

Example

See also

intex15c.c ...

7.6.4.4. SciFunction

Synopsis

```
procedure SciFunction
(
);
```

Description

FIXME: Write it!

execute the function; SciFunction(&ibegin, &lf, &mlhs, &mrhs);

Example

See also

intex8c.c ...

7.6.4.5. SciString

Synopsis

```
procedure SciString
(
);
```

Description

FIXME: Write it!

eval()? exec()? SciString(&ibegin, name, &mlhs, &mrhs);

Example

See also

intex7.c.c, intex11c.c, intex12c.c ...

Chapter 8. Further Information

We are done with our tour through Scilab, but we are not done with sci-BOT! A few things outside the Scilab application have to be mentioned. First, the sheer size of the sources requires some tools to handle it efficiently. We address this topic in Section 8.1. Moreover, we should not forget that Scilab is shipped with a lot of helpful documentation. Section 8.2 gives an overview of this part of the documentation. We wrap up the chapter with Section 8.3, a small collection of hyper links connected to Scilab

8.1. Coping With Scilab

Scilab is a large package – no doubt about that. The source for version 2.5 comprises of more than 48 MB, and builds to over 88 MB on an IA32 GNU/Linux system.

8.1.1. Distribution Size

We use several tools to cope with Scilab’s size and complexity. The most important ones are introduced in the following section.

8.1.1.1. CVS

CVS is one of the most commonly used version control systems. A set of source files (which can be binary) is put under revision control by “checking it in”. The important difference to an older version control system, RCS is the notion of a module which refers to the complete set of sources. Usually the set consists of a whole directory tree, as e.g. all Scilab sources.

Also check out Pascal Molli’s *information and FAQ* on CVS. In larger development environments CVS with its relaxed rules might not be the adequate tool. In these cases *Aegis* could be used.

8.1.1.2. locate

The **locate**(1) command is the fast brother of the **find**(1) command. More precisely, **locate** accesses a precomputed database of filenames (usually `/var/lib/locatedb`; for its structure see `locatedb(5)`). The database is generated by **updatedb**(1) with a `find / -print` and then processed for faster access.

We have found a local filename database very useful for the work with large projects. Therefore, we have set up two aliases that create and access a project-specific list of filenames.

```
alias upd='updatedb -output=./.locatedb -localpaths=.'
alias loc='locate -database=./.locatedb'
```

The **upd** sequence is typically run after a CVS checkout, add, or remove in the directory SCI.

We run **loc** whenever we are looking for a file in the Scilab distribution. This is much faster than running **find** every time, especially when working with a slow file server. The only inconvenience remaining is that **loc** must be executed in the directory where the database resides, here: SCI. However this is more than compensated by the fact that **locate** does a substring search, i.e. given the filename *fpat* it returns all file- and directory names matching the regular expression **fpat.**.

If we want to scan the complete database and postprocess the output with our tools-of-choice, issuing a `loc .` and piping the output through the desired filters does the job.

8.1.1.3. Glimpse

What the **updatedb/locate** pair is for filenames the **glimpseindex/glimpse** pair is for file contents. **glimpseindex(1)** generates a database that is accessed by the user via **glimpse(1)**. So,

```
glimpse pattern
```

corresponds to the non-database backed command, namely a recursive **grep** over a set of directories like

```
find . -print | xargs grep pattern
```

assuming that the database has been generated for “.”. Again the fast version is so helpful that we have defined two aliases.

```
alias glidx='if test -f .glimpse_index; then
    glimpseindex -H . -o -f -B .;
else
    glimpseindex -H . -o -B .;
fi'
alias gl='glimpse -H .'
```

The first alias, **glidx**, is *one* line. It has been broken into several lines only to make its workings clear; namely if an index file already exists it is updated (`-f` option), otherwise it is generated from scratch.

Like our **locate** aliases everything is happening in the current directory which means that **glidx** should be called from SCI.

Glimpse is not part of most of the standard GNU/Linux distributions (at least not *SuSE*, and *RedHat*, the ones we checked). The University of Arizona currently hosts the *Glimpse home page*, and Glimpse can also be downloaded from *SCO's software archive*, which is mirrored by *Sunsite UK*.

8.1.2. Bug Hunting

In preparation of this document (lvd), and in our daily work (cls) we have found it very useful to have more than one Scilab. What? More than one running process? – No, more than one binary of **scilex**! In fact three different versions all come in handy depending on the task:

scilex binaries

Code optimized for execution speed

The common name is “production quality code”, but Scilab is so far away from production quality that we shall not use that term.

This `scilex` is built with all compiler optimizations enabled. Furthermore all compiler switches and options are specifically tuned for the machine the code will run on in the future (see Section 6.3). Maximum performance is the only goal and no attempt is made to retain any debugging information.

Debugging Code

This `scilex` is not optimized, instead it carries complete debug information. Thus, it is ideally suited for interactive debugging sessions, and single-line tracing.

Profiling Code

The third variant is a profiling version of `scilex`. It is not optimized for speed either.

Profiling is the first step of any tuning. Furthermore, during our work with and on Scilab we have found it very helpful to be able to answer the notorious question: “Where is it burning the cycles?” Profiling – done right – is much faster than timing individual “suspects”, although analyzing the profiler output requires some skill.

See also Section 4.6.2.

8.2. Local Documents

Following documents come with every source distribution of Scilab. They live in the directory `SCI/doc`.

Standard Documentation

`Comm.ps` – Communication Toolbox

Description of `geci` an interactive communication manager.

Whenever Scilab is started with the **scilab**-script in fact `geci` takes over and starts **scilex** as a process on the local machine. `geci` is not limited to local processes; remote machines can be accessed transparently.

`Comm.ps` describes the commands available from within Scilab to communicate with other applications via `geci`. Moreover it elaborates how to write C-applications that communicate with Scilab via `geci` and the associated library.

`Internals.ps` – Guide for Developers

`Internals.ps` is the terse breakdown of the innermost core of `scilex`. It contains descriptions of the most holy variables like `stk`, the stack structure, and the internal variable representation.

The last third treats interfacing user routines with the core and consists mostly of scarcely documented Fortran program listings.

This is a document intended for gurus, and certainly not suited for casual reading.

`Intro.ps` – Users Guide

This is not a must-read this is more, it is a must-print-and-store-near-the-computer. The “Intro” is *much* more than just an introduction to Scilab.

The chapter breakdown is as follows:

- Introduction,
- Data Types,
- Programming,
- Basic Primitives,
- Graphics, and
- Interfacing.

The appendices cover a Demo Session, System Interconnection, and a brief section about converting Scilab code to Fortran-77 code.

Most interesting for beginners is the chapter “Introduction”. Combining it with browsing over “Data Types” and “Graphics” a novice should be all set for her/his first steps.

The advanced user will want to come back the “Data Types” regularly and also study “Programming” in detail. Any chapter but “Interfacing” should be understandable at that skill level. It is no shame to come back to the Intro over and over again as Scilab is rich in data types and graphics commands.

Far beyond the introductory stuff the chapter on “Interfacing C or Fortran routines” wraps up the Intro. The topics treated here are for the aspiring Scilab master. Dynamic and static extensions are discussed in depth. The companion program `intersci` for automatic Fortran77-interface generation is treated in detail.

`Lmi.ps` – LMI-Optimization Toolbox

FIXME: someone please write it!

`Manual.ps` – Reference Manual

The `Manual.ps` is an automatically generated compilation of all Scilab user-variables and user-commands. It is a compilation in the truest sense of the word as all the help texts available online through the **help**-command are catenated to one huge (over 700 pages) file.

`Metanet.ps` – Graphs and Networks Toolbox

FIXME: someone please write it!

`Scicos.ps` – Dynamic System Builder And Simulator Toolbox

FIXME: someone please write it!

`Signal.ps`– Signal Processing

FIXME: someone please write it!

8.3. Hyperlinks

Here are a few links that are useful in connection with Scilab.

Links

INRIA official Scilab pages

- *Scilab Home page*
- *Parallel Scilab Home page*
- *Scilab FAQ*
- *Scilab FTP Site*

Pages Of Scilab Enthusiasts (alphabetically)

- *Stéphane Mottelet's Scilab page*
- *Jesus Oliván's Scilab page* focuses on signal processing in medicine.
- *Bruno Pinçon's Scilab page* featuring a nice French introduction to Scilab.
- *Enrico Segre's page*

- *Alexander Vigoder's Scilab page* (mainly scilab-mode for emacs)

Chapter 9. Notes for Contributors

“You’re the voice” by John Farnham

We have the chance, to turn the pages over
We can write what we want to write
We got to make ends meet before we get much older

These notes should help contributors to adapt their writing to the format fo sci-BOT. The first part, Section 9.1, treats stylistic problems, the second, Section 9.2, technical matter.

9.1. Writing Style

We do not require a contributor to follow Strunk and White’s, “The Elements of Style” [Strunk:1979], though it is not a bad idea for any author to study this classical book on writing well. Instead here is some general and simple advice for writing. The following list is an edited excerpt from the author instructions of the American Institute of Physics.

Be clear

Consider the beauty and efficiency of simple declarative sentences as the perfect medium for communicating complex information. Avoid long sentences in which the meaning may be obscured by complicated or unclear construction.

Be concise

Avoid vague and inexact usage. Be as qualitative as the subject matter permits. Avoid idle words; make every word count.

Be complete

Do not assume that your reader has all the background information that you have on the subject matter. Make sure your argument is complete, logical, and continuous.

Put yourself constantly in the place of the reader

Be rigorously self-critical as you review your first drafts, and ask yourself: “Is there any way in which this passage could be misunderstood by someone reading it for the first time?”

Some very good hints come from an article of George D. Gopen and Judith A. Swan [Gopen:1990]:

1. Follow a grammatical subject as soon as possible with its verb.
2. Place in the “stress position” the new information you want the reader to emphasize.

3. Place the person or thing whose “story” a sentence is telling at the beginning of the sentence, in the topic position.
4. Place appropriate “old information” (material already stated in the discourse) in the topic position for linkage backward and contextualization forward.
5. Articulate the action of every clause or sentence in its verb.
6. In general, provide context for your reader before asking that reader to consider anything new.
7. In general, try to ensure that the relative emphases of the substance coincide with the relative expectations for emphasis raised by the structure.

Always bear in mind that sci-BOT is a DocBook document, which means that

- i. it is stored in machine readable format,
- ii. its HTML output is viewed online with a multitude of different browsers, and
- iii. its PostScript® or PDF output will be printed on paper with black-and-white printers.

Item i enables the author to make extensive use of cross-references (`<xref>`), preferably in both directions. Furthermore, authors should make heavy use of the automatic index generation (`<indexentry>`). `<xref>`s can go everywhere in the running text, but `<indexentry>`s should only go into `chapter` or `sectionN` elements even if this separates the indexed term from the index entry.

The different output media, as mentioned in item ii and iii, require that all included graphics are provided in at least two formats, one suitable for online viewing and the other for high resolution (assume at least 720 dpi) monochrome printouts.

FIXMEs. Probably has seen several almost empty sections marked with “FIXME” and some comment following it. These markers are a *good* thing. They remind the author[s] that the discussion is still incomplete, but the missing part has been identified and sits already in the right section. On the other hand, a “FIXME” tells the reader that the author is aware of a gap in the flow of information, and is probably working on it.

9.2. Technicalities

9.2.1. DocBook

sci-BOT is written in xml-DocBook. Any aspiring author should get a copy of Norman Walsh and Leonard Mueller’s, “DocBook: The Definitive Guide”. See, [Walsh:1999] for full bibliographical details. An online version of the book is available at DocBook-Online, and an archive of DocBook-Online is provided, too.

Furthermore, we recommend Nik Clayton’s “FreeBSD Documentation Project Primer for New Contributors”. It gives a gentle introduction to the whole business of using DocBook, and moreover, elaborates on semantical and stylistic issues.

9.2.1.1. Guidelines

- The preferred text width is 79 characters.
- Tabulator characters are forbidden; always use spaces.
- CDATA sections are forbidden.
- The use of double quotes in `para` or similar elements is deprecated; use `quote` or `blockquote` instead.
- All `id` attributes are made of the characters a-z (lowercase only), 0-9, and the dash `-`.
- For the following elements identifiers are mandatory: `chapter`, `sect1`, `sect2`, ... `sect6`, `indexterm`, `example`, `figure`, `biblioentry`, and `co`.
- Identifiers are made up of lowercase alphanumeric characters and dashes. If an element carries an identifier, the identifier must have a prefix. The prefix is separated from the rest of the identifier with a dash.

tag	prefix
<code><chapter></code>	chap
<code><sect1></code> , <code><sect2></code> , ..., <code><sect6></code>	sect
<code><titleabbrev></code>	chsh or sesh for <code><chapter></code> or <code><sect>N</code> . In general: the first two letters from the prefix plus "sh".
<code><indexterm></code>	idx
<code><biblioentry></code>	bib
<code><equation></code> , <code><informalequation></code>	eq
<code><table></code> , <code><informaltable></code>	tab
<code><figure></code> , <code><informalfigure></code>	fig
<code><example></code>	ex
<code><co></code>	co
<code><listitem></code>	item

9.2.1.2. Indentation style

Our indentation style mostly follows the common SGML/XML DocBook indentation, with few exceptions:

- The principal indentation is 4 characters, not 2.
- Sectioning tags like `<chapter>`, `<sect1>`, `<sect2>`, ..., or `<sect6>`, do not increase the indentation level; they always start at column 1;

- `<para>` is never formatted inline, but as block with the appropriate indentation.
- `<title>`, `<entry>`, and `<term>` change their inlining character depending on the length of the contained text. As long as the tag and the contained text fit in one line, the tags are formatted inline, otherwise they go on lines by themselves and the contents is indented.

9.2.2. Tables

We prefer tables to be formatted according to DIN 55301. Some formatting systems might not be able to produce a fully compliant output, as well as some output formats are incapable of representing a correctly formatted table. Therefore, we do not only show the SGML-source code, Example 9-1, to mark up a table, followed by the rendered table, Table 9-1, but also a correctly rendered one, Figure 9-1, which was produced by LaTeX.

The DIN 55301 takes about ten pages, explaining the construction of a table in detail; most important for sci-BOT are the following rules:

- Top and bottom of the table header are framed, but never are the header's left or right sides.
- Logical lines in the table header are separated by rules.
- In the header different columns within one logical line are separated with vertical rules. The width of a column must be less or equal the width of the column above it.
- Entries in the header are centered horizontally *and* vertically.
- The table body lacks *any* rule.

Example 9-1. SGML-code for a DIN compliant table

```
<table id = "tab-table-din55301" frame = "top">
  <title>DocBook approximation of a DIN conforming table</title>

  <tgroup cols = "12">
    <colspec colname = "col-pid" align = "right"></colspec>
    <colspec colname = "col-usr" align = "left"></colspec>
    ...
    <colspec colname = "col-cmd" align = "left"></colspec>

    <spanspec spanname = "sp-id" namest = "col-pid" nameend = "col-
usr"
                align = "center" colsep = "1"></spanspec>
    <spanspec spanname = "sp-sched" namest = "col-pri" nameend = "col-
ni"
                align = "center" colsep = "1"></spanspec>
    ...

  <thead>
```

```
<row>
  <entry spanname = "sp-id">Identification</entry>
  <entry spanname = "sp-sched">Scheduling</entry>
  <entry spanname = "sp-mem">Memory</entry>
  <entry align = "center" colsep = "1"
    morerows = "1" valign = "middle">Stat</entry>
  <entry spanname = "sp-res">Resources</entry>
  <entry align = "center" colsep = "1"
    morerows = "1" valign = "middle">Command</entry>
</row>

<row>
  <entry align = "center" colsep = "1">PID</entry>
  <entry align = "center" colsep = "1">User</entry>
  ...
  <entry align = "center" colsep = "1">Share</entry>
  <!-- space occupied by "Stat" -->
  <entry align = "center" colsep = "1">%CPU</entry>
  <entry align = "center" colsep = "1">%Mem</entry>
  <entry align = "center" colsep = "1">Time</entry>
  <!-- space occupied by "Command" -->
</row>
</thead>

<tbody>
  <row>
    <entry>737</entry>
    <entry>root</entry>
    ...
    <entry>X</entry>
  </row>
  ...
</tbody>
</tgroup>
</table>
```

Table 9-1. DocBook approximation of a DIN conforming table

Identification		Scheduling		Memory			Stat	Resources			Com- mand
PID	User	Pri	Ni	Size	RSS	Share	%CPU	%Mem	Time		
737	root	15	0	56644	55M	1556	S	0.9	22.1	0:07	X

Identification		Scheduling		Memory			Stat	Resources			Com- mand
PID	User	Pri	Ni	Size	RSS	Share	%CPU	%Mem	Time		
1675	cspiel	13	0	1752	1752	1424	S	0.5	0.6	0:00	xis- dnload
1973	lvandijk	14	0	1176	1176	988	R	0.5	0.4	0:00	top
2	root	10	0	0	0	0	SW	0.1	0.0	0:00	kswapd
1656	cspiel	10	0	1356	1356	1108	S	0.1	0.5	0:00	bbsload
1764	lvandijk	18	0	6380	6380	3112	S	0.1	2.4	0:09	emacs

Figure 9-1. Table formatted according to DIN 55301

Identification		Scheduling		Memory			Stat	Resources			Command
PID	User	Pri	Ni	Size	RSS	Share		%CPU	%Mem	Time	
737	root	15	0	56644	55M	1556	S	0.9	22.1	0:07	X
1675	cspiel	13	0	1752	1752	1424	S	0.5	0.6	0:00	xisdnload
1973	lvandijk	14	0	1176	1176	988	R	0.5	0.4	0:00	top
2	root	10	0	0	0	0	SW	0.1	0.0	0:00	kswapd
1656	cspiel	10	0	1356	1356	1108	S	0.1	0.5	0:00	bbsload
1764	lvandijk	18	0	6380	6380	3112	S	0.1	2.4	0:09	emacs

The output of the **top(1)** command formatted according to DIN 55301.

9.2.3. Examples

Examples within the main text, should be as short as possible, retaining their completeness. If an author likes to give the complete version of an example, she is encouraged to do so by including it in the Appendix. <xref>s should be added in the main text and in the Appendix to allow for easy moving between the example's abridged and the full version.

Particularly interesting code can go into the Appendix, without having a short equivalent in the running text. No cross-references are necessary in this case.

If the Scilab source of an example is to be included in the file `scibot-examples`, its listing must be marked up in a special way.

- <programlisting> *must* contain the two attributes `role`, and `id`.

- `role` must be `scilab`.
- `id` must end in `.sci`. The `id` will be the filename of the example when stored in the archive.

Hints for preparing examples. Our policy forbids the use of CDATA marked sections in sci-BOT. Therefore, all examples are contained in `<programlisting>` elements, where the usual DocBook formatting rules are active.

- Relational operators containing “<” or “>” must be written as `<` or `>` respectively. Note that this also holds for digraph operators like “<=”, and “>=”, or anything else – think of comments – that contain “<” or “>”.
- Logical and, “&”, must be written as `&`.
- Additional comments to the code should be wrapped in `<lineannotation>` elements. These will be stripped off when generating `scibot-examples`.

9.2.4. Graphics

The inclusion of graphics is a relatively labor intensive job, despite the use of a publishing system that does well support graphics. (The future is so bright, we’ve gotta wear shades.) So, use it sparingly.

Graphics are wrapped in `<mediaobject>`s. Graphics must always be supplied in at least Encapsulated PostScript® and in Portable Network Graphics (png) formats. Providing additional formats is up to the author, however gif-encoded images are excluded from sci-BOT because of copyright issues. A `<textobject>` holding the textual description of the image is as mandatory as an image caption. A complete definition looks like this:

Example 9-2. Inclusion of graphics

```
<mediaobject>
  <imageobject>
    <imagedata fileref = "graphic.eps" format = "EPS"></imagedata>
  </imageobject>

  <imageobject>
    <imagedata fileref = "graphic.png" format = "PNG"></imagedata>
  </imageobject>

  <textobject>
    <!-- textual description of the image -->
    <phrase>...</phrase>
  </textobject>

  <caption>
    <para>
      ...
```

```

        </para>
    </caption>
</mediaobject>

```

In the example we have assumed that there are two graphics files named `graphic.eps`, and `graphic.png`. If the author wants an image or set of images go into the List of Figures, the whole `<mediaobject>` can be wrapped in `<figure>`.

9.2.5. Mathematics

In the current version of sci-BOT, the use of MathML is not allowed, but as soon as a wider variety of browsers will support MathML, it will be recommended. At present, the only mathematical notation which can be encoded directly in DocBook are superscripts (`<superscript>`), and subscripts (`<subscript>`). All “higher” mathematics is encoded with LaTeX as described below.

Equations are treated as graphics (see Section 9.2.4), where the filenames of the graphics files have a special meaning, the `textobject` has a special format, and the `caption` is missing.

The `textobject` must contain exactly one phrase element with attribute `role` having the value “formula”, and the attribute `id` being the LaTeX-filename of the formula.

Example 9-3. Inclusion of mathematics

```

<informalequation id = "eq-diff">
  <mediaobject>
    <imageobject>
      <imagedata fileref = "eq-diff.eps" format = "EPS"></imagedata>
    </imageobject>

    <imageobject>
      <imagedata fileref = "eq-diff.png" format = "PNG"></imagedata>
    </imageobject>

    <textobject>
      <phrase role = "formula" id = "eq-diff.tex">
        \[
          \frac{df}{dx}(x_0) :=
          \lim_{x \rightarrow x_0} \frac{f(x) - f(x_0)}{x -
x_0},
          \]
      </phrase>
    </textobject>
  </mediaobject>
</informalequation>

```

Whenever a formula is set up like this, the build system automatically converts the LaTeX formula in the `phrase` to the required eps and png files.

9.2.6. Index terms

Index term start tags `<indexterm>` can carry a `role` attribute that further specifies which kind of term gets indexed. Marking up `<indexterm>`'s this way is *not* required. Currently only Texinfo can handle these `role` attributes.

Attribute Value	Index Class	Candidate Elements
c	A concept index listing concepts that are discussed.	various
f	A function index listing functions.	funcsynopsis, function
v	A variables index listing variables.	classname, constant, structfield, structname, symbol, token, type, varname
k	A keystroke index listing keyboard and mouse commands.	accel, action, guibutton, guiicon, guilabel, guimenu, guimenuitem, guisubmenu, keycap, keycode, keycombo, keysym, menuchoice, mousebutton, shortcut
p	A program index listing names of programs.	application, command

Chapter 10. Complete Examples

Welcome to our attic! Following the style of the bag-of-tricks, the examples gathered here are an unsorted collection of hacks that has piled up over the years. A few functions are used or discussed in the earlier section, but were truncated to emphasized the important parts. Here you only find complete versions. All programs in this Appendix are available in a single tar or zip file; see Section 3 for details.

These example programs are free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License at the end of this document for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

10.1. `frac.sci`

`frac.sci` implements a rather complete class of fractions which are based on floating point numbers.

```
// name:          frac.sci - a class of fractions implemented
//                                     with operator overloading

//
// The names 'gcd', 'lcm', and 'qr' are already occupied
// by Scilab, so we had to invent new ones.  ;-)
//

function w = gcd_int(u, v)
// gcd of _positive_ u and v! See e.g.: Knuth, vol2, p337
while v ~= 0
    r = modulo(u, v)
    u = v
    v = r
end
w = u

function [p_red, q_red] = reduce_int(p, q)
// reduce fraction p/q and return reduced fraction p_red/q_red as vector
```

```

if q == 0, error('not a fraction'), end
r = gcd_int(abs(p), abs(q))
if q < 0 then
    r = -r // force positive denominator
end
[p_red, q_red] = (p/r, q/r)

function assert_int(p)
if type(p) ~= 1 | p ~= int(p) | imag(p) ~= 0 then
    error('assertion failed: non-integral or non-real p = ' + string(p))
end

function f = frac(p, q, reduce)
// constructor for fractions
//
// p is the numerator, q is the denominator. If q is
// omitted, 1 is assumed. The boolean reduce controls whether
// p/q will be reduced. If reduce is omitted or %t the p/q
// will be reduced.
//
// frac(int, int, bool = %t):    /* constructor */
// frac(2, 6)      -> 1/3
// frac(2)         -> 2/1 which is displayed as 2
// frac(2, 6, %t)  -> 1/3
// frac(2, 6, %f)  -> 2/6
//
// frac(frac, frac, bool = %t):  /* copy constructor */
// f = frac(1, 3);
// frac(f)         -> 1/3
// frac(f, f)      -> 1
// frac(1, f)      -> 3

select type(p)
case 1 then // constant
    if size(p, '*') ~= 1 then
        error('argument p is non-scalar')
    end
    p0 = p
    q0 = 1
case 16 then // tlist
    // copy constructor behavior
    p0 = p('num')
    q0 = p('denom')
else
    error('argument p has wrong type')
end
end

```

```

if exists('q', 'local') then // q is an optional argument
    select type(q)
    case 1 then // constant
        if size(q, '*') ~= 1 then
            error('argument q is non-scalar')
        end
        q0 = q0 * q
    case 16 then // tlist
        // copy constructor behavior
        p0 = p0 * q('denom')
        q0 = q0 * q('num')
    else
        error('argument q has wrong type')
    end
end

// ensure that arguments match
assert_int(p0)
assert_int(q0)

if exists('reduce', 'local') then // (isdef('reduce') & re-
    duce == %t) does not work, for
        // Scilab performs a complete boolean evaluation
    if reduce == %t then
        [p_red, q_red] = reduce_int(p0, q0)
    else
        p_red = p0
        q_red = q0
    end
else
    [p_red, q_red] = reduce_int(p0, q0)
end
f = tlist(['frac'; 'num'; 'denom'], p_red, q_red)

function s = %frac_p(f)
// display function for fractions
s = string(f)
disp(s)

//
// comparison
//

function b = %frac_o_frac(f1, f2)
b = f1('num') == f2('num') & f1('denom') == f2('denom')

```

```

function b = %frac_n_frac(f1, f2)
b = ~%frac_o_frac(f1, f2)

function b = %frac_o_s(f, s)
assert_int(s)
b = %frac_o_frac(f, frac(s))

function b = %s_o_frac(s, f)
assert_int(s)
b = %frac_o_s(f, s)

function b = %frac_n_s(f, s)
assert_int(s)
b = ~%frac_n_frac(f, frac(s))

function b = %s_n_frac(s, f)
assert_int(s)
b = %frac_n_s(f, s)

function b = %frac_1_frac(f1, f2)
b = f1('num')*f2('denom') < f1('denom')*f2('num')

function b = %frac_2_frac(f1, f2)
b = f1('num')*f2('denom') > f1('denom')*f2('num')

function b = %frac_3_frac(f1, f2)
// <=
b = %frac_1_frac(f1, f2) | %frac_o_frac(f1, f2)

function b = %frac_4_frac(f1, f2)
// >=
b = %frac_2_frac(f1, f2) | %frac_o_frac(f1, f2)

function b = %frac_1_s(f, s)
assert_int(s)
b = %frac_1_frac(f, frac(s))

function b = %s_1_frac(s, f)
assert_int(s)

```

```

b = %frac_1_frac(frac(s), f)

function b = %frac_2_s(f, s)
assert_int(s)
b = %frac_2_frac(f, frac(s))

function b = %s_2_frac(s, f)
assert_int(s)
b = %frac_2_frac(frac(s), f)

function b = %frac_3_s(f, s)
assert_int(s)
b = %frac_3_frac(f, frac(s))

function b = %s_3_frac(s, f)
assert_int(s)
b = %frac_3_frac(frac(s), f)

function b = %frac_4_s(f, s)
assert_int(s)
b = %frac_4_frac(f, frac(s))

function b = %s_4_frac(s, f)
assert_int(s)
b = %frac_4_frac(frac(s), f)

//
// addition/subtraction
//

function r = %frac_a_frac(f1, f2)
d1 = gcd_int(f1('denom'), f2('denom'))
if d1 == 1 then
    r = frac(f1('num')*f2('denom') + f1('denom')*f2('num'), ..
            f1('denom')*f2('denom'))
else
    t = f1('num')*(f2('denom') / d1) + f2('num')*(f1('denom') / d1)
    d2 = gcd_int(t, d1)
    r = frac(t/d2, (f1('denom') / d1)*(f2('denom') / d2))
end

```

```

function r = %frac_s_frac(f1, f2)
d1 = gcd_int(f1('denom'), f2('denom'))
if d1 == 1 then
    r = frac(f1('num')*f2('denom') - f1('denom')*f2('num'), ..
            f1('denom')*f2('denom'))
else
    t = f1('num')*(f2('denom') / d1) - f2('num')*(f1('denom') / d1)
    d2 = gcd_int(t, d1)
    r = frac(t/d2, (f1('denom') / d1)*(f2('denom') / d2))
end

function r = %frac_s(f)
r = frac(-f('num'), f('denom'), %f) // do not reduce here

function r = %frac_a_s(f, s)
assert_int(s)
r = f + frac(s)

function r = %s_a_frac(s, f)
assert_int(s)
r = %frac_a_s(f, s)

function r = %frac_s_s(f, s)
assert_int(s)
r = f - frac(s)

function r = %s_s_frac(s, f)
assert_int(s)
r = frac(s) - f

//
// multiplication, division, power
//

function r = %frac_m_frac(f1, f2)
r = frac(f1('num')*f2('num'), f1('denom')*f2('denom'))

function r = %frac_r_frac(f1, f2)
r = frac(f1('num')*f2('denom'), f1('denom')*f2('num'))

function r = %frac_m_s(f, s)

```

```

assert_int(s)
r = frac(f('num')*s, f('denom'))

function r = %s_m_frac(s, f)
assert_int(s)
r = %frac_m_s(f, s)

function r = %frac_r_s(f, s)
assert_int(s)
r = frac(f('num'), f('denom')*s)

function r = %s_r_frac(s, f)
assert_int(s)
r = frac(f('denom')*s, f('num'))

function r = %frac_p_s(f, s)
assert_int(s)
r = frac(f('num')^s, f('denom')^s)

function r = %frac_abs(f)
r = frac(abs(f('num')), f('denom'), %f)

//
// conversion
//

function fl = frac2float(f)
// convert a fraction to a floating point number
fl = f('num') / f('denom')

function s = %frac_string(f)
// string( frac(...) )
if f('denom') == 1 then
    s = sprintf('%0f', f('num'))
else
    s = sprintf('%0f/%0f', f('num'), f('denom'))
end

//
// continued fraction functions (and their helper functions)
// See: Knuth, vol2, p356-359

```

```

//

function f = cfe2frac(cfe)
// *private function*
// convert the continued fraction expansion CFE to a floating point number F
// (recursive implementation)
select length(cfe)
case 0 then
    f = frac(0, 1, %f)
case 1 then
    f = frac(1, cfe(1), %f)
else
    q = cfe2frac( cfe(2:$) )
    f = 1 / (frac(cfe(1), 1, %f) + q)
end

function f = cfe2frac_it(cfe)
// *private function*
// convert the continued fraction expansion CFE to a floating point number F
// (iterative implementation)
if cfe == [] then
    f = frac(0, 1, %f)
else
    f = frac(1, cfe($), %f)
    for x = cfe($-1 : -1 : 1)
        f = 1 / (f + x)
    end
end

function cfe = contfrac(fl, eps)
// *private function*
// continued fraction expansion of floating point number FL with a
// maximum error of EPS.
// CAUTION: contfrac() only accepts numbers in the range 0 <= fl < 1!
if fl < 0 | fl >= 1, error('fl out of range'), end

if ~isdef('eps'), eps = sqrt(%eps), end

guard = 100 // maximum length of expansion
i = 0

cfe = []
if fl == 0, return, end
x = fl
while abs(1 - frac2float(cfe2frac(cfe))/fl) > eps & i < guard

```

```

        a = round(1 / x)
        cfe = [cfe a]
        x = 1/x - a
        i = i + 1
    end
    if i == guard then
        warning('could not achieve precision after ' ..
            + string(guard) + ' iterations')
    end
end

function f = float2frac(fl, eps)
// convert the floating point number FL to the fraction F
// with a maximum relative error of EPS
intpart = floor(fl)           // floor([3.33 -3.33]) -> [3 -4]
f = intpart + cfe2frac( contfrac(fl - intpart, eps) )

testfrac.sci provides a simple test-frame for the class of fractions describes above.

// name:          testfrac.sci - test fractions class

getf('frac.sci');

f = frac(2, 3);
g = frac(1, 3);
h = frac(-1, 3);
i = frac(5, 3);

//
// each of the following tests should give %t
//

frac(0) == 0
frac(1) == 1
frac(-1) == -1
frac(0, 1) == 0
frac(1, 1) == 1
frac(2, 2) == 1

f + g == 1
g + h == 0
f - (g - h) == 0

1 + f == i
f + 1 == i
1 - f == g

```

```

f - 1 == h

-g == h

3 * f == 2
f * 3 == 2

f / g == 2
f / 2 == g
2 / f == 3

9 * g^2 - 1 == 0

g < f
f > g
g <= f
f >= g
f < 1
f > 0
f <= 1
g >= 0

abs(g) == g
abs(h) == g

//
// continued fraction expansion
//

frac(8, 29) == cfe2frac([3 1 1 1 2]) // recursive implementation
frac(8, 29) == cfe2frac_it([3 1 1 1 2]) // iterative implementation

n = 10

// 0 <= x < 1 in this test
x = 1/%pi;
timer();
for d = 1:n
    eps = 10^(-d);
    c = contfrac(x, eps);
    f = cfe2frac(c);
    fl = frac2float(f);
    delta = abs( x - fl );
    if delta <= eps, passed = 'T'; else passed = 'F'; end;
    printf('%10.g      %20.16g      %c', eps, delta, passed);
end
printf('time: %f', timer());

```

```

// x should be larger than 1 or less than 0 in this test
x = %pi;
timer();
for d = 1:n
    eps = 10^(-d);
    fl = frac2float( float2frac(x, eps) );
    delta = abs( x - fl );
    if delta <= eps, passed = 'T'; else passed = 'F'; end;
    printf('%10.g    %20.16g    %c', eps, delta, passed);
end
printf('time: %f', timer());

x = -(1 + sqrt(5)) / 2;
timer();
for d = 1:n
    eps = 10^(-d);
    fl = frac2float( float2frac(x, eps) );
    delta = abs( x - fl );
    if delta <= eps, passed = 'T'; else passed = 'F'; end;
    printf('%10.g    %20.16g    %c', eps, delta, passed);
end
printf('time: %f', timer());

//
// Some power series
//

// geometric series
z = frac(1, 3);
for n = 1:20
    s = frac(1);
    q = z;
    for i = 1 : n-1
        s = s + q;
        q = q * z;
    end
    rhs = (1 - z^n) / (1 - z);
    if s == rhs, passed = 'T'; else passed = 'F'; end;
    disp(string(n) + ': ' + string(rhs) + '    ' + passed);
end;

// exponential sums
limit = 1e-8

s = frac(1);

```

```

q = frac(1, 2);
while frac2float(abs(s - 2)) > limit
    s = s + q;
    q = q / 2;
end;
if frac2float(abs(s - 2)) <= limit, disp(%t); else disp(%f); end;

s = frac(1);
q = frac(1, 2);
sgn = -1;
while frac2float(abs(s - frac(2, 3))) > limit
    s = s + q * sgn;
    sgn = -sgn;
    q = q / 2;
end;
if frac2float(abs(s - frac(2, 3))), disp(%t); else disp(%f); end;

```

10.2. benchmark.sci

This example shows a benchmark function that tries hard to do better than others. In the first step the timer resolution is determined. Next the function under test is executed in a loop and the time taken is estimated. This time in turn is used for the final test. The number of loop iterations is chosen according to the preliminary test. Finally, the median of the timings is returned.

```

function res = calibrate(max_len, n_avg, log_inc)
// determine the resolution of Scilab's built-in timer
// Return vector with measured timer resolution(s)

[nl, nr] = argn()
if nr <= 2, log_inc = 1.1, end
if nr <= 1, n_avg = 31, end
if nr == 0, max_len = 100000, end

r = []
n = 1
while n <= max_len
    //disp(n)
    tv = []
    iter = 0:n
    for k = 1:n_avg
        timer()
        for i = iter, end
            t = timer()
            tv = [tv; t]
        end
    end
    tv = sort(tv)

```

```

        r = [r; [n, tv($/2 + 1)]]
        n = log_inc * n
    end

    // xbascc(); plot2d(r(:,1), r(:,2), -1)

    delta = [r(:, 2); r($, 2)] - [r(1, 2); r(:, 2)]
    idx = find(delta > %eps)
    res = delta(idx)

    function tpl = benchmark()
    // return the time for one loop round trip

    verbose = %t
    min_test = 10 // minimum multiple of the timer
                  // resolution to run coarse test
    std_test = 200 // as min_test but for real test
    n_avg = 31 // number of samples to calculate median
    log_inc = 2.0 // logarithmic increment in coarse test

    // inquire timer properties
    disp('+++ calibrating timer')
    resol = calibrate()
    if size(resol, '*') <= 2 then
        error('calibration failed')
    end
    if resol(1) ~= resol(2) then
        warning('calibration botched; proceeding anyway...')
    end
    t_resol = resol(1);
    if verbose
        disp('timer resolution is ' + string(t_resol) + 's')
    end

    // rough estimate of time
    disp('+++ calibrating test')
    np = 1
    timer()
    my_expensive_test()
    t = timer()
    while t < min_test * t_resol
        //disp(np, t, min_test * t_resol)
        np = log_inc * np
        timer()
        for i = 1:np
            my_expensive_test()
        end
        t = timer()
    end
end

```

```

if verbose then
    disp('coarse, ' + string(np) + ' round trips in '
        + string(t) + 's')
end
if np == 1 then
    warning('slow procedure under test - time may be excessive')
end

// run real test
disp('+++ running test')
tc = t / np
ne = ceil(std_test * t_resol / tc)
if verbose then
    disp('fine, test will take about '
        + string(ceil(tc * ne * n_avg)) + 's')
end

r = []
for k = 1:n_avg
    timer()
    for i = 1:ne
        my_expensive_test()
    end
    t = timer()
    r = [r; t]
end
if verbose then
    disp('fine, ' + string(ne) + ' round trips in '
        + string(t) + 's')
end

// get median and return
r = sort(r)
//disp(r)
tpl = r($/2 + 1) / ne

function my_expensive_test()
    exact = -2.5432596188;
    z = abs( exact - intg(0, 2 * %pi, f) )

function y = my_cheap_test(x)
    y = x

```

10.3. listdiff.sci

`listdiff` returns the differences of two vectors in the style of the **diff**(1) command. It is a funny example of doing something completely non-numerical with Scilab.

```

function diff = listdiff(lst1, lst2, equ)
// listdiff() implements a diff(1) like Scilab-function
// for vectors.
// The caller can supply a boolean equ(x, y) function
// that will be used in all comparisons, otherwise
// operator '==' is used.
//
// RETURN VALUE
// 2-column vector describing the differences.
// Column 1 contains the element and column 2
// the element's index. A '+' in front of the
// index means: 'Extra element in lst2', a '-'
// means missing element in lst1.
//
// AUTHOR
// Christoph L. Spiel
//
// Copyright 1999, 2000 by Christoph Spiel
// listdiff is free software distributed under the terms
// of the GNU General Public License, version 2.
//!

[nl, nr] = argn(0);
select nr
case 0 then
    error('Too few arguments. Got 0, require 2 or 3.');
```

```

case 1 then
    error('Too few arguments. Got 1, require 2 or 3.');
```

```

case 2 then
    deff('b = equ(s1, s2)', 'b = s1 == s2');
```

```

case 3 then
    // caller supplied equ()
    if type(equ) ~= 13 then
        error('Function expected, got a ' + typeof(equ) + '.');
```

```

    end
else
    error('Too many arguments. Got ' + string(nr) + ' require 2 or 3.');
```

```

end

if type(lst1) ~= 1 & type(lst2) ~= 1 then
    // none of the lists is empty
    if type(lst1) ~= type(lst2) then
        error('Both lists must be of the same type, or be empty.');
```

```

    end
end

fuzz = 10;

diff = [];
```

```

n1 = size(lst1, 1);
n2 = size(lst2, 1);

// special cases

if n1 == 0 & n2 == 0, return, end

if n1 == 0 then
    p = 1 : n2;
    diff = [lst2, '+' + string(p')];
    return;
end

if n2 == 0 then
    p = 1 : n1;
    diff = [lst1, string(-p')];
    return;
end

// general case (neither list is empty)

i = 1;
j = 1;
while i <= n1 & j <= n2
    while i <= n1 & j <= n2
        if ~equ(lst1(i), lst2(j)), break, end
        i = i + 1;
        j = j + 1;
    end
    if i >= n1 | j >= n2, break, end

    icurs = i;
    while icurs <= min(n1, i+fuzz)
        if equ(lst1(icurs), lst2(j)), break, end
        icurs = icurs + 1;
    end
    if icurs <= n1 then
        if equ(lst1(icurs), lst2(j)) then
            // record element(s) missing from lst1
            for p = i : icurs-1
                this_diff = [lst1(p), string(-p)];
                diff = [diff; this_diff];
            end
            // re-sync
            i = icurs;
        end
    end
end
end

```

```

    jcurs = j;
    while jcurs <= min(n2, j+fuzz)
        if equ(lst1(i), lst2(jcurs)), break, end
        jcurs = jcurs + 1;
    end
    if jcurs <= n2 then
        if equ(lst1(i), lst2(jcurs)) then
            // record extra element(s) in lst2
            for p = j : jcurs-1
                this_diff = [lst2(p), '+' + string(p)];
                diff = [diff; this_diff];
            end
            // re-sync
            j = jcurs;
        end
    end
end
end

```

10.4. `whatis.sci`

`whatis.sci` defines `whatis`, a function that lists all information of a variable a user can access.

```

function rv = whatis(name_arr)
// NAME
//   whatis - listing of variables in extended format
//
// CALLING SEQUENCE
//   whatis()
//   whatis(name_arr)
//
// PARAMETER
//   name_arr : array of variables names
//
// DESCRIPTION
//   whatis returns a column-vector with the names,
//   types, and sizes of all local variables
//   (first form), or only of the variables whose
//   names (as strings!) are given in the matrix
//   name_arr (second form).
//
// EXAMPLES
//   whatis()
//   whatis('my_mat')
//   whatis(['foo' 'bar' 'baz'; 'foobar' 'morefoo' 'foobaz'])
//
// SEE ALSO

```

```

// who, whos
//
// AUTHORS
// Enrico Segre, Lydia van Dijk
//
// Copyright 1999, 2000 by Enrico Segre and Lydia van Dijk.
// whatis is free software distributed under the terms
// of the GNU General Public License, version 2.
//!

// LAST REVISION
// lvd, Fri Dec 3 01:01:45 UTC 1999

// TO DO/TO FIX
//
// - Accepting a regexp as an argument would be nice. This in turn
// leads to complete boolean expressions doing the variable selection
// resembling what the UNI* find utility does. Example:
// All vars ending in a 'v' that are complex and larger than
// 1000 words.
// - The behavior with undefined variables is unsatisfactory.

[nl, nr] = argn(0);
clear nl;
if nr == 0 then
    // no arguments -> take all variables like whos() does
    clear nr;
    name_arr = sort(who('get'));
end
clear nr;

if type(name_arr) ~= 10 then
    error('Expecting a string or an array of strings, got a ' ..
        + typeof(name_arr) + '!');
    return;
end

[namev, memv] = who('get'); // get memory usage of all local vars

// define isreal() for hypermatrices
deff('b = %hm_isreal(hm)', ..
    'if size(hm, "") == 0 then b = %t; ..
    else ..
        b = isreal(hm(1)); ..
    end');

deff('b = hm_isbool(hm)', ..
    'if size(hm, "") == 0 then ..

```

```

        b = %t; ..
    else ..
        b = type(hm(1)) == 4; ..
    end');

deff('b = hm_isstring(hm)', ..
    'if size(hm, "**") == 0 then ..
        b = %t; ..
    else ..
        b = type(hm(1)) == 10; ..
    end');

deff('b = hm_isint(hm)', ..
    'if size(hm, "**") == 0 then ..
        b = %t; ..
    else ..
        b = type(hm(1)) == 8; ..
    end');

rv = [];
for name = matrix(name_arr, 1, size(name_arr, '*')) do
    if isdef(name) then
        clear var;
        var = evstr(name);      // convert var's name back into var
        //
        // type classification
        //
        ty = type(var);          // type number
        select ty                // type 16 and 17 are not recognized
        case 16 then             // by the function typeof()
            tgenp = %f;           // we know the tlist's type for these
            lab = var(1);         // vector of labels
            select lab(1)         // 1st label defines the type
            case 'ar' then
                tnam = 'ARMAX process';
            case 'des' then
                tnam = 'descriptor system';
            case 'linpro' then
                tnam = 'linear programming data';
            case 'lss' then
                tnam = 'linear system';
            case 'r' then
                tnam = 'rational';
            case 'scs_tree' then
                tnam = 'SCICOS navigator data';
            case 'xxx' then
                tnam = 'SCICOS menu data';
            else
                tnam = 'generic tlist ' + lab(1);
        end
    end
end

```

```

        tgenp = %t;
    end // select lab(1)
case 17 then
    tnam = 'hypermatrix';
else
    tnam = typeof(var);           // type name, a string
end // select ty
if ty==1 | ty==2 | ty==5 | ty==17 then
    // boolean, string, integral, real, or complex,
    // possibly sparse matrix or hypermatrix (yuck!)
    if hm_isbool(var) then
        tnam = 'boolean ' + tnam;
    elseif hm_isstring(var) then
        tnam = 'string ' + tnam;
    elseif hm_isint(var) then
        tnam = 'int ' + tnam;
    else
        if isreal(var) then
            tnam = 'real ' + tnam;
        else
            tnam = 'complex ' + tnam;
        end
    end
end
tmp = name + ': ';
//
// size determination
//
if ty==1 | ty==2 | ty==4 | ty==5 | ty==8 | ty==10 | ty==17 then
    // any kind of matrix
    sz = size(var);               // var's dimensions
    tmp = tmp + string(sz(1));
    for j = 2:length(sz)
        tmp = tmp + 'x' + string(sz(j));
    end
    tmp = tmp + ' ';
elseif ty==16
    // user-defined aka generic tlist
    if tgenp then
        tmp = tmp + string(size(var(1), '*')) + ' element ';
    end
else
    // function, library, or other non-atomic object
end
//
// memory usage
//
i = find(namev == evstr('name')); // index of var's entry
tmp = tmp + tnam + ', ' + string(memv(i)) + ' words';

```

```

else
    tmp = "" + name + "' is not defined';
    warning('variable ' + tmp);
end
rv = [rv; tmp];
end

```

10.5. Auto-Determination of Precedence and Associativity

`assoc.sci`, `prec.sci`, and `parser.sci` are the scripts that determine the precedence and the associativity of the arithmetic Scilab operators. The results are used in Section 4.3.

10.5.1. `assoc.sci`

```

function a = assoc(oper, typ)
// Return the associativity a of
// operator oper which accepts type typ.
// oper can be a matrix of operators.
//
// typ can be 'n' for numeric, or 'b' for boolean.
// If typ is omitted, numeric is assumed.

[nl, nr] = argn()
if nr == 1 then
    typ = 'n'
end

select typ
case 'n' then
    args = string([1.1 1.2 1.5])
    deff('b = equal(x, y)', 'b = abs(x - y) < 1.2*%eps')
case 'b' then
    args = string(['%f' '%t' '%f'])
    deff('b = equal(x, y)', 'b = x == y')
else
    error('unknown type ' + typ)
end

a = []
for op = oper
    expr = '[' + args(1) + op
           + args(2) + op + args(3) + ', ' ..
           + '(' + args(1) + op + args(2) + ')'

```

```

        + op + args(3) + ',' ..
        + args(1) + op + '(' + args(2)
        + op + args(3) + ')]'
//disp(expr)
r = evstr(expr)
//disp(r)

if equal(r(2), r(3)) then
    a = [a 'non']
elseif equal(r(1), r(2)) then
    a = [a 'left']
elseif equal(r(1), r(3)) then
    a = [a 'right']
else
    error('could not determine associativity')
end
end
end

```

10.5.2. prec.sci

```

function p = prec(op1, op2)
// determine the relative precedence of operator op1 vs op2
// If operator op1 has a higher precedence than op2 then p = -1.
// In the opposite case p = 1. If both have the same precedence
// level p = 0

args = string([1.1 1.2 1.5])
deff('b = equal(x, y)', 'b = abs(x - y) < 1.2*%eps')

expr = ..
[' ..
    + args(1) + op1 + args(2) + op2 + args(3) + ',' ..
    + '(' + args(1) + op1 + args(2) + ')' + op2 + args(3) + ',' ..
    + args(1) + op1 + '(' + args(2) + op2 + args(3) ..
    + ')]'

//disp(expr)
r = evstr(expr)
//disp(r)

if equal(r(2), r(3)) then
    p = 0
elseif equal(r(1), r(2)) then
    p = -1
elseif equal(r(1), r(3)) then
    p = 1
else

```

```

        error('could not determine precedence level')
    end

function p = prec1(uop, op)
// determine what relative precedence the unary operator uop has
// with respect to operator op. The return values are like those
// of prec()

args = string([1.1 1.2])
//args = string([(1.1+0.9*i) (1.2-0.8*i)])
deff('b = equal(x, y)', 'b = abs(x - y) < 1.2*%eps')

expr = '[' + uop + args(1) + op + args(2) + ',' ..
        + '(' + uop + args(1) + ')' + op + args(2) + ']'

//disp(expr)
r = evstr(expr)
//disp(r)

if equal(r(1), r(2)) then
    p = -1
else
    p = 1
end

function p = lprec(op1, op2)
// determine relative precedence of the
// logical operators op1 and op2

v = ['%f' '%t']
for i = 1:2
    for j = 1:2
        for k = 1:2
            args = string([v(i) v(j) v(k)])
            expr = '[' ..
                + args(1) + op1 + args(2) + op2 + args(3) + ',' ..
                + '(' + args(1) + op1 + args(2) + ')' ..
                + op2 + args(3) + ',' ..
                + args(1) + op1 + '(' + args(2) + op2 + args(3) + ')]'

            //disp(expr)
            r = evstr(expr)
            //disp(r)

            if r(2) == r(3) then
                p = 0
            elseif r(1) == r(2) then
                p = -1
            else
                return
            end
        end
    end
end

```

```

elseif r(1) == r(3) then
    p = 1
    return
else
    error('could not determine precedence level')
end
end
end
end
end
end

```

10.5.3. parser.sci

```

// determine properties of Scilab's parser:
// associativity and precedence level of operators

getf('assoc.sci');
getf('prec.sci');

numop1 = ['+' '-'];
numop2 = ['+' '-' '*' '/' '\' '^' '.*' './' '.\' '.^'];
logop1 = ['~'];
logop2 = ['&' '|'];

// inquire associativity
an = assoc(numop2, 'n');
ab = assoc(logop2, 'b');

// figure out the relative precedence of binary numeric operators
pm2 = [];
for i = numop2
    row = [];
    for j = numop2
        row = [row prec(i, j)];
    end
    pm2 = [pm2; row];
end
[lev, idx] = sort( sum(pm2, 'r') );
lev = lev - min(lev) + 1; // minimum := 1

nop2 = numop2;
for op = numop1 // mark binary operators that have a unary twin
    patch = find(op == nop2);
    nop2(patch) = op + '/2';
end

relp2 = [string(lev); nop2(idx); an(idx)];

```

```

relp1 = [];
for i = numop1
    row = [];
    for j = numop2
        row = [row, prec1(i, j)];
    end
    hop = numop2(find(row > 0.5)); // operators with higher precedence
    minhop = 0;
    for op = hop
        minhop = max( minhop, find(relp2(:, 2) == op) );
    end
    // now minhop is the index of the lowest precedence binary operator
    // that has a higher precedence than the unary operator i, or 0 if
    // there is none
    if minhop == 0
        uop = evstr(relp2(1, 1)) + 1;
    else
        uop = evstr(relp2(minhop, 1)) - 1;
    end
    relp1 = [relp1; [string(uop), i+ '/1', 'right']];
end
//relp1

// Merge unary operators into matrix of binary operators
relp = [relp1; relp2];
[dummy, idx] = sort(evstr(relp(:, 1)));
relp(idx, :)

```

10.6. cat.sci

`cat.sci` defines the function `cat` which prints the source of a macro (function) if it is available. The argument-, type-, and size-checking part is used in Example 5-4.

```

function [res] = cat(macname)
// Print definition of function 'macname'
// if it has been loaded via a library.

// AUTHOR
//   Lydia van Dijk
//
//   Copyright 1999, 2000 by Lydia van Dijk.
//   cat is free software distributed under the terms
//   of the GNU General Public License, version 2.

[nl, nr] = argn(0);

```

```

if nr ~= 1 then
    error('Call with: cat(macro_name)');
end
if type(macname) ~= 10 then
    error('Expecting a string, got a ' + typeof(macname));
end
if size(macname, '*' ) ~= 1 then
    sz = size(macname);
    error('Expecting a scalar, got a ' ..
        + sz(1) + 'x' + sz(2) + ' matrix')
end

[res, err] = evstr(macname);
if err ~= 0 then
    select err
    case 4 then
        disp(macname + ' is undefined. ');
        return;
    case 25 then
        disp(macname + ' is a builtin function');
        return;
    else
        error('unexpected error', err);
    end // select err
end // err ~= 0

name = whereis(macname);
//disp('name = <' + name + '>');
if name == [] then
    disp(macname + ' is defined, but its definition is unaccessible');
    clear ans;
    return;
end

cont = string(evstr(name)); // path (1) and contents (2..$) of library
fpath = cont(1);
if part(fpath, 1:4) == 'SCI/' then
    fpath = SCI + '/' + part(fpath, 5:length(fpath));
end
fname = fpath + macname + '.sci';

[fh, err] = file('open', fname, 'old');
if err ~= 0 then
    error('Could not open file ' + fname, err);
end
text = read(fh, -1, 1, '(a)');
file('close', fh);
write(%io(2), text);

```

10.7. quadpack.sci

Here is the complete example from Section 7.3.3.1. Function `quadpack` loads, unloads, or queries the load status of a Scilab extension. In this case the extension is the famous QuadPack library.

The foremost goal in the design of `quadpack` was user friendliness. Therefore, we condensed the function's interface to its minimum, providing only three different actions:

`load`

Link library and glue code with Scilab; do nothing if the library/glue code already has been linked. Return the actual linkage status afterwards.

`unload`

Unload library and glue code; do nothing if the library/glue code was not linked with Scilab before. Return the actual linkage status afterwards.

`query`

Do nothing, but return the actual linkage status.

Additional goodies are that `quadpack` defaults to action `query` if the function's argument is omitted, that the case of the argument does not matter, and that a minimal prefix of the argument is enough to select the right action.

```
function state = quadpack(action)
// name:      quadpack.sci - load/unload QUADPACK or query
//                                     the load-status
// author:    Lydia van Dijk
// last rev.: Sat Mar 18 19:23:58 UTC 2000
// Scilab ver.: 2.5

// The variable 'quadpacklibs' is the *only* one that needs
// adjustment on a per-system basis. It is safe to leave
// all the other stuff untouched.

quadpacklibs = ['/site/src/netlib/quadpack/libquadpack-1.0.so', ..
                '/site/src/netlib/quadpack/intersci/libqpif-1.0.so']

//
// No user servicable parts below this line.
//

gateway = 'quadpack_gw' // name of the gateway function

// The order of the interface names in
// interfaces *MUST* be the same
// as in qptab in file 'quadpack-gw.c'!
interfaces = ['intals', 'intcau', 'intexc', 'intfou', ..
```

```

        'intgen', 'intgk', 'intinf', 'intosc', ..
        'intsm']

[lhs, rhs] = argn()
if rhs == 0 then
    action = 'query' – Default if no args
end

if rhs >= 2 then
    error('Too many arguments; expecting 0 or 1')
end
if type(action) ~= 10 then // 10 means string
    error('Expecting a string in argument 1')
end
if size(action, '*') ~= 1 then
    error('Expecting a scalar in argument 1')
end

action = code2str( abs(str2code(action)) ) – Convert to lowercase
[state, number] = c_link(gateway)

if strindex('query', action) == 1 then – Check prefix
    // do nothing
elseif strindex('load', action) == 1 then
    if state == %t then
        disp('already loaded; no action taken')
        return
    end
    addinter(quadpacklibs, gateway, interfaces)
elseif strindex('unload', action) == 1 then
    if state == %f then
        disp('not loaded; no action taken')
        return
    end
    ulink(number)
else
    error('Expecting "query", "load" or "unload" in arg 1')
end
state = c_link(gateway) – Always return actual status

```

Appendix A. GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330
Boston, MA 02111-1307
USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall

subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by

you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the

compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See *GNU Copyleft*.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Appendix B. GNU General Public License

Version 2, June 1991

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330
Boston, MA 02111-1307
USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation’s software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author’s protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors’ reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone’s free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU General Public License

Terms And Conditions For Copying, Distribution And Modification

0. APPLICABILITY¹

This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The “Program”, below, refers to any such program or work, and a “work based on the Program” means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term “modification”.) Each licensee is addressed as “you”.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. VERBATIM COPYING

You may copy and distribute verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

1. The titles of the sections have been added by the authors. They do not occur in the original GNU General Public License. Everything else has been copied in verbatim.

2. MODIFICATIONS

You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
- b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. DISTRIBUTION

You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

- a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

- b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
- c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. TERMINATION

You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. ACCEPTANCE

You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. REDISTRIBUTION

Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. CONSEQUENCES

If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. LIMITATIONS

If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and “any later version”, you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. AGGREGATION WITH INDEPENDENT WORKS

If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. NO WARRANTY

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. LIABILITY

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM

(INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Bibliography

- [Cameron:1996] Debra Cameron, Bill Rosenblatt, and Eric Raymond, *Learning GNU Emacs*, second edition, O'Reilly&Associates, 1996, 1-56592-152-6.
- [Fowler:1999] Martin Fowler, *Refactoring*, first edition, Addison Wesley, 1999, 0-20148-567-2.
- [Golub:1996] Gene H. Golub and Charles F. van Loan, *Matrix Computations*, third edition, The Johns Hopkins University Press, 1996, 0-8018-5413-X.
- [Gopen:1990] *American Scientist*, George D. Gopen and Judith A. Swan, "The Science of Scientific Writing", 78, 1990, 550-558.
- [Kernighan:1999] Brian W. Kernighan and Rob Pike, *The practice of programming*, Addison Wesley Longman, Inc., 1999, 0-201-61586-X.
- [Myers:EffCPP:1998] Scott Douglas Myers, *Effective C++: 50 specific ways to improve your programs and designs*, Addison Wesley Longman, Inc., 1998, 0-201-92488-9.
- [Myers:MoreEffCPP:1996] Scott Douglas Myers, *More Effective C++: 35 new ways to improve your programs and designs*, Addison Wesley Longman, Inc., 1996, 0-201-63371-X.
- [Strunk:1979] Wiliam Strunk, jr. and E. B. White, *The Elements of Style*, Third edition, Macmillan, 1979.
- [Wall:1996] Larry Wall, Tom Christiansen, and Randal L. Schwartz, *Programming Perl*, Second edition, O'Reilly&Associates, 1996, 1-56592-149-6.
- [Walsh:1999] Norman Walsh and Leonard Muellner, *DocBook: The Definitive Guide*, First edition, O'Reilly&Associates, 1999, 156592-580-7.

Index

Symbols

\$ constant, 104
 &
 (See overload, operator, &)
 & operator, 114
 >
 (See overload, operator, &#x27;)
 >=
 (See overload, operator, &#x3D;)
 <
 (See overload, operator, &#x27;)
 <>
 (See overload, operator, &#x27;#x26;#x27;)
 <=
 (See overload, operator, &#x3D;)
 ,
 (See overload, operator, ')
 ()
 (See overload, operator, ())
 *
 (See overload, operator, *)
 *.
 (See overload, operator, *.)
 +
 (See overload, operator, +)
 -
 (See overload, operator, -)
 .
 (See dot, decimal)
 .'
 '
 (See overload, operator, .')
 .*
 (See overload, operator, .*)
 .*.
 (See overload, operator, .*.)
 ./
 (See overload, operator, ./)
 ./ operator
 (See division, element wise)

(See multiplication, element wise)
 ./.
 (See overload, operator, ./.)
 .\
 (See overload, operator, .\
 .\
 (See overload, operator, .\
 .^
 (See overload, operator, .^)
 /
 (See overload, operator, /)
 ./.
 (See overload, operator, ./.)
 :
 (See overload, operator, :)
 : operator, 106
 danger with fractional reals, 32
 ==
 (See overload, operator, ==)
 geci , 195
 [,]
 (See overload, operator, [,])
 [;]
 (See overload, operator, [;])
 [] operator
 (See vector, construction)
 \
 (See overload, operator, \)
 \.
 (See overload, operator, \.)
 ^
 (See overload, operator, ^)
 |
 (See overload, operator, |)
 | operator
 (See & operator)
 ~
 (See overload, operator, ~)
 ~=
 (See overload, operator, ~=)

A

- abort
 - command
 - (See command, abort)
- Ada
 - pseudo
 - (See pseudo Ada)
- Ada extensions
 - (See subroutines, external, Ada)
- Aerosmith, 139
- apropos
 - command
 - (See command, apropos)
- apropos-command
 - (See command, apropos)
- arguments
 - named
 - (See parameters, named)
 - positional
 - (See parameters, positional)
- array ordering
 - Fortran-77, 126
- assignment
 - tuple, 75
- associativity
 - (See operator, precedence and associativity)

B

- benchmark
 - mirror functions , 124
- boolean
 - used as index, 61
- Borland C extensions
 - (See subroutines, external, Borland C)
- break
 - keyword
 - (See keyword, break)

C

- C extensions
 - (See subroutines, external, C)
- C++ extensions
 - (See subroutines, external, C++)
- calling convention
 - by-reference, 125
- canonicalization of Scilab scripts, 26
- case
 - keyword
 - (See keyword, case)
- clear
 - (See command, clear)
 - command
 - (See command, clear)
- clearglobal
 - (See command, clearglobal)
- clearing global variables
 - (See variable, clearing global)
- clearing local variables
 - (See variable, clearing local)
- clearing variables
 - (See variable, clearing)
- code
 - operand type, 52
 - operator, 53
- column-major ordering
 - (See array ordering, Fortran-77)
- command
 - abort, 47
 - apropos, 21 , 47
 - clear, 32 , 47
 - clearglobal, 31
 - exit, 47
 - help, 21 , 47
 - pause, 47
 - pwd, 47
 - quit, 47
 - resume, 47
 - return, 47
 - what, 47
 - while, 47
 - who, 47 , 90

- Commands, 47
- common pitfalls
 - (See pitfalls)
- compile sci-files, 91
- control structures
 - block structure, 44
 - choice of, 41
 - early return, 44
 - for, 41
 - if, 42
 - select, 42
 - while, 41
- conventions, typographic, 17
- CVS, 193

D

- Debugging
 - Scilab, 194
- desc-file
 - (See file, interface description)
- Dirac distribution, 86
- dispatch tables, 158
- division
 - element wise, 23
- do
 - keyword
 - (See keyword, do)
- DocBook, 15
- documentation
 - local, 195
 - Comm.ps, 195
 - Internals.ps, 195
 - Intro.ps, 195
 - Lmi.ps, 195
 - Manual.ps, 195
 - Metanet.ps, 195
 - Scicos.ps, 195
 - Signal.ps, 195
- dot
 - as member selector, 77
 - decimal, 23
- dynamic scope

- (See scope, dynamic)

E

- else
 - keyword
 - (See keyword, else)
- elseif
 - keyword
 - (See keyword, elseif)
- emacs
 - add missing last newline, 26
- enclosing scope
 - (See scope, enclosing)
- end
 - keyword
 - (See keyword, end)
- endfunction
 - keyword
 - (See keyword, endfunction)
 - (See keyword, function)
- environment variables
 - SCI, 74
- error
 - generation
 - API routines
 - (See Scilab, error handling)
- evaluation
 - boolean, 61
 - short-circuit
 - (See evaluation, boolean)
- examples
 - assoc.sci, 229
 - benchmark.sci, 220
 - cat.sci, 233
 - determination of precedence and associativity, 229
 - frac.sci, 209
 - listdiff.sci, 222
 - parser.sci, 232
 - prec.sci, 230
 - quadpack.sci, 235
 - testfrac.sci, 217

- whatis.sci, 225
- examples, complete, 209
- exit
 - command
 - (See command, exit)

F

- fatal error
 - (See Scilab, error handling, fatal errors)
- Fibonacci function, 42
- file
 - interface description, 122
- find
 - (See locate)
- for
 - keyword
 - (See keyword, for)
- formats, other of sci-BOT
 - (See sci-BOT, formats)
- Fortran-77 extensions
 - (See subroutines, external, Fortran-77)
- Fortran-9x
 - equivalent functions
 - all
 - (See function, builtin, and)
 - (See function, builtin, matrix)
 - any
 - (See function, builtin, or)
- Fortran-9x extensions
 - (See subroutines, external, Fortran-9x)
- function
 - builtin
 - whereis, 94
 - Dirac
 - (See Dirac distribution)
 - exec
 - used without parenthesis, 88
 - Fibonacci
 - (See Fibonacci function)

- Fortran-77
 - name mangling
 - (See name mangling, Fortran-77 function)
- getf
 - used without parenthesis, 87
- head, 25
- keyword
 - (See keyword, function)
 - (See keyword, function)
- lib, 90
- mirror and variants , 102
- predefined
 - genlib, 94
 - size of, 44
- functions, 79
 - API
 - error, 163
 - out, 164 , 165
 - as members of mlist, 88
 - as members of tlist, 88
 - as parameters, 86
 - as variables, 85
 - builtin
 - addinter, 123
 - and, 114
 - argn, 84
 - cumprod
 - (See function sum)
 - cumsum
 - (See function sum)
 - dec2hex, 72
 - deff, 79
 - diag, 109
 - emptystr, 110
 - error, 84
 - exec, 88
 - execstr, 39
 - exists, 83 , 90
 - eye, 108
 - find, 111
 - fort, 122
 - freq, 120
 - fsolve, 80 , 125

- getf, 90
- gsort, 115
- hex2dec, 72
- horner, 120
- iconvert, 69
- intg, 80 , 125
- link, 125 , 130
- linspace, 33 , 107
- load, 64
- logspace, 107
- matrix, 118
- max, 113
- mget, 64
- mgeti, 64
- min
 - (See function max)
- modulo, 62
- mput, 64
- norm, 87
- ones, 108
- optim, 86
- or
 - (See function and)
- plot2d, 82 , 87
- poly, 120
- prod
 - (See function sum)
- rand, 110
- read, 64
- save, 64 , 91
- size, 84 , 117
- sort
 - (See function gsort)
- sum, 100 , 114
- type, 39 , 84
- typename, 53
- zeros, 107
- bulletproof, 83
- call without parenthesis, 87
- defined online, 79
- gateway
 - (See dispatch tables)
- native
 - (See Scilab, native functions)

- nested, 80
- not working with integers, 65
- parameters
 - named, 82
 - optional, 83
- predefined
 - evstr, 84
 - linspace, 87
 - macrovar, 89
 - who, 39
 - xbasc, 87
- safe
 - (See functions, bulletproof)
- user defined, 79
 - parameter less, 79
 - without return value, 79
- without parameters, 82
- without return value, 81
- working with integers, 65

G

- gateway functions
 - (See dispatch tables)
- genlib
 - (See function, genlib)
- getf
 - (See function, getf)
- Glimpse, 194
- global variable attribute
 - (See variable attribute, global)
- global variables
 - (See variable, global)
- GNU
 - Free Documentation License (FDL), 237
 - General Public License (GPL), 243
- grep
 - (See Glimpse)

H

- help

- command
 - (See command, help)
- help-command
 - (See command, help)
- hyperlinks, 197

I

- identifier, length, 52 , 74
- if
 - keyword
 - (See keyword, if)
- index
 - boolean
 - (See boolean, used as index)
 - highest
 - (See \$ constant)
 - last
 - (See \$ constant)
- indexing
 - avoiding, 101
- INRIA, 197
- integers, 62
 - array concatenation, 67
 - as array index, 63
 - automatic conversion, 63
 - bitwise operations, 72
 - arrays, 73
 - booleans cast to, 68
 - in dada files, 64
 - int8 type on PPC systems, 72
 - missing literals, 62
 - mixed type comparisons, 70
 - mixed type expressions, 67
 - modular, 66
 - raised to a power, 68
 - vector-scalar comparison, 71
- interface
 - library
 - (See library, interface)
- interface description file
 - (See file, interface description)
- interface generator, automatic, 161

- intersci, 122 , 195
 - example Makefile, 123

J

- Johnson, Richard B., 15

K

- keyword
 - break, 47
 - case, 47
 - do, 47
 - else, 47
 - elseif, 47
 - end, 47
 - endfunction, 47 , 79
 - for, 47
 - function, 47 , 79
 - if, 47
 - select, 47
 - then, 47
- Keywords, 47

L

- ld
 - incremental linking, 131
- lexical scope
 - (See scope, lexical)
- lib
 - (See function, lib)
- libraries, 90
- library
 - interface, 158
- library (variable type)
 - (See type, library)
- line continuation
 - function head, 25
- link overhead
 - (See overhead, link)

- links
 - (See hyperlinks)
- list
 - extraction, 77
- local variable
 - (See variable, local)
- locate, 193

M

- matrix
 - column-major form, 106
 - construction, 107
 - flattened representation, 105
 - operations, 111
 - reshape, 105
 - shaping a
 - (See functions, builtin matrix)
- md5sum, 16
- missing integer literals
 - (See integers, missing literals)
- mlist
 - functions as elements of, 88
- multiplication
 - element wise, 100

N

- name mangling
 - Fortran-77 function, 125
- native functions
 - (See Scilab, native functions)
- newline
 - missing last, 26
- Newton root finder, 42
- nm, 125 , 129
- numbers
 - decimal
 - (See dot, decimal)

O

- online function definition
 - (See functions, defined online)
- operand
 - type code, 52
 - b, 52
 - c, 52
 - f, 52
 - hm, 53
 - i, 52
 - ip, 52
 - l, 52
 - lss, 53
 - m, 52
 - mc, 52
 - ml, 52
 - msp, 52
 - p, 52
 - ptr, 52
 - r, 53
 - s, 52
 - sp, 52
 - spb, 52
- operation
 - vectorized, 99
- operator
 - ', 100
 - *, 100
 - .*, 100
 - ./
 - (See division, element wise)
 - :, 102
 - code
 - (See code, operator)
 - colon
 - (See : operator)
 - overloading, 48
 - precedence and associativity, 58
 - logical operators, 60
 - numeric operators, 58
 - relational operators, 59
 - []
 - (See vector, construction)

- optimizing
 - Scilab
 - (See scilex, binaries, building optimized)
- overhead
 - link, 121
 - runtime, 121
- overload
 - disp, 53
 - operator
 - &, 53
 - >, 53
 - >=, 53
 - <, 53
 - <>, 53
 - <=, 53
 - ', 53
 - (), 53
 - *, 53
 - *., 53
 - +, 53
 - , 53
 - ., 53
 - .*, 53
 - .*, 53
 - ./, 53
 - ./., 53
 - .\, 53
 - .\., 53
 - .^, 53
 - /, 53
 - ./., 53
 - ., 53
 - ==, 53
 - [,], 53
 - [;], 53
 - \, 53
 - .\, 53
 - ^, 53
 - |, 53
 - ~, 53
 - ~=, 53
- overloading operators
 - (See operator, overloading)

P

- Parallel Scilab
 - home page, 197
- parameters
 - named
 - (See functions, named parameters)
 - optional
 - (See functions, optional parameters)
 - positional, 82
- pause
 - command
 - (See command, pause)
- performance, 99
 - building an optimized Scilab, 135
 - extending Scilab, 121
 - high-level operations, 99
 - avoiding indexing, 101
 - avoiding resizing, 101
 - built-in vector-/matrix-functions, 106
 - vectorized, 99
- pitfalls, 23
- plain old documentation
 - (See POD)
- plot2d
 - (See functions, builtin, plot2d)
- POD (Perl's plain old documentation), 15
- point, decimal
 - (See dot, decimal)
- polynomials
 - evaluation of, 119
- precedence
 - (See operator, precedence and associativity)
- programming style
 - (See style, programming)
- pseudo Ada, 139
 - type
 - AccessDataStackIndex, 140
 - AccessNatural, 140
 - AccessString, 140
 - Boolean, 139

- Character, 139
- Complex, 141 , 147
- ComplexFlag, 140
- ComplexMatrix, 147
- ComplexVector, 147
- ConstAccessNatural, 140
- ConstAccessString, 140
- DataStackIndex, 140
- Float, 139
- FloatMatrix, 147
- FloatVector, 147
- Integer, 139
- Natural, 139
- ParameterStackAddress, 140
- ParameterStackIndex, 140
- SimpleFunctionType, 154
- String, 139
- TypeString, 140
- types, 140
- pushing the limits, 135
- prototyping, 135
- pwd
 - command
 - (See command, pwd)

Q

- quit
 - command
 - (See command, quit)
- quotes
 - single vs. double, 41

R

- range
 - dangerous generation, 32
- resizing
 - avoiding, 101
- resume
 - command
 - (See command, resume)

- return
 - command
 - (See command, return)
- runscilab, 74
- runtime overhead
 - (See overhead, runtime)

S

- save
 - (See function, save)
- scalar product, 99
- SCI (environment variable), 74
- sci-BOT
 - cross referencing, 200
 - DocBook, 200 , 200
 - FIXME, 200
 - formats, 16
 - HTML, 16
 - PDF, 16
 - PS, 16
 - SGML, 16
 - FreeBSD Documentation Project Primer
 - for New Contributors , 200
 - indexing, 200
 - writing style, 199
- Scilab
 - API
 - checklhs, 148
 - CheckLhs (C), 175
 - checklhs (F77) , 166
 - checkrhs, 148
 - CheckRhs (C), 175
 - checkrhs (F77) , 165
 - CreateCVar (C), 186
 - CreateCVarFromPtr (C), 187
 - CreateData (C), 187
 - Createlist (C), 188
 - CreateListCVarFromPtr (C), 187
 - CreateListVarFromPtr (C), 188
 - CreateVar (C), 184
 - CreateVarFromPtr (C) , 185
 - cremat, 148

- cremat (F77), 174
- error, 148 , 154
- FreePtr (C), 186
- getexternal, 154
- getexternal (F77) , 171
- GetFuncPtr (C), 182
- GetListRhsCVar (C), 182
- GetListRhsVar (C), 182
- getmat, 148
- getmat (F77), 168
- GetMatrixDims (C), 181
- GetMatrixptr (C), 180
- GetRhsCVar (C), 181
- GetRhsVar (C), 179
- getrmat (F77), 169
- getrvect (F77 , 169
- getscalar, 154
- getscalar (F77) , 170
- GetType (C), 177
- getvect (F77), 170
- lhs, 148 , 154
- Lhs (C), 176
- lhs (F77), 167
- LhsVar (C), 183
- PExecSciFunction (C), 190
- ReadString (C), 191
- rhs, 148 , 154
- Rhs (C), 177
- rhs (F77), 167
- SciError (C), 190
- SciFunction (C), 192
- SciString (C), 192
- WriteMatrix (C), 189
- WriteString (C), 189
- C API, 174
- command-line option
 - comp, 91
- compiler to Fortran-77, 135
- coping with, 193
- core, 139
- debugging
 - (See debugging, Scilab)
 - (See Debugging, Scilab)
- enthusiasts
 - Mottelet, Stéphane, 197
 - Pinçon, Bruno, 197
 - Segre, Enrico, 197
- error handling, 163
 - fatal errors, 163
 - messages, 165
 - warnings, 164
- extending, 121
- FAQ, 21 , 24 , 197
- Fortran API, 165
- FTP site, 197
- further information, 193
- home page, 197
- internal data structure, 141
 - complex matrices, 141
 - data stack
 - (See Scilab, internal data structure, stack)
 - parameter stack
 - (See Scilab, internal data structure, stack)
 - stack, 141
- Introduction to Scilab (documentation) , 21
- native functions, 146
 - functionals, 153
 - simple, 146
- scripts
 - canonicalization
 - (See canonicalization of Scilab scripts)
- scilab shell script, 74
- scilex, 74
 - binaries, 194
 - building optimized, 135
 - debugging code, 194
 - optimized code, 194
 - profiling code, 194
 - starting, 74
- scope
 - dynamic, 28
 - enclosing, 27
 - lexical, 28
- scoping

- (See variable, scoping)
- Segre, Enrico, 23
- select
 - keyword
 - (See keyword, select)
- session
 - restart
 - persistent global variables, 31
- sorting
 - lexicographical, 115
- space
 - white
 - (See whitespace)
- spacing
 - emphasizing brackets, 36
 - in an expression, 35
 - indentation, 39
 - line breaks, 35
 - vertical, 37
- starting scilex
 - (See scilex, starting)
- style
 - control structures, 41
 - emphasizing brackets, 36
 - formatting, 35
 - Golden Rule, 41
 - indentation, 39
 - line breaks, 35
 - paradigms, 40
 - programming, 35
 - quotes, 41
 - spacing
 - (See style, formatting)
 - vertical-spacing, 37
- subroutines
 - external
 - Ada, 130
 - Borland C, 134
 - C, 127
 - C++, 128
 - compiling, 125
 - Fortran-77, 126
 - Fortran-9x, 126
 - Visual C++, 131

T

- then
 - keyword
 - (See keyword, then)
- tlist
 - functions as elements of, 88
- Torvalds, Linus, 44
- tuple
 - assignment
 - (See assignment, tuple)
- type
 - cast
 - implicit, 60
 - library, 94
 - operand code
 - (See operand, type code)
 - promotion
 - implicit
 - (See type, cast implicit)
- typographic conventions
 - (See conventions, typographic)

U

- unknown spots, 47

V

- variable
 - clearing, 32
 - global, 31 , 32
 - local, 32
 - global, 30
 - to pass function results, 30
 - local, 29
 - name length
 - (See identifier length)
 - scoping
 - local, 27

- shadowing, 27
- visibility rules
 - (See variable, scoping)
- variable attribute
 - global, 30
- variable types
 - integers, 62
 - int16, 62
 - int32, 62
 - int8, 62
 - uint16, 63
 - uint32, 63
 - uint8, 63
- variables
 - API
 - buf, 163
 - err2, 164
- vector
 - construction, 24
 - generation, 106
- vectorized, operation
 - (See operation, vectorized)
- Visual C++ extensions
 - (See subroutines, external, Visual C++)

W

- warnings
 - (See Scilab, error handling, warnings)
- what
 - command
 - (See command, what)
- whereis
 - (See function, whereis)
- while
 - command
 - (See command, while)
- whitespace
 - around dotted operator, 23
 - as column separator, 24
 - last newline, 26
- who
 - (See command, who)
 - command
 - (See command, who)

